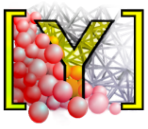


Yade Documentation

Václav Šmilauer, Emanuele Catalano, Bruno Chareyre, Sergei Dorofeenko, Jerome Duriez, Anton Gladky, Janek Kozicki, Chiara Modenese, Luc Scholtès, Luc Sibille, Jan Stránský, Klaus Thoeni

Release 1.20.0, November 14, 2015

Authors

Václav Šmilauer
Freelance consultant (http://woodem.eu)

Emanuele Catalano
Grenoble INP, UJF, CNRS, lab. 3SR

Bruno Chareyre
Grenoble INP, UJF, CNRS, lab. 3SR

Sergei Dorofeenko
IPCP RAS, Chernogolovka

Jerome Duriez
Grenoble INP, UJF, CNRS, lab. 3SR

Anton Gladky
TU Bergakademie Freiberg

Janek Kozicki
Gdansk University of Technology - lab. 3SR Grenoble University

Chiara Modenese
University of Oxford

Luc Scholtès
Grenoble INP, UJF, CNRS, lab. 3SR

Luc Sibille
University of Nantes, lab. GeM

Jan Stránský
CVUT Prague

Klaus Thoeni The University of Newcastle (Australia)

Citing this document

In order to let users cite Yade consistently in publications, we provide a list of bibliographic references for the different parts of the documentation. This way of acknowledging Yade is also a way to make developments and documentation of Yade more attractive for researchers, who are evaluated on the basis of citations of their work by others. We therefore kindly ask users to cite Yade as accurately as possible in their papers, as explained in http://yade-dem/doc/citing.html.

Contents

Chapter 1

Introduction

1.1 Getting started

Before you start moving around in Yade, you should have some prior knowledge.

- Basics of command line in your Linux system are necessary for running yade. Look on the web for tutorials.
- Python language; we recommend the official Python tutorial. Reading further documents on the topis, such as Dive into Python will certainly not hurt either.

You are advised to try all commands described yourself. Don't be afraid to experiment.

1.1.1 Starting yade

Yade is being run primarily from terminal; the name of command is `yade`. [1] (In case you did not install from package, you might need to give specific path to the command [2]):

```
$ yade
Welcome to Yade
TCP python prompt on localhost:9001, auth cookie `sdksuy'
TCP info provider on localhost:21000
[[ ^L clears screen, ^U kills line. F12 controller, F11 3d view, F10 both, F9 generator, F8 plot. ]]
Yade [1]:
```

These initial lines give you some information about

- some information for *Remote control*, which you are unlikely to need now;
- basic help for the command-line that just appeared (`Yade [1]:`).

Type `quit()`, `exit()` or simply press `^D` to quit Yade.

[1] The executable name can carry a suffix, such as version number (`yade-0.20`), depending on compilation options. Packaged versions on Debian systems always provide the plain `yade` alias, by default pointing to latest stable version (or latest snapshot, if no stable version is installed). You can use `update-alternatives` to change this.

[2] In general, Unix *shell* (command line) has environment variable `PATH` defined, which determines directories searched for executable files if you give name of the file without path. Typically, $PATH contains `/usr/bin/`, `/usr/local/bin`, `/bin` and others; you can inspect your `PATH` by typing `echo $PATH` in the shell (directories are separated by `:`).

If Yade executable is not in directory contained in `PATH`, you have to specify it by hand, i.e. by typing the path in front of the filename, such as in `/home/user/bin/yade` and similar. You can also navigate to the directory itself (`cd ~/bin/yade`, where `~` is replaced by your home directory automatically) and type `./yade` then (the `.` is the current directory, so `./` specifies that the file is to be found in the current directory).

To save typing, you can add the directory where Yade is installed to your `PATH`, typically by editing `~/.profile` (in normal cases automatically executed when shell starts up) file adding line like `export PATH=/home/user/bin:$PATH`. You can also define an *alias* by saying `alias yade="/home/users/bin/yade"` in that file.

Details depend on what shell you use (bash, zsh, tcsh, …) and you will find more information in introductory material on Linux/Unix.

The command-line is ipython, python shell with enhanced interactive capabilities; it features persistent history (remembers commands from your last sessions), searching and so on. See ipython's documentation for more details.

Typically, you will not type Yade commands by hand, but use *scripts*, python programs describing and running your simulations. Let us take the most simple script that will just print "Hello world!":

```
print "Hello world!"
```

Saving such script as `hello.py`, it can be given as argument to yade:

```
$ yade hello.py
Welcome to Yade
TCP python prompt on localhost:9001, auth cookie `askcsu'
TCP info provider on localhost:21000
Running script hello.py                                          ## the script is being run
Hello world!                                                     ## output from the script
[[ ^L clears screen, ^U kills line. F12 controller, F11 3d view, F10 both, F9 generator, F8 plot. ]]
Yade [1]:
```

Yade will run the script and then drop to the command-line again. [3] If you want Yade to quit immediately after running the script, use the `-x` switch:

```
$ yade -x script.py
```

There is more command-line options than just `-x`, run `yade -h` to see all of them.

Options:

--version show program's version number and exit

-h, --help show this help message and exit

-j THREADS, --threads=THREADS Number of OpenMP threads to run; defaults to 1. Equivalent to setting OMP_NUM_THREADS environment variable.

--cores=CORES Set number of OpenMP threads (as –threads) and in addition set affinity of threads to the cores given.

--update Update deprecated class names in given script(s) using text search & replace. Changed files will be backed up with ~ suffix. Exit when done without running any simulation.

--nice=NICE Increase nice level (i.e. decrease priority) by given number.

-x Exit when the script finishes

-n Run without graphical interface (equivalent to unsetting the DISPLAY environment variable)

--test Run regression test suite and exit; the exists status is 0 if all tests pass, 1 if a test fails and 2 for an unspecified exception.

--check Run a series of user-defined check tests as described in /build/buildd/yade-daily-1+3021+27~lucid1/scripts/test/checks/README

--performance Starts a test to measure the productivity

--no-gdb Do not show backtrace when yade crashes (only effective with –debug).

[3] Plain Python interpreter exits once it finishes running the script. The reason why Yade does the contrary is that most of the time script only sets up simulation and lets it run; since computation typically runs in background thread, the script is technically finished, but the computation is running.

1.1.2 Creating simulation

To create simulation, one can either use a specialized class of type *FileGenerator* to create full scene, possibly receiving some parameters. Generators are written in c++ and their role is limited to well-defined scenarios. For instance, to create triaxial test scene:

```
Yade [70]: TriaxialTest(numberOfGrains=200).load()

Yade [71]: len(O.bodies)
Out[71]: 206
```

Generators are regular yade objects that support attribute access.

It is also possible to construct the scene by a python script; this gives much more flexibility and speed of development and is the recommended way to create simulation. Yade provides modules for streamlined body construction, import of geometries from files and reuse of common code. Since this topic is more involved, it is explained in the *User's manual*.

1.1.3 Running simulation

As explained below, the loop consists in running defined sequence of engines. Step number can be queried by `O.iter` and advancing by one step is done by `O.step()`. Every step advances *virtual time* by current timestep, `O.dt`:

```
Yade [72]: O.iter
Out[72]: 0

Yade [73]: O.time
Out[73]: 0.0

Yade [74]: O.dt=1e-4

Yade [75]: O.step()

Yade [76]: O.iter
Out[76]: 1

Yade [77]: O.time
Out[77]: 0.0001
```

Normal simulations, however, are run continuously. Starting/stopping the loop is done by `O.run()` and `O.pause()`; note that `O.run()` returns control to Python and the simulation runs in background; if you want to wait for it finish, use `O.wait()`. Fixed number of steps can be run with `O.run(1000)`, `O.run(1000,True)` will run and wait. To stop at absolute step number, `O.stopAtIter` can be set and `O.run()` called normally.

```
Yade [78]: O.run()

Yade [79]: O.pause()

Yade [80]: O.iter
Out[80]: 1672

Yade [81]: O.run(100000,True)

Yade [82]: O.iter
Out[82]: 101672

Yade [83]: O.stopAtIter=500000

Yade [84]: O.wait()
```

```
Yade [85]: O.iter
Out[85]: 101672
```

1.1.4 Saving and loading

Simulation can be saved at any point to a binary file (optionaly compressed if the filename has extensions such as ".gz" or ".bz2"). Saving to a XML file is also possible though resulting in larger files and slower save/load, it is used when the filename contains "xml". With some limitations, it is generally possible to load the scene later and resume the simulation as if it were not interrupted. Note that since the saved scene is a dump of Yade's internal objects, it might not (probably will not) open with different Yade version.

```
Yade [86]: O.save('/tmp/a.yade.bz2')

Yade [87]: O.reload()

Yade [88]: O.load('/tmp/another.yade.bz2')
```

The principal use of saving the simulation to XML is to use it as temporary in-memory storage for checkpoints in simulation, e.g. for reloading the initial state and running again with different parameters (think tension/compression test, where each begins from the same virgin state). The functions `O.saveTmp()` and `O.loadTmp()` can be optionally given a slot name, under which they will be found in memory:

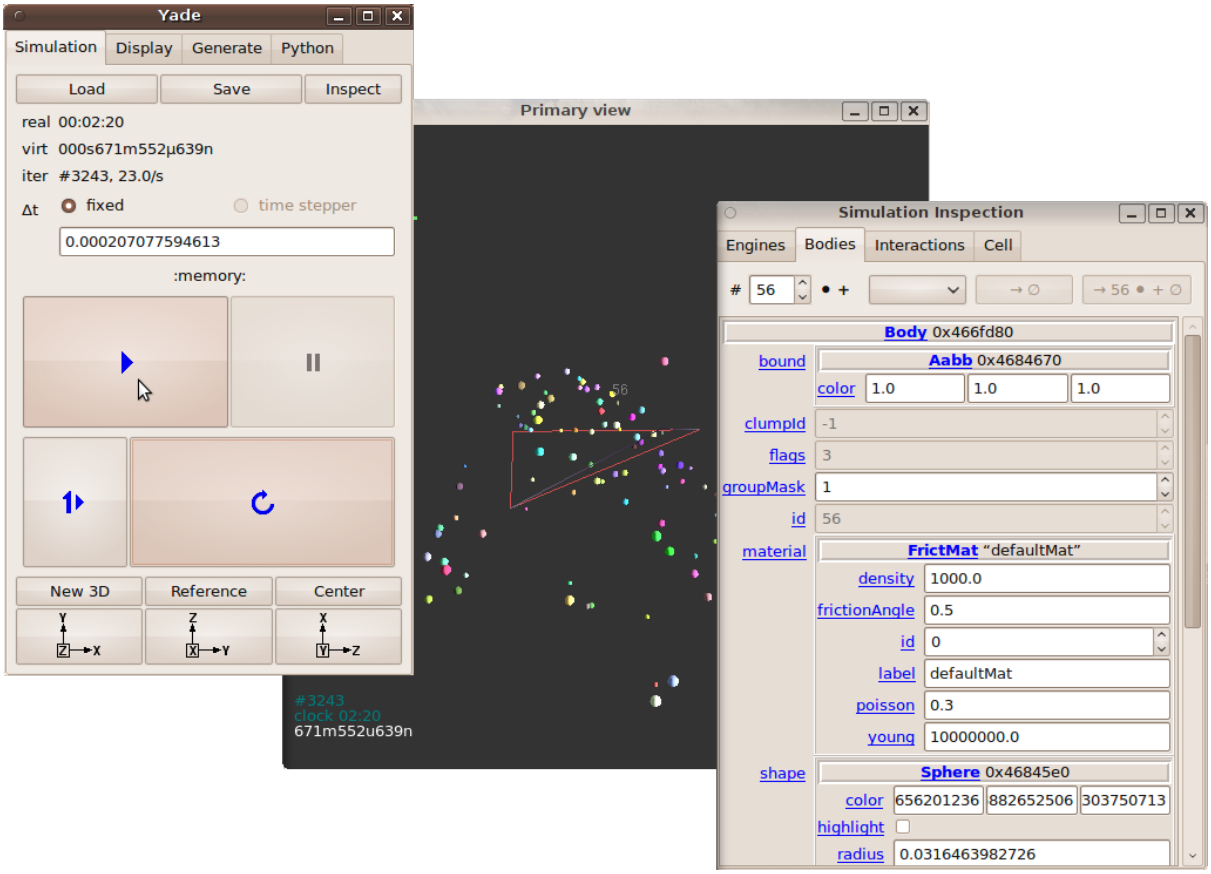
```
Yade [89]: O.saveTmp()

Yade [90]: O.loadTmp()

Yade [91]: O.saveTmp('init') ## named memory slot

Yade [92]: O.loadTmp('init')
```

Simulation can be reset to empty state by `O.reset()`.

It can be sometimes useful to run different simulation, while the original one is temporarily suspended, e.g. when dynamically creating packing. `O.switchWorld()` toggles between the primary and secondary simulation.

1.1.5 Graphical interface

Yade can be optionally compiled with qt4-based graphical interface. It can be started by pressing `F12` in the command-line, and also is started automatically when running a script.



The windows with buttons is called `Controller` (can be invoked by `yade.qt.Controller()` from python):

1. The *Simulation* tab is mostly self-explanatory, and permits basic simulation control.
2. The *Display* tab has various rendering-related options, which apply to all opened views (they can be zero or more, new one is opened by the *New 3D* button).
3. The *Python* tab has only a simple text entry area; it can be useful to enter python commands while the command-line is blocked by running script, for instance.

3d views can be controlled using mouse and keyboard shortcuts; help is displayed if you press the `h` key while in the 3d view. Note that having the 3d view open can slow down running simulation significantly, it is meant only for quickly checking whether the simulation runs smoothly. Advanced post-processing is described in dedicated section.

1.2 Architecture overview

In the following, a high-level overview of Yade architecture will be given. As many of the features are directly represented in simulation scripts, which are written in Python, being familiar with this language will help you follow the examples. For the rest, this knowledge is not strictly necessary and you can ignore code examples.

1.2.1 Data and functions

To assure flexibility of software design, yade makes clear distinction of 2 families of classes: *data* components and *functional* components. The former only store data without providing functionality, while the latter define functions operating on the data. In programming, this is known as *visitor* pattern (as functional components "visit" the data, without being bound to them explicitly).

Entire simulation, i.e. both data and functions, are stored in a single `Scene` object. It is accessible through the *Omega* class in python (a singleton), which is by default stored in the `O` global variable:

```
Yade [93]: O.bodies       # some data components
Out[93]: <yade.wrapper.BodyContainer at 0x7fbe625d75f0>

Yade [94]: len(O.bodies)  # there are no bodies as of yet
Out[94]: 0

Yade [95]: O.engines      # functional components, empty at the moment
Out[95]: []
```

Data components

Bodies

Yade simulation (class `Scene`, but hidden inside *Omega* in Python) is represented by *Bodies*, their *Interactions* and resultant generalized *forces* (all stored internally in special containers).

Each *Body* comprises the following:

Shape represents particle's geometry (neutral with regards to its spatial orientation), such as *Sphere*, *Facet* or inifinite *Wall*; it usually does not change during simulation.

Material stores characteristics pertaining to mechanical behavior, such as Young's modulus or density, which are independent on particle's shape and dimensions; usually constant, might be shared amongst multiple bodies.

State contains state variable variables, in particular spatial *position* and *orientation*, *linear* and *angular* velocity, *linear* and *angular* accelerator; it is updated by the *integrator* at every step.

Derived classes can hold additional data, e.g. *averaged damage*.

Bound is used for approximate ("pass 1") contact detection; updated as necessary following body's motion. Currently, *Aabb* is used most often as *Bound*. Some bodies may have no *Bound*, in which case they are exempt from contact detection.

(In addition to these 4 components, bodies have several more minor data associated, such as *Body::id* or *Body::mask*.)

All these four properties can be of different types, derived from their respective base types. Yade frequently makes decisions about computation based on those types: *Sphere* + *Sphere* collision has to be treated differently than *Facet* + *Sphere* collision. Objects making those decisions are called *Dispatcher*'s and are essential to understand Yade's functioning; they are discussed below.

Explicitly assigning all 4 properties to each particle by hand would be not practical; there are utility functions defined to create them with all necessary ingredients. For example, we can create sphere particle using *utils.sphere*:

```
Yade [96]: s=utils.sphere(center=[0,0,0],radius=1)

Yade [97]: s.shape, s.state, s.mat, s.bound
Out[97]:
(<Sphere instance at 0x790ba10>,
 <State instance at 0x859d270>,
 <FrictMat instance at 0x80de340>,
 None)

Yade [98]: s.state.pos
Out[98]: Vector3(0,0,0)

Yade [99]: s.shape.radius
Out[99]: 1.0
```

We see that a sphere with material of type *FrictMat* (default, unless you provide another *Material*) and bounding volume of type *Aabb* (axis-aligned bounding box) was created. Its position is at origin and its radius is 1.0. Finally, this object can be inserted into the simulation; and we can insert yet one sphere as well.

```
Yade [100]: O.bodies.append(s)
Out[100]: 0

Yade [101]: O.bodies.append(utils.sphere([0,0,2],.5))
Out[101]: 1
```

In each case, return value is *Body.id* of the body inserted.

Since till now the simulation was empty, its id is 0 for the first sphere and 1 for the second one. Saving the id value is not necessary, unless you want access this particular body later; it is remembered internally in *Body* itself. You can address bodies by their id:

```
Yade [102]: O.bodies[1].state.pos
Out[102]: Vector3(0,0,2)

Yade [103]: O.bodies[100]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/build/yade-Swrxdd/yade-1.20.0/debian/tmp/usr/lib/x86_64-linux-gnu/yade/py/yade/__init__.pyc in <module>()
----> 1 O.bodies[100]

IndexError: Body id out of range.
```

Adding the same body twice is, for reasons of the id uniqueness, not allowed:

```
Yade [104]: O.bodies.append(s)
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/build/yade-Swrxdd/yade-1.20.0/debian/tmp/usr/lib/x86_64-linux-gnu/yade/py/yade/__init__.pyc in <module>()
----> 1 O.bodies.append(s)

IndexError: Body already has id 0 set; appending such body (for the second time) is not allowed.
```

Bodies can be iterated over using standard python iteration syntax:

```
Yade [105]: for b in O.bodies:
   .....:    print b.id,b.shape.radius
   .....:
0 1.0
1 0.5
```

Interactions

Interactions are always between pair of bodies; usually, they are created by the collider based on spatial proximity; they can, however, be created explicitly and exist independently of distance. Each interaction has 2 components:

IGeom holding geometrical configuration of the two particles in collision; it is updated automatically as the particles in question move and can be queried for various geometrical characteristics, such as penetration distance or shear strain.

Based on combination of types of *Shapes* of the particles, there might be different storage requirements; for that reason, a number of derived classes exists, e.g. for representing geometry of contact between *Sphere+Sphere*, *Cylinder+Sphere* etc. Note, however, that it is possible to represent many type of contacts with the basic sphere-sphere geometry (for instance in *Ig2_Wall_Sphere_ScGeom*).

IPhys representing non-geometrical features of the interaction; some are computed from *Materials* of the particles in contact using some averaging algorithm (such as contact stiffness from Young's moduli of particles), others might be internal variables like damage.

Suppose now interactions have been already created. We can access them by the id pair:

```
Yade [106]: O.interactions[0,1]
Out[106]: <Interaction instance at 0xa031080>

Yade [107]: O.interactions[1,0]     # order of ids is not important
Out[107]: <Interaction instance at 0xa031080>

Yade [108]: i=O.interactions[0,1]

Yade [109]: i.id1,i.id2
Out[109]: (0, 1)

Yade [110]: i.geom
Out[110]: <ScGeom instance at 0x82d5870>

Yade [111]: i.phys
Out[111]: <FrictPhys instance at 0xc1f6470>

Yade [112]: O.interactions[100,10111]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/build/yade-Swrxdd/yade-1.20.0/debian/tmp/usr/lib/x86_64-linux-gnu/yade/py/yade/__init__.pyc in <module>()
----> 1 O.interactions[100,10111]

IndexError: No such interaction
```

Generalized forces

Generalized forces include force, torque and forced displacement and rotation; they are stored only temporarliy, during one computation step, and reset to zero afterwards. For reasons of parallel computation, they work as accumulators, i.e. only can be added to, read and reset.

```
Yade [113]: O.forces.f(0)
Out[113]: Vector3(0,0,0)

Yade [114]: O.forces.addF(0,Vector3(1,2,3))

Yade [115]: O.forces.f(0)
Out[115]: Vector3(1,2,3)
```

You will only rarely modify forces from Python; it is usually done in c++ code and relevant documentation can be found in the Programmer's manual.

Function components

In a typical DEM simulation, the following sequence is run repeatedly:

- reset forces on bodies from previous step
- approximate collision detection (pass 1)
- detect exact collisions of bodies, update interactions as necessary
- solve interactions, applying forces on bodies
- apply other external conditions (gravity, for instance).
- change position of bodies based on forces, by integrating motion equations.

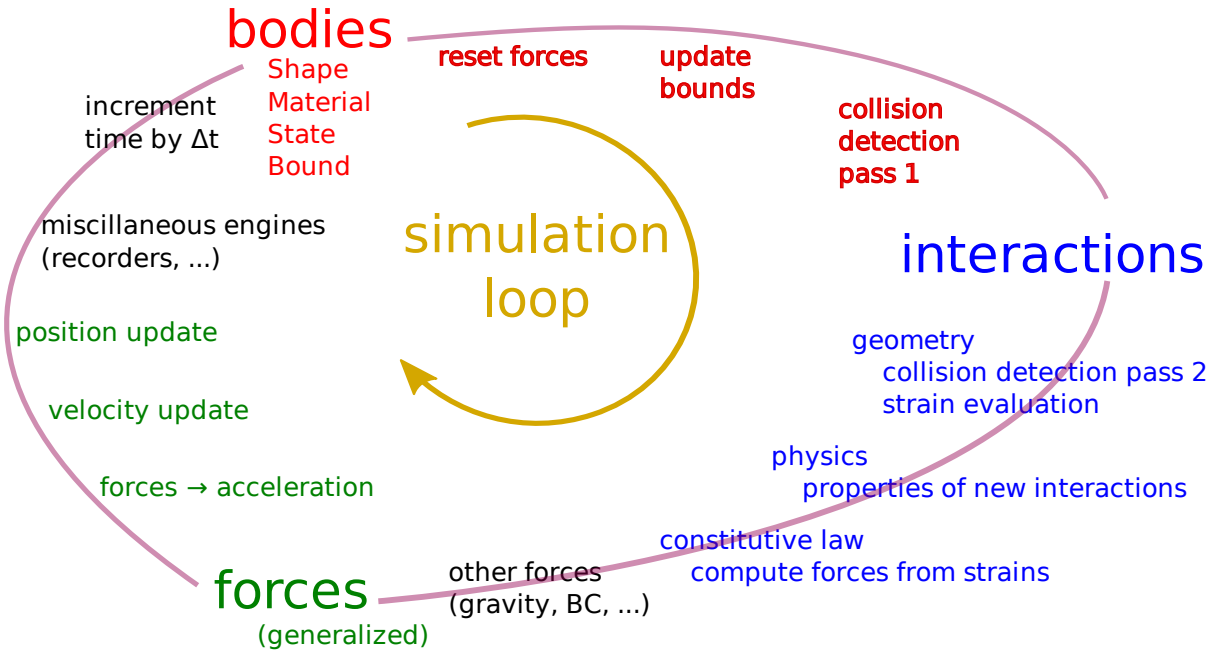


Fig. 1.1: Typical simulation loop; each step begins at body-centered bit at 11 o'clock, continues with interaction bit, force application bit, miscillanea and ends with time update.

Each of these actions is represented by an *Engine*, functional element of simulation. The sequence of engines is called *simulation loop*.

Engines

Simulation loop, shown at img. *img-yade-iter-loop*, can be described as follows in Python (details will be explained later); each of the `O.engine` items is instance of a type deriving from *Engine*:

```
O.engines=[
        # reset forces
        ForceResetter(),
        # approximate collision detection, create interactions
        InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb()]),
        # handle interactions
        InteractionLoop(
                [Ig2_Sphere_Sphere_ScGeom(),Ig2_Facet_Sphere_ScGeom()],
                [Ip2_FrictMat_FrictMat_FrictPhys()],
                [Law2_ScGeom_FrictPhys_CundallStrack()],
        ),
        # apply other conditions
        GravityEngine(gravity=(0,0,-9.81)),
        # update positions using Newton's equations
        NewtonIntegrator()
]
```

There are 3 fundamental types of Engines:

GlobalEngines operating on the whole simulation (e.g. *GravityEngine* looping over all bodies and applying force based on their mass)

PartialEngine operating only on some pre-selected bodies (e.g. *ForceEngine* applying constant force to some bodies)

Dispatchers do not perform any computation themselves; they merely call other functions, represented by function objects, *Functors*. Each functor is specialized, able to handle certain object types, and will be dispatched if such obejct is treated by the dispatcher.

Dispatchers and functors

For approximate collision detection (pass 1), we want to compute *bounds* for all *bodies* in the simulation; suppose we want bound of type *axis-aligned bounding box*. Since the exact algorithm is different depending on particular *shape*, we need to provide functors for handling all specific cases. The line:

```
InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb()])
```

creates *InsertionSortCollider* (it internally uses *BoundDispatcher*, but that is a detail). It traverses all bodies and will, based on *shape* type of each *body*, dispatch one of the functors to create/update *bound* for that particular body. In the case shown, it has 2 functors, one handling *spheres*, another *facets*.

The name is composed from several parts: `Bo` (functor creating *Bound*), which accepts `1` type *Sphere* and creates an *Aabb* (axis-aligned bounding box; it is derived from *Bound*). The *Aabb* objects are used by *InsertionSortCollider* itself. All `Bo1` functors derive from *BoundFunctor*.

The next part, reading

```
InteractionLoop(
        [Ig2_Sphere_Sphere_ScGeom(),Ig2_Facet_Sphere_ScGeom()],
        [Ip2_FrictMat_FrictMat_FrictPhys()],
        [Law2_ScGeom_FrictPhys_CundallStrack()],
),
```

hides 3 internal dispatchers within the *InteractionLoop* engine; they all operate on interactions and are, for performance reasons, put together:

IGeomDispatcher uses the first set of functors (`Ig2`), which are dispatched based on combination of `2` *Shapes* objects. Dispatched functor resolves exact collision configuration and creates *IGeom* (whence `Ig` in the name) associated with the interaction, if there is collision. The functor might as well fail on approximate interactions, indicating there is no real contact between the bodies, even if they did overlap in the approximate collision detection.

1. The first functor, *Ig2_Sphere_Sphere_ScGeom*, is called on interaction of 2 *Spheres* and creates *ScGeom* instance, if appropriate.
2. The second functor, *Ig2_Facet_Sphere_ScGeom*, is called for interaction of *Facet* with *Sphere* and might create (again) a *ScGeom* instance.

All `Ig2` functors derive from *IGeomFunctor* (they are documented at the same place).

IPhysDispatcher dispatches to the second set of functors based on combination of `2` *Materials*; these functors return return *IPhys* instance (the `Ip` prefix). In our case, there is only 1 functor used, *Ip2_FrictMat_FrictMat_FrictPhys*, which create *FrictPhys* from 2 *FrictMat's*.

`Ip2` functors are derived from *IPhysFunctor*.

LawDispatcher dispatches to the third set of functors, based on combinations of *IGeom* and *IPhys* (wherefore `2` in their name again) of each particular interaction, created by preceding functors. The `Law2` functors represent "constitutive law"; they resolve the interaction by computing forces on the interacting bodies (repulsion, attraction, shear forces, …) or otherwise update interaction state variables.

`Law2` functors all inherit from *LawFunctor*.

There is chain of types produced by earlier functors and accepted by later ones; the user is responsible to satisfy type requirement (see img. *img-dispatch-loop*). An exception (with explanation) is raised in the contrary case.

Note: When Yade starts, O.engines is filled with a reasonable default list, so that it is not strictly necessary to redefine it when trying simple things. The default scene will handle spheres, boxes, and facets with *frictional* properties correctly, and adjusts the timestep dynamically. You can find an example in simple-scene-default-engines.py.

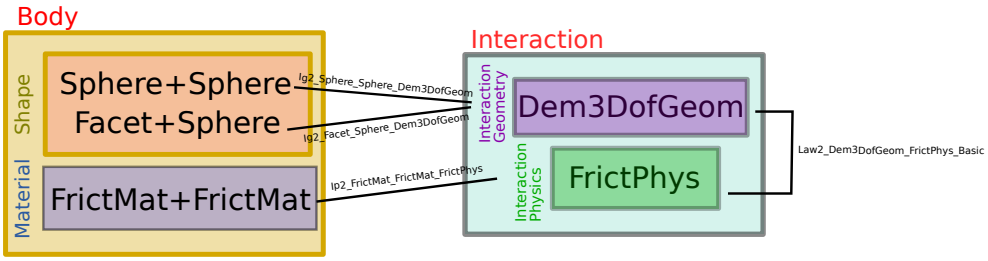


Fig. 1.2: Chain of functors producing and accepting certain types. In the case shown, the `Ig2` functors produce *ScfGeom* instances from all handled *Shape* combinations; the `Ig2` functor produces *FrictMat*. The constitutive law functor `Law2` accepts the combination of types produced. Note that the types are stated in the functor's class names.

Chapter 2

Tutorial

This tutorial originated as handout for a course held at Technische Universität Dresden / Fakultät Bauingenieurwesen / Institut für Geotechnik in Jaunary 2011. The focus was to give quick and rather practical introduction to people without prior modeling experience, but with knowledge of mechanics. Some computer literacy was assumed, though basics are reviewed in the Hands-on section.

The course did not in reality follow this document, but was based on interactive writing and commenting simple Examples, which were mostly suggested by participants; many thanks to them for their ideas and suggestions.

A few minor bugs were discovered during the course. They were all fixed in rev. 2640 of Yade which is therefore the minimum recommended version to run the examples (notably, 0.60 will not work).

2.1 Introduction

Slides Yade: past, present and future (updated version)

2.2 Hands-on

2.2.1 Shell basics

Directory tree

Directory tree is hierarchical way to organize files in operating systems. A typical (reduced) tree looks like this:

```
/           Root
 --boot       System startup
 --bin        Low-level programs
 --lib        Low-level libraries
 --dev        Hardware access
 --sbin       Administration programs
 --proc       System information
 --var        Files modified by system services
 --root       Root (administrator) home directory
 --etc        Configuration files
 --media      External drives
 --tmp        Temporary files
 --usr        Everything for normal operation (usr = UNIX system resources)
|     --bin      User programs
|     --sbin     Administration programs
|     --include  Header files for c/c++
```

```
|     --lib       Libraries
|     --local     Locally installed software
|     --doc       Documentation
--home        Contains the user's home directories
      --user      Home directory for user
      --user1     Home directory for user1
```

Note that there is a single root `/`; all other disks (such as USB sticks) attach to some point in the tree (e.g. in `/media`).

Shell navigation

Shell is the UNIX command-line, interface for conversation with the machine. Don't be afraid.

Moving around

The shell is always operated by some `user`, at some concrete `machine`; these two are constant. We can move in the directory structure, and the current place where we are is *current directory*. By default, it is the *home directory* which contains all files belonging to the respective user:

```
user@machine:~$                          # user operating at machine, in the directory ~ (= user's home director
user@machine:~$ ls .                     # list contents of the current directory
user@machine:~$ ls foo                   # list contents of directory foo, relative to the dcurrent directory ~
user@machine:~$ ls /tmp                  # list contents of /tmp
user@machine:~$ cd foo                   # change directory to foo
user@machine:~/foo$ ls ~                 # list home directory (= ls /home/user)
user@machine:~/foo$ cd bar               # change to bar (= cd ~/foo/bar)
user@machine:~/foo/bar$ cd ../../foo2    # go to the parent directory twice, then to foo2 (cd ~/foo/bar/../../fo
user@machine:~/foo2$ cd                  # go to the home directory (= ls ~ = ls /home/user)
user@machine:~$
```

Users typically have only permissions to write (i.e. modify files) only in their home directory (abbreviated `~`, usually is `/home/user`) and `/tmp`, and permissions to read files in most other parts of the system:

```
user@machine:~$ ls /root    # see what files the administrator has
ls: cannot open directory /root: Permission denied
```

Keys

Useful keys on the command-line are:

<tab>	show possible completions of what is being typed (use abundantly!)
^C (=Ctrl+C)	delete current line
^D	exit the shell
↑↓	move up and down in the command history
^C	interrupt currently running program
^\	kill currently running program
Shift-PgUp	scroll the screen up (show part output)
Shift-PgDown	scroll the screen down (show future output; works only on quantum computers)

Running programs

When a program is being run (without giving its full path), several directories are searched for program of that name; those directories are given by `$PATH`:

```
user@machine:~$ echo $PATH     # show the value of $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
user@machine:~$ which ls       # say what is the real path of ls
```

The first part of the command-line is the program to be run (`which`), the remaining parts are *arguments* (`ls` in this case). It is upt to the program which arguments it understands. Many programs can take special arguments called *options* starting with `-` (followed by a single letter) or `--` (followed by words); one of the common options is `-h` or `--help`, which displays how to use the program (try `ls --help`).

Full documentation for each program usually exists as *manual page* (or *man page*), which can be shown using e.g. `man ls` (`q` to exit)

Starting yade

If yade is installed on the machine, it can be (roughly speaking) run as any other program; without any arguments, it runs in the "dialog mode", where a command-line is presented:

```
user@machine:~$ yade
Welcome to Yade bzr2616
TCP python prompt on localhost:9002, auth cookie `adcusk'
XMLRPC info provider on http://localhost:21002
[[ ^L clears screen, ^U kills line. F12 controller, F11 3d view, F10 both, F9 generator, F8 plot. ]]
Yade [1]:                                          #### hit ^D to exit
Do you really want to exit ([y]/n)?
Yade: normal exit.
```

The command-line is in fact `python`, enriched with some yade-specific features. (Pure python interpreter can be run with `python` or `ipython` commands).

Instead of typing commands on-by-one on the command line, they can be be written in a file (with the .py extension) and given as argument to Yade:

```
user@machine:~$ yade simulation.py
```

For a complete help, see `man yade`

Exercises

1. Open the terminal, navigate to your home directory
2. Create a new empty file and save it in `~/first.py`
3. Change directory to `/tmp`; delete the file `~/first.py`
4. Run program `xeyes`
5. Look at the help of Yade.
6. Look at the *manual page* of Yade
7. Run Yade, exit and run it again.

2.2.2 Python basics

We assume the reader is familar with Python tutorial and only briefly review some of the basic capabilities. The following will run in pure-python interpreter (`python` or `ipython`), but also inside Yade, which is a super-set of Python.

Numerical operations and modules:

```
Yade [171]: (1+3*4)**2          # usual rules for operator precedence, ** is exponentiation
Out[171]: 169

Yade [172]: import math         # gain access to "module" of functions

Yade [173]: math.sqrt(2)        # use a function from that module
Out[173]: 1.4142135623730951
```

```
Yade [174]: import math as m  # use the module under a different name

Yade [175]: m.cos(m.pi)
Out[175]: -1.0

Yade [176]: from math import *  # import everything so that it can be used without module name

Yade [177]: cos(pi)
Out[177]: -1.0
```

Variables:

```
Yade [178]: a=1; b,c=2,3       # multiple commands separated with ;, multiple assignment

Yade [179]: a+b+c
Out[179]: 6
```

Sequences

Lists

Lists are variable-length sequences, which can be modified; they are written with braces `[...]`, and their elements are accessed with numerical indices:

```
Yade [180]: a=[1,2,3]          # list of numbers

Yade [181]: a[0]               # first element has index 0
Out[181]: 1

Yade [182]: a[-1]              # negative counts from the end
Out[182]: 3

Yade [183]: a[3]               # error
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/build/yade-Swrxdd/yade-1.20.0/debian/tmp/usr/lib/x86_64-linux-gnu/yade/py/yade/__init__.pyc in <module>()
----> 1 a[3]               # error

IndexError: list index out of range

Yade [184]: len(a)             # number of elements
Out[184]: 3

Yade [185]: a[1:]              # from second element to the end
Out[185]: [2, 3]

Yade [186]: a+=[4,5]           # extend the list

Yade [187]: a+=[6]; a.append(7) # extend with single value, both have the same effect

Yade [188]: 9 in a             # test presence of an element
Out[188]: False
```

Lists can be created in various ways:

```
Yade [189]: range(10)
Out[189]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Yade [190]: range(10)[-1]
Out[190]: 9
```

List of squares of even number smaller than 20, i.e. $\left\{a^2\ \forall a \in \{0,\cdots,19\} \;\middle|\; 2 \| a\right\}$ (note the similarity):

```
Yade [191]: [a**2 for a in range(20) if a%2==0]
Out[191]: [0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

Tuples

Tuples are constant sequences:

```
Yade [192]: b=(1,2,3)

Yade [193]: b[0]
Out[193]: 1

Yade [194]: b[0]=4              # error
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/build/yade-Swrxdd/yade-1.20.0/debian/tmp/usr/lib/x86_64-linux-gnu/yade/py/yade/__init__.pyc in <module>()
----> 1 b[0]=4              # error

TypeError: 'tuple' object does not support item assignment
```

Dictionaries

Mapping from keys to values:

```
Yade [195]: czde={'jedna':'ein','dva':'zwei','tri':'drei'}

Yade [196]: de={1:'ein',2:'zwei',3:'drei'}; cz={1:'jedna',2:'dva',3:'tri'}

Yade [197]: czde['jedna']          ## access values
Out[197]: 'ein'

Yade [198]: de[1], cz[2]
Out[198]: ('ein', 'dva')
```

Functions, conditionals

```
Yade [199]: 4==5
Out[199]: False

Yade [200]: a=3.1

Yade [201]: if a<pi: b=0           # conditional statement
   .....: else: b=1
   .....:
  File "<ipython-input-202-bebe87cdf86d>", line 1
    else: b=1
       ^
SyntaxError: invalid syntax


Yade [203]: c=0 if a<1 else 1      # conditional expression

Yade [204]: def square(x): return x**2    # define a new function
   .....:

Yade [205]: square(2)                     # and call that function
Out[205]: 4
```

Exercises

1. Read the following code and say what wil be the values of `a` and `b`:

```
a=range(5)
b=[(aa**2 if aa%2==0 else -aa**2) for aa in a]
```

2.2.3 Yade basics

Yade objects are constructed in the following manner (this process is also called "instantiation", since we create concrete instances of abstract classes: one individual sphere is an instance of the abstract *Sphere*, like Socrates is an instance of "man"):

```
Yade [206]: Sphere            # try also Sphere?
Out[206]: yade.wrapper.Sphere

Yade [207]: s=Sphere()        # create a Sphere, without specifying any attributes

Yade [208]: s.radius          # 'nan' is a special value meaning "not a number" (i.e. not defined)
Out[208]: nan

Yade [209]: s.radius=2        # set radius of an existing object

Yade [210]: s.radius
Out[210]: 2.0

Yade [211]: ss=Sphere(radius=3)   # create Sphere, giving radius directly

Yade [212]: s.radius, ss.radius     # also try typing s.<tab> to see defined attributes
Out[212]: (2.0, 3.0)
```

Particles

Particles are the "data" component of simulation; they are the objects that will undergo some processes, though do not define those processes yet.

Singles

There is a number of pre-defined functions to create particles of certain type; in order to create a sphere, one has to (see the source of *utils.sphere* for instance):

1. Create *Body*
2. Set *Body.shape* to be an instance of *Sphere* with some given radius
3. Set *Body.material* (last-defined material is used, otherwise a default material is created)
4. Set position and orientation in *Body.state*, compute mass and moment of inertia based on *Material* and *Shape*

In order to avoid such tasks, shorthand functions are defined in the *utils* module; to mention a few of them, they are *utils.sphere*, *utils.facet*, *utils.wall*.

```
Yade [213]: s=utils.sphere((0,0,0),radius=1)    # create sphere particle centered at (0,0,0) with radius=1

Yade [214]: s.shape                       # s.shape describes the geometry of the particle
Out[214]: <Sphere instance at 0x9878500>

Yade [215]: s.shape.radius                # we already know the Sphere class
Out[215]: 1.0
```

```
Yade [216]: s.state.mass, s.state.inertia # inertia is computed from density and geometry
Out[216]:
(4188.790204786391,
 Vector3(1675.5160819145563,1675.5160819145563,1675.5160819145563))

Yade [217]: s.state.pos                   # position is the one we prescribed
Out[217]: Vector3(0,0,0)

Yade [218]: s2=utils.sphere((-2,0,0),radius=1,fixed=True)     # explanation below
```

In the last example, the particle was fixed in space by the `fixed=True` parameter to *utils.sphere*; such a particle will not move, creating a primitive boundary condition.

A particle object is not yet part of the simulation; in order to do so, a special function is called:

```
Yade [219]: O.bodies.append(s)            # adds particle s to the simulation; returns id of the particle(s) ad
Out[219]: 13
```

Packs

There are functions to generate a specific arrangement of particles in the *pack* module; for instance, cloud (random loose packing) of spheres can be generated with the *pack.SpherePack* class:

```
Yade [220]: from yade import pack

Yade [221]: sp=pack.SpherePack()                    # create an empty cloud; SpherePack contains only geometrica

Yade [222]: sp.makeCloud((1,1,1),(2,2,2),rMean=.2) # put spheres with defined radius inside box given by corner
Out[222]: 6

Yade [223]: for c,r in sp: print c,r                # print center and radius of all particles (SpherePack is a
   .....:
Vector3(1.7299137562791944,1.4669309282367406,1.222844402140793) 0.2
Vector3(1.4488812384650869,1.7462698406039456,1.3914770006122785) 0.2
Vector3(1.412264788255342,1.2336025888413216,1.4305786907026345) 0.2
Vector3(1.2061586561670141,1.4845051861223812,1.7498464051166982) 0.2
Vector3(1.7886450114554213,1.4833482197330783,1.7019218530523794) 0.2
Vector3(1.5467049756522337,1.795885402519149,1.7842650943289187) 0.2

Yade [224]: sp.toSimulation()                       # create particles and add them to the simulation
Out[224]: [14, 15, 16, 17, 18, 19]
```

Boundaries

utils.facet (triangle *Facet*) and *utils.wall* (infinite axes-aligned plane *Wall*) geometries are typically used to define boundaries. For instance, a "floor" for the simulation can be created like this:

```
Yade [225]: O.bodies.append(utils.wall(-1,axis=2))
Out[225]: 20
```

There are other conveinence functions (like *utils.facetBox* for creating closed or open rectangular box, or family of *ymport* functions)

Look inside

The simulation can be inspected in several ways. All data can be accessed from python directly:

```
Yade [226]: len(O.bodies)
Out[226]: 21
```

```
Yade [227]: O.bodies[1].shape.radius   # radius of body #1 (will give error if not sphere, since only spheres h
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/build/yade-Swrxdd/yade-1.20.0/debian/tmp/usr/lib/x86_64-linux-gnu/yade/py/yade/__init__.pyc in <module>()
----> 1 O.bodies[1].shape.radius   # radius of body #1 (will give error if not sphere, since only spheres have

AttributeError: 'NoneType' object has no attribute 'shape'

Yade [228]: O.bodies[2].state.pos      # position of body #2
Out[228]: Vector3(0,0,0)
```

Besides that, Yade says this at startup (the line preceding the command-line):

```
[[ ^L clears screen, ^U kills line. F12 controller, F11 3d view, F10 both, F9 generator, F8 plot. ]]
```

Controller Pressing `F12` brings up a window for controlling the simulation. Although typically no human intervention is done in large simulations (which run "headless", without any graphical interaction), it can be handy in small examples. There are basic information on the simulation (will be used later).

3d view The 3d view can be opened with F11 (or by clicking on button in the *Controller* – see below). There is a number of keyboard shortcuts to manipulate it (press `h` to get basic help), and it can be moved, rotated and zoomed using mouse. Display-related settings can be set in the "Display" tab of the controller (such as whether particles are drawn).

Inspector *Inspector* is opened by clicking on the appropriate button in the *Controller*. It shows (and updated) internal data of the current simulation. In particular, one can have a look at engines, particles (*Bodies*) and interactions (*Interactions*). Clicking at each of the attribute names links to the appropriate section in the documentation.

Exercises

1. What is this code going to do?

```
Yade [229]: O.bodies.append([utils.sphere((2*i,0,0),1) for i in range(1,20)])
Out[229]: [21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
```

2. Create a simple simulation with cloud of spheres enclosed in the box `(0,0,0)` and `(1,1,1)` with mean radius .1. (hint: *pack.SpherePack.makeCloud*)
3. Enclose the cloud created above in box with corners `(0,0,0)` and `(1,1,1)`; keep the top of the box open. (hint: *utils.facetBox*; type `utils.facetBox?` or `utils.facetBox??` to get help on the command line)
4. Open the 3D view, try zooming in/out; position axes so that `z` is upwards, `y` goes to the right and `x` towards you.

Engines

Engines define processes undertaken by particles. As we know from the theoretical introduction, the sequence of engines is called *simulation loop*. Let us define a simple interaction loop:

```
Yade [230]: O.engines=[                    # newlines and indentations are not important until the brace is clos
   .....:    ForceResetter(),
   .....:    InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Wall_Aabb()]),
   .....:    InteractionLoop(           # dtto for the parenthesis here
   .....:        [Ig2_Sphere_Sphere_L3Geom(),Ig2_Wall_Sphere_L3Geom()],
   .....:        [Ip2_FrictMat_FrictMat_FrictPhys()],
   .....:        [Law2_L3Geom_FrictPhys_ElPerfPl()]
   .....:    ),
   .....:    NewtonIntegrator(damping=.2,label='newton')      # define a name under which we can access this en
   .....: ]
```

```
   .....:

Yade [231]: O.engines
Out[231]:
[<ForceResetter instance at 0xc8c4a00>,
 <InsertionSortCollider instance at 0x3fb2420>,
 <InteractionLoop instance at 0xd49b090>,
 <NewtonIntegrator instance at 0x51cc830>]

Yade [232]: O.engines[-1]==newton    # is it the same object?
Out[232]: True

Yade [233]: newton.damping
Out[233]: 0.2
```

Instead of typing everything into the command-line, one can describe simulation in a file (*script*) and then run yade with that file as an argument. We will therefore no longer show the command-line unless necessary; instead, only the script part will be shown. Like this:

```
O.engines=[                    # newlines and indentations are not important until the brace is closed
        ForceResetter(),
        InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Wall_Aabb()]),
        InteractionLoop(           # dtto for the parenthesis here
                [Ig2_Sphere_Sphere_L3Geom_Inc(),Ig2_Wall_Sphere_L3Geom_Inc()],
                [Ip2_FrictMat_FrictMat_FrictPhys()],
                [Law2_L3Geom_FrictPhys_ElPerfPl()]
        ),
        GravityEngine(gravity=(0,0,-9.81)),              # 9.81 is the gravity acceleration, and we say that
        NewtonIntegrator(damping=.2,label='newton')      # define a name under which we can access this engine
]
```

Besides engines being run, it is likewise important to define how often they will run. Some engines can run only sometimes (we will see this later), while most of them will run always; the time between two successive runs of engines is *timestep* (Δt). There is a mathematical limit on the timestep value, called *critical timestep*, which is computed from properties of particles. Since there is a function for that, we can just set timestep using *utils.PWaveTimeStep*:

```
O.dt=utils.PWaveTimeStep()
```

Each time when the simulation loop finishes, time `O.time` is advanced by the timestep `O.dt`:

```
Yade [234]: O.dt=0.01

Yade [235]: O.time
Out[235]: 0.0

Yade [236]: O.step()

Yade [237]: O.time
Out[237]: 0.01
```

For experimenting with a single simulations, it is handy to save it to memory; this can be achieved, once everything is defined, with:

```
O.saveTmp()
```

Exercises

1. Define *engines* as in the above example, run the *Inspector* and click through the engines to see their sequence.
2. Write a simple script which will

   (a) define particles as in the previous exercise (cloud of spheres inside a box open from the top)

   (b) define a simple simulation loop, as the one given above

   (c) set Δt equal to the critical P-Wave Δt

   (d) save the initial simulation state to memory

3. Run the previously-defined simulation multiple times, while changing the value of timestep (use the button to reload the initial configuration).

   (a) See what happens as you increase Δt above the P-Wave value.

   (b) Try changing the *gravity* parameter, before running the simulation.

   (c) Try changing *damping*

4. Reload the simulation, open the 3d view, open the *Inspector*, select a particle in the 3d view (shift-click). Then run the simulation and watch how forces on that particle change; pause the simulation somewhere in the middle, look at interactions of this particle.

5. At which point can we say that the deposition is done, so that the simulation can be stopped?

See also:

The *Bouncing sphere* example shows a basic simulation.

2.3 Data mining

2.3.1 Read

Local data

All data of the simulation are accessible from python; when you open the *Inspector*, blue labels of various data can be clicked – left button for getting to the documentation, middle click to copy the name of the object (use `Ctrl-V` or middle-click to paste elsewhere). The interesting objects are among others (see *Omega* for a full list):

1. *O.engines*

   Engines are accessed by their index (position) in the simulation loop:

```
O.engines[0]      # first engine
O.engines[-1]     # last engine
```

   Note: The index can change if *O.engines* is modified. *Labeling* introduced below is a better solution for reliable access to a particular engine.

2. *O.bodies*

   Bodies are identified by their *id*, which is guaranteed to not change during the whole simulation:

```
O.bodies[0]                                                  # first body
[b.shape.radius in O.bodies if isinstance(b.shape,Sphere)]   # list of radii of all spherical bodies
sum([b.state.mass for b in O.bodies])                        # sum of masses of all bodies
```

   Note: Uniqueness of *Body.id* is not guaranteed, since newly created bodies might recycle *ids* of *deleted* ones.

3. *O.force*

   Generalized forces (forces, torques) acting on each particle. They are (usually) reset at the beginning of each step with *ForceResetter*, subsequently forces from individual interactions are accumulated in *InteractionLoop*. To access the data, use:

```
O.forces.f(0)      # force on #0
O.forces.t(1)      # torque on #1
```

4. *O.interactions*

   Interactions are identified by *ids* of the respective interacting particles (they are created and deleted automatically during the simulation):

```
O.interactions[0,1]    # interactions of #0 with #1
O.interactions[1,0]    # the same object
O.bodies[0].intrs      # all interactions of body #0
```

Labels

Engines and *functors* can be *labeled*, which means that python variable of that name is automatically created.

```
Yade [164]: O.engines=[
   .....:    NewtonIntegrator(damping=.2,label='newton')
   .....: ]
   .....:

Yade [165]: newton.damping=.4

Yade [166]: O.engines[0].damping   # O.engines[0] and newton are the same objects
Out[166]: 0.4
```

Exercises

1. Find meaning of this expression:

```
max([b.state.vel.norm() for b in O.bodies])
```

2. Run the gravity deposition script, pause after a few seconds of simulation. Write expressions that compute
   (a) kinetic energy $\sum \frac{1}{2} m_i |v_i|^2$
   (b) average mass (hint: use numpy.average)
   (c) maximum z-coordinate of all particles
   (d) number of interactions of body #1

Global data

Useful measures of what happens in the simulation globally:

unbalanced force ratio of maximum contact force and maximum per-body force; measure of staticity, computed with *utils.unbalancedForce*.

porosity ratio of void volume and total volume; computed with *utils.porosity*.

coordination number average number of interactions per particle, *utils.avgNumInteractions*

stress tensor (periodic boundary conditions) averaged force in interactions, computed with *utils.normalShearStressTensor* and *utils.stressTensorOfPeriodicCell*

fabric tensor distribution of contacts in space (not yet implemented); can be visualized with *utils.plotDirections*

Energies

Evaluating energy data for all components in the simulation (such as gravity work, kinetic energy, plastic dissipation, damping dissipation) can be enabled with

```
O.trackEnergy=True
```

Subsequently, energy values are accessible in the *O.energy*; it is a dictionary where its entries can be retrived with `keys()` and their values with `O.energy[key]`.

2.3.2 Save

PyRunner

To save data that we just learned to access, we need to call Python from within the *simulation loop*. *PyRunner* is created just for that; it inherits periodicy control from *PeriodicEngine* and takes the code to run as text (must be quoted, i.e. inside `'...'`) attributed called *command*. For instance, adding this to *O.engines* will print the current step number every second:

```
O.engines=O.engines+[ PyRunner(command='print O.iter',realPeriod=1) ]
```

Writing complicated code inside *command* is awkward; in such case, we define a function that will be called:

```
def myFunction():
        '''Print step number, and pause the simulation is unbalanced force is smaller than 0.05.'''
        print O.iter
        if utils.unbalancedForce()<0.05:
                print 'Unbalanced force is smaller than 0.05, pausing.'
                O.pause()
O.engines=[
        # ...
        PyRunner(command='myFunction()',iterPeriod=100) # call myFunction every 100 steps
]
```

Exercises

1. Run the gravity deposition simulation, but change it such that:
   (a) *utils.unbalancedForce* is printed every 2 seconds.
   (b) check every 1000 steps the value of unbalanced force
      - if smaller than 0.2, set *damping* to 0.8 (hint: use labels)
      - if smaller than 0.1, pause the simulation

Keeping history

Yade provides the *plot* module used for storing and plotting variables (plotting itself will be discussed later). Periodic storing of data is done with *PyRunner* and the *plot.addData* function, for instance:

```
from yade import plot
O.engines=[  # ...,
        PyRunner(command='addPlotData()',realPeriod=2)                # call the addPlotData function every 2
]
def addPlotData():
        # this function adds current values to the history of data, under the names specified
        plot.addData(i=O.iter,t=O.time,Ek=utils.kineticEnergy(),coordNum=utils.avgNumInteractions(),unForce=uti
```

History is stored in *plot.data*, and can be accessed using the variable name, e.g. `plot.data['Ek']`, and saved to text file (for post-processing outside yade) with *plot.saveTxt*.

2.3.3 Plot

plot provides facilities for plotting history saved with *plot.addData* as 2d plots. Data to be plotted are specified using dictionary *plot.plots*

```
plot.plots={'t':('coordNum','unForce',None,'Ek')}
```

History of all values is given as the name used for *plot.addData*; keys of the dictionary are x-axis values, and values are sequence of data on the y axis; the `None` separates data on the left and right axes (they are scaled independently). The plot itself is created with

```
plot.plot()          # on the command line, F8 can be used as shorthand
```

While the plot is open, it will be updated periodically, so that simulation evolution can be seen in real-time.

Energy plots

Plotting all energy contributions would be difficult, since names of all energies might not be known in advance. Fortunately, there is a way to handle that in Yade. It consists in two parts:

1. *plot.addData* is given all the energies that are currently defined:

   ```
   plot.addData(i=O.iter,total=O.energy.total(),**O.energy)
   ```

   The *O.energy.total* functions, which sums all energies together. The `**O.energy` is special python syntax for converting dictionary (remember that *O.energy* is a dictionary) to named functions arguments, so that the following two commands are identical:

   ```
   function(a=3,b=34)              # give arguments as arguments
   function(**{'a':3,'b':34})      # create arguments from dictionary
   ```

2. Data to plot are specified using a *function* that gives names of data to plot, rather than providing the data names directly:

   ```
   plot.plots={'i':['total',O.energy.keys()]}
   ```

   where `total` is the name we gave to `O.energy.total()` above, while `O.energy.keys()` will always return list of currently defined energies.

Exercises

1. Run the gravity deposition script, plotting unbalanced force and kinetic energy.
2. While the script is running, try changing the *NewtonIntegrator.damping* parameter (do it from both *Inspector* and from the command-line). What influence does it have on the evolution of unbalanced force and kinetic energy?
3. Think about and write down all energy sources (input); write down also all energy sinks (dissipation).
4. Simulate gravity deposition and plot all energies as they evolve during the simulation.

See also:

Most *Examples* use plotting facilities of Yade, some of them also track energy of the simulation.

2.4 Towards geomechanics

See also:

Examples *Gravity deposition*, *Oedometric test*, *Periodic simple shear*, *Periodic triaxial test* deal with topics discussed here.

2.4.1 Parametric studies

Input parameters of the simulation (such as size distribution, damping, various contact parameters, …) influence the results, but frequently an analytical relationship is not known. To study such influence, similar simulations differing only in a few parameters can be run and results compared. Yade can be run in *batch mode*, where one simulation script is used in conjunction with *parameter table*, which specifies parameter values for each run of the script. Batch simulation are run non-interactively, i.e. without user intervention; the user must therefore start and stop the simulation explicitly.

Suppose we want to study the influence of *damping* on the evolution of kinetic energy. The script has to be adapted at several places:

1. We have to make sure the script reads relevant parameters from the *parameter table*. This is done using *utils.readParamsFromTable*; the parameters which are read are created as variables in the `yade.params.table` module:

```
utils.readParamsFromTable(damping=.2)      # yade.params.table.damping variable will be created
from yade.params import table              # typing table.damping is easier than yade.params.table.damping
```

   Note that *utils.readParamsFromTable* takes default values of its parameters, which are used if the script is not run in non-batch mode.

2. Parameters from the table are used at appropriate places:

```
NewtonIntegrator(damping=table.damping),
```

3. The simulation is run non-interactively; we must therefore specify at which point it should stop:

```
O.engines+=[PyRunner(iterPeriod=1000,command='checkUnbalancedForce()')]    # call our function defined bel

def checkUnbalancedForce():
   if utils.unbalancedForce<0.05:                    # exit Yade if unbalanced force drops below 0.05
      utils.saveDataTxt(O.tags['d.id']+'.data.bz2')  # save all data into a unique file before exiting
      import sys
      sys.exit(0)                                    # exit the program
```

4. Finally, we must start the simulation at the very end of the script:

```
O.run()               # run forever, until stopped by checkUnbalancedForce()
utils.waitIfBatch()   # do not finish the script until the simulation ends; does nothing in non-batch mode
```

The *parameter table* is a simple text-file, where each line specifies a simulation to run:

```
# comments start with # as in python
damping     # first non-comment line is variable name
.2
.4
.6
```

Finally, the simulation is run using the special batch command:

```
user@machine:~$ yade-batch parameters.table simulation.py
```

Exercises

1. Run the gravity deposition script in batch mode, varying *damping* to take values of `.2`, `.4`, `.6`. See the http://localhost:9080 overview page while the batch is running.

2.4.2 Boundary

Particles moving in infinite space usually need some constraints to make the simulation meaningful.

Supports

So far, supports (unmovable particles) were providing necessary boundary: in the gravity deposition script, *utils.facetBox* is by internally composed of *facets* (triangulation elements), which is `fixed` in space; facets are also used for arbitrary triangulated surfaces (see relevant sections of the *User's manual*). Another frequently used boundary is *utils.wall* (infinite axis-aligned plane).

Periodic

Periodic boundary is a "boundary" created by using periodic (rather than infinite) space. Such boundary is activated by *O.periodic=True* , and the space configuration is decribed by *O.cell* . It is well suited for studying bulk material behavior, as boundary effects are avoided, leading to smaller number of particles. On the other hand, it might not be suitable for studying localization, as any cell-level effects (such as shear bands) have to satisfy periodicity as well.

The periodic cell is described by its *reference size* of box aligned with global axes, and *current transformation*, which can capture stretch, shear and rotation. Deformation is prescribed via *velocity gradient*, which updates the *transformation* before the next step. *Homothetic deformation* can smear *velocity gradient* accross the cell, making the boundary dissolve in the whole cell.

Stress and strains can be controlled with *PeriTriaxController*; it is possible to prescribe mixed strain/stress *goal* state using *PeriTriaxController.stressMask*.

The following creates periodic cloud of spheres and compresses to achieve σ_x=-10 kPa, σ_y=-10kPa and ε_z=-0.1. Since stress is specified for y and z, *stressMask* is `0b011` (x→1, y→2, z→4, in decimal 1+2=3).

```
Yade [167]: sp=pack.SpherePack()

Yade [168]: sp.makeCloud((1,1,1),(2,2,2),rMean=.2,periodic=True)
Out[168]: 9

Yade [169]: sp.toSimulation()              # implicitly sets O.periodic=True, and O.cell.refSize to the packing
Out[169]: [4, 5, 6, 7, 8, 9, 10, 11, 12]

Yade [170]: O.engines+=[PeriTriaxController(goal=(-1e4,-1e4,-.1),stressMask=0b011,maxUnbalanced=.2,doneHook='fu
```

When the simulation *runs*, *PeriTriaxController* takes over the control and calls *doneHook* when *goal* is reached. A full simulation with PeriTriaxController might look like the following:

```
from yade import pack,plot
sp=pack.SpherePack()
rMean=.05
sp.makeCloud((0,0,0),(1,1,1),rMean=rMean,periodic=True)
sp.toSimulation()
O.engines=[
   ForceResetter(),
   InsertionSortCollider([Bo1_Sphere_Aabb()],verletDist=.05*rMean),
   InteractionLoop([Ig2_Sphere_Sphere_L3Geom()],[Ip2_FrictMat_FrictMat_FrictPhys()],[Law2_L3Geom_FrictPhys_ElPe
   NewtonIntegrator(damping=.6),
   PeriTriaxController(goal=(-1e6,-1e6,-.1),stressMask=0b011,maxUnbalanced=.2,doneHook='goalReached()',label='t
   PyRunner(iterPeriod=100,command='addPlotData()')
]
O.dt=.5*utils.PWaveTimeStep()
O.trackEnergy=True
def goalReached():
   print 'Goal reached, strain',triax.strain,' stress',triax.stress
   O.pause()
def addPlotData():
   plot.addData(sx=triax.stress[0],sy=triax.stress[1],sz=triax.stress[2],ex=triax.strain[0],ey=triax.strain[1],
      i=O.iter,unbalanced=utils.unbalancedForce(),
      totalEnergy=O.energy.total(),**O.energy       # plot all energies
   )
plot.plots={'i':(('unbalanced','go'),None,'kinetic'),' i':('ex','ey','ez',None,'sx','sy','sz'),'i  ':(O.energy.k
```

```
plot.plot()
O.saveTmp()
O.run()
```

2.5 Advanced & more

2.5.1 Particle size distribution

See *Periodic triaxial test*

2.5.2 Clumps

Clump; see *Periodic triaxial test*

2.5.3 Testing laws

LawTester, scripts/test/law-test.py

2.5.4 New law

2.5.5 Visualization

See the example *3d postprocessing*

- *VTKRecorder* & Paraview
- *qt.SnapshotEngine*

2.6 Examples

2.6.1 Bouncing sphere

```
# basic simulation showing sphere falling ball gravity,
# bouncing against another sphere representing the support

# DATA COMPONENTS

# add 2 particles to the simulation
# they the default material (utils.defaultMat)
O.bodies.append([
   # fixed: particle's position in space will not change (support)
   sphere(center=(0,0,0),radius=.5,fixed=True),
   # this particles is free, subject to dynamics
   sphere((0,0,2),.5)
])

# FUNCTIONAL COMPONENTS

# simulation loop -- see presentation for the explanation
O.engines=[
   ForceResetter(),
   InsertionSortCollider([Bo1_Sphere_Aabb()]),
   InteractionLoop(
      [Ig2_Sphere_Sphere_L3Geom()],        # collision geometry
      [Ip2_FrictMat_FrictMat_FrictPhys()], # collision "physics"
```

```
      [Law2_L3Geom_FrictPhys_ElPerfPl()]   # contact law -- apply forces
   ),
   # Apply gravity force to particles. damping: numerical dissipation of energy.
   NewtonIntegrator(gravity=(0,0,-9.81),damping=0.1)
]

# set timestep to a fraction of the critical timestep
# the fraction is very small, so that the simulation is not too fast
# and the motion can be observed
O.dt=.5e-4*PWaveTimeStep()

# save the simulation, so that it can be reloaded later, for experimentation
O.saveTmp()
```

2.6.2 Gravity deposition

```
# gravity deposition in box, showing how to plot and save history of data,
# and how to control the simulation while it is running by calling
# python functions from within the simulation loop

# import yade modules that we will use below
from yade import pack, plot

# create rectangular box from facets
O.bodies.append(geom.facetBox((.5,.5,.5),(.5,.5,.5),wallMask=31))

# create empty sphere packing
# sphere packing is not equivalent to particles in simulation, it contains only the pure geometry
sp=pack.SpherePack()
# generate randomly spheres with uniform radius distribution
sp.makeCloud((0,0,0),(1,1,1),rMean=.05,rRelFuzz=.5)
# add the sphere pack to the simulation
sp.toSimulation()

O.engines=[
   ForceResetter(),
   InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb()]),
   InteractionLoop(
      # handle sphere+sphere and facet+sphere collisions
      [Ig2_Sphere_Sphere_L3Geom(),Ig2_Facet_Sphere_L3Geom()],
      [Ip2_FrictMat_FrictMat_FrictPhys()],
      [Law2_L3Geom_FrictPhys_ElPerfPl()]
   ),
   NewtonIntegrator(gravity=(0,0,-9.81),damping=0.4),
   # call the checkUnbalanced function (defined below) every 2 seconds
   PyRunner(command='checkUnbalanced()',realPeriod=2),
   # call the addPlotData function every 200 steps
   PyRunner(command='addPlotData()',iterPeriod=100)
]
O.dt=.5*PWaveTimeStep()

# enable energy tracking; any simulation parts supporting it
# can create and update arbitrary energy types, which can be
# accessed as O.energy['energyName'] subsequently
O.trackEnergy=True

# if the unbalanced forces goes below .05, the packing
# is considered stabilized, therefore we stop collected
# data history and stop
def checkUnbalanced():
   if unbalancedForce()<.05:
```

```
      O.pause()
      plot.saveDataTxt('bbb.txt.bz2')
      # plot.saveGnuplot('bbb') is also possible

# collect history of data which will be plotted
def addPlotData():
   # each item is given a names, by which it can be the unsed in plot.plots
   # the **O.energy converts dictionary-like O.energy to plot.addData arguments
   plot.addData(i=O.iter,unbalanced=unbalancedForce(),**O.energy)

# define how to plot data: 'i' (step number) on the x-axis, unbalanced force
# on the left y-axis, all energies on the right y-axis
# (O.energy.keys is function which will be called to get all defined energies)
# None separates left and right y-axis
plot.plots={'i':('unbalanced',None,O.energy.keys)}

# show the plot on the screen, and update while the simulation runs
plot.plot()

O.saveTmp()
```

2.6.3 Oedometric test

```
# gravity deposition, continuing with oedometric test after stabilization
# shows also how to run parametric studies with yade-batch

# The components of the batch are:
# 1. table with parameters, one set of parameters per line (ccc.table)
# 2. readParamsFromTable which reads respective line from the parameter file
# 3. the simulation muse be run using yade-batch, not yade
#
# $ yade-batch --job-threads=1 03-oedometric-test.table 03-oedometric-test.py
#

# load parameters from file if run in batch
# default values are used if not run from batch
readParamsFromTable(rMean=.05,rRelFuzz=.3,maxLoad=1e6,minLoad=1e4)
# make rMean, rRelFuzz, maxLoad accessible directly as variables later
from yade.params.table import *

# create box with free top, and ceate loose packing inside the box
from yade import pack, plot
O.bodies.append(geom.facetBox((.5,.5,.5),(.5,.5,.5),wallMask=31))
sp=pack.SpherePack()
sp.makeCloud((0,0,0),(1,1,1),rMean=rMean,rRelFuzz=rRelFuzz)
sp.toSimulation()

O.engines=[
   ForceResetter(),
   # sphere, facet, wall
   InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb(),Bo1_Wall_Aabb()]),
   InteractionLoop(
      # the loading plate is a wall, we need to handle sphere+sphere, sphere+facet, sphere+wall
      [Ig2_Sphere_Sphere_L3Geom(),Ig2_Facet_Sphere_L3Geom(),Ig2_Wall_Sphere_L3Geom()],
      [Ip2_FrictMat_FrictMat_FrictPhys()],
      [Law2_L3Geom_FrictPhys_ElPerfPl()]
   ),
   NewtonIntegrator(gravity=(0,0,-9.81),damping=0.5),
   # the label creates an automatic variable referring to this engine
   # we use it below to change its attributes from the functions called
   PyRunner(command='checkUnbalanced()',realPeriod=2,label='checker'),
```

```
]
O.dt=.5*PWaveTimeStep()

# the following checkUnbalanced, unloadPlate and stopUnloading functions are all called by the 'checker'
# (the last engine) one after another; this sequence defines progression of different stages of the
# simulation, as each of the functions, when the condition is satisfied, updates 'checker' to call
# the next function when it is run from within the simulation next time

# check whether the gravity deposition has already finished
# if so, add wall on the top of the packing and start the oedometric test
def checkUnbalanced():
   # at the very start, unbalanced force can be low as there is only few contacts, but it does not mean the pac
   if O.iter<5000: return
   # the rest will be run only if unbalanced is < .1 (stabilized packing)
   if unbalancedForce()>.1: return
   # add plate at the position on the top of the packing
   # the maximum finds the z-coordinate of the top of the topmost particle
   O.bodies.append(wall(max([b.state.pos[2]+b.shape.radius for b in O.bodies if isinstance(b.shape,Sphere)]),ax
   global plate        # without this line, the plate variable would only exist inside this function
   plate=O.bodies[-1]  # the last particles is the plate
   # Wall objects are "fixed" by default, i.e. not subject to forces
   # prescribing a velocity will therefore make it move at constant velocity (downwards)
   plate.state.vel=(0,0,-.1)
   # start plotting the data now, it was not interesting before
   O.engines=O.engines+[PyRunner(command='addPlotData()',iterPeriod=200)]
   # next time, do not call this function anymore, but the next one (unloadPlate) instead
   checker.command='unloadPlate()'

def unloadPlate():
   # if the force on plate exceeds maximum load, start unloading
   if abs(O.forces.f(plate.id)[2])>maxLoad:
      plate.state.vel*=-1
      # next time, do not call this function anymore, but the next one (stopUnloading) instead
      checker.command='stopUnloading()'

def stopUnloading():
   if abs(O.forces.f(plate.id)[2])<minLoad:
      # O.tags can be used to retrieve unique identifiers of the simulation
      # if running in batch, subsequent simulation would overwrite each other's output files otherwise
      # d (or description) is simulation description (composed of parameter values)
      # while the id is composed of time and process number
      plot.saveDataTxt(O.tags['d.id']+'.txt')
      O.pause()

def addPlotData():
   if not isinstance(O.bodies[-1].shape,Wall):
      plot.addData(); return
   Fz=O.forces.f(plate.id)[2]
   plot.addData(Fz=Fz,w=plate.state.pos[2]-plate.state.refPos[2],unbalanced=unbalancedForce(),i=O.iter)

# besides unbalanced force evolution, also plot the displacement-force diagram
plot.plots={'i':('unbalanced',),'w':('Fz',)}
plot.plot()

O.run()
# when running with yade-batch, the script must not finish until the simulation is done fully
# this command will wait for that (has no influence in the non-batch mode)
waitIfBatch()
```

Batch table

```
rMean rRelFuzz maxLoad
.05 .1 1e6
.05 .2 1e6
.05 .3 1e6
```

2.6.4 Periodic simple shear

```
# encoding: utf-8

# script for periodic simple shear test, with periodic boundary
# first compresses to attain some isotropic stress (checkStress),
# then loads in shear (checkDistorsion)
#
# the initial packing is either regular (hexagonal), with empty bands along the boundary,
# or periodic random cloud of spheres
#
# material friction angle is initially set to zero, so that the resulting packing is dense
# (sphere rearrangement is easier if there is no friction)
#


# setup the periodic boundary
O.periodic=True
O.cell.refSize=(2,2,2)

from yade import pack,plot

# the "if 0:" block will be never executed, therefore the "else:" block will be
# to use cloud instead of regular packing, change to "if 1:" or something similar
if 0:
   # create cloud of spheres and insert them into the simulation
   # we give corners, mean radius, radius variation
   sp=pack.SpherePack()
   sp.makeCloud((0,0,0),(2,2,2),rMean=.1,rRelFuzz=.6,periodic=True)
   # insert the packing into the simulation
   sp.toSimulation(color=(0,0,1)) # pure blue
else:
   # in this case, add dense packing
   O.bodies.append(
      pack.regularHexa(pack.inAlignedBox((0,0,0),(2,2,2)),radius=.1,gap=0,color=(0,0,1))
   )

# create "dense" packing by setting friction to zero initially
O.materials[0].frictionAngle=0

# simulation loop (will be run at every step)
O.engines=[
   ForceResetter(),
   InsertionSortCollider([Bo1_Sphere_Aabb()]),
   InteractionLoop(
      [Ig2_Sphere_Sphere_L3Geom()],
      [Ip2_FrictMat_FrictMat_FrictPhys()],
      [Law2_L3Geom_FrictPhys_ElPerfPl()]
   ),
   NewtonIntegrator(damping=.4),
   # run checkStress function (defined below) every second
   # the label is arbitrary, and is used later to refer to this engine
   PyRunner(command='checkStress()',realPeriod=1,label='checker'),
   # record data for plotting every 100 steps; addData function is defined below
```

```
   PyRunner(command='addData()',iterPeriod=100)
]

# set the integration timestep to be 1/2 of the "critical" timestep
O.dt=.5*PWaveTimeStep()

# prescribe isotropic normal deformation (constant strain rate)
# of the periodic cell
O.cell.velGrad=Matrix3(-.1,0,0, 0,-.1,0, 0,0,-.1)

# when to stop the isotropic compression (used inside checkStress)
limitMeanStress=-5e5

# called every second by the PyRunner engine
def checkStress():
   # stress tensor as the sum of normal and shear contributions
   # Matrix3.Zero is the intial value for sum(...)
   stress=sum(normalShearStressTensors(),Matrix3.Zero)
   print 'mean stress',stress.trace()/3.
   # if mean stress is below (bigger in absolute value) limitMeanStress, start shearing
   if stress.trace()/3.<limitMeanStress:
      # apply constant-rate distorsion on the periodic cell
      O.cell.velGrad=Matrix3(0,0,.1, 0,0,0, 0,0,0)
      # change the function called by the checker engine
      # (checkStress will not be called anymore)
      checker.command='checkDistorsion()'
      # block rotations of particles to increase tanPhi, if desired
      # disabled by default
      if 0:
         for b in O.bodies:
            # block X,Y,Z rotations, translations are free
            b.state.blockedDOFs='XYZ'
            # stop rotations if any, as blockedDOFs block accelerations really
            b.state.angVel=(0,0,0)
      # set friction angle back to non-zero value
      # tangensOfFrictionAngle is computed by the Ip2_* functor from material
      # for future contacts change material (there is only one material for all particles)
      O.materials[0].frictionAngle=.5 # radians
      # for existing contacts, set contact friction directly
      for i in O.interactions: i.phys.tangensOfFrictionAngle=tan(.5)

# called from the 'checker' engine periodically, during the shear phase
def checkDistorsion():
   # if the distorsion value is >.3, exit; otherwise do nothing
   if abs(O.cell.trsf[0,2])>.5:
      # save data from addData(...) before exiting into file
      # use O.tags['id'] to distinguish individual runs of the same simulation
      plot.saveDataTxt(O.tags['id']+'.txt')
      # exit the program
      #import sys
      #sys.exit(0) # no error (0)
      O.pause()

# called periodically to store data history
def addData():
   # get the stress tensor (as 3x3 matrix)
   stress=sum(normalShearStressTensors(),Matrix3.Zero)
   # give names to values we are interested in and save them
   plot.addData(exz=O.cell.trsf[0,2],szz=stress[2,2],sxz=stress[0,2],tanPhi=stress[0,2]/stress[2,2],i=O.iter)
   # color particles based on rotation amount
   for b in O.bodies:
      # rot() gives rotation vector between reference and current position
      b.shape.color=scalarOnColorScale(b.state.rot().norm(),0,pi/2.)
```

```
# define what to plot (3 plots in total)
## exz(i), [left y axis, separate by None:] szz(i), sxz(i)
## szz(exz), sxz(exz)
## tanPhi(i)
# note the space in 'i ' so that it does not overwrite the 'i' entry
plot.plots={'i':('exz',None,'szz','sxz'),'exz':('szz','sxz'),'i ':('tanPhi',)}

# better show rotation of particles
Gl1_Sphere.stripes=True

# open the plot on the screen
plot.plot()

O.saveTmp()
```

2.6.5 3d postprocessing

```
# demonstrate 3d postprocessing with yade
#
# 1. qt.SnapshotEngine saves images of the 3d view as it appears on the screen periodically
#    makeVideo is then used to make real movie from those images
# 2. VTKRecorder saves data in files which can be opened with Paraview
#    see the User's manual for an intro to Paraview

# generate loose packing
from yade import pack, qt
sp=pack.SpherePack()
sp.makeCloud((0,0,0),(2,2,2),rMean=.1,rRelFuzz=.6,periodic=True)
# add to scene, make it periodic
sp.toSimulation()

O.engines=[
   ForceResetter(),
   InsertionSortCollider([Bo1_Sphere_Aabb()]),
   InteractionLoop(
      [Ig2_Sphere_Sphere_L3Geom()],
      [Ip2_FrictMat_FrictMat_FrictPhys()],
      [Law2_L3Geom_FrictPhys_ElPerfPl()]
   ),
   NewtonIntegrator(damping=.4),
   # save data for Paraview
   VTKRecorder(fileName='3d-vtk-',recorders=['all'],iterPeriod=1000),
   # save data from Yade's own 3d view
   qt.SnapshotEngine(fileBase='3d-',iterPeriod=200,label='snapshot'),
   # this engine will be called after 20000 steps, only once
   PyRunner(command='finish()',iterPeriod=20000)
]
O.dt=.5*PWaveTimeStep()

# prescribe constant-strain deformation of the cell
O.cell.velGrad=Matrix3(-.1,0,0, 0,-.1,0, 0,0,-.1)

# we must open the view explicitly (limitation of the qt.SnapshotEngine)
qt.View()

# this function is called when the simulation is finished
def finish():
   # snapshot is label of qt.SnapshotEngine
   # the 'snapshots' attribute contains list of all saved files
   makeVideo(snapshot.snapshots,'3d.mpeg',fps=10,bps=10000)
   O.pause()
```

```
# set parameters of the renderer, to show network chains rather than particles
# these settings are accessible from the Controller window, on the second tab ("Display") as well
rr=yade.qt.Renderer()
rr.shape=False
rr.intrPhys=True
```

2.6.6 Periodic triaxial test

```
# encoding: utf-8

# periodic triaxial test simulation
#
# The initial packing is either
#
# 1. random cloud with uniform distribution, or
# 2. cloud with specified granulometry (radii and percentages), or
# 3. cloud of clumps, i.e. rigid aggregates of several particles
#
# The triaxial consists of 2 stages:
#
# 1. isotropic compaction, until sigmaIso is reached in all directions;
#    this stage is ended by calling compactionFinished()
# 2. constant-strain deformation along the z-axis, while maintaining
#    constant stress (sigmaIso) laterally; this stage is ended by calling
#    triaxFinished()
#
# Controlling of strain and stresses is performed via PeriTriaxController,
# of which parameters determine type of control and also stability
# condition (maxUnbalanced) so that the packing is considered stabilized
# and the stage is done.
#

sigmaIso=-1e5

#import matplotlib
#matplotlib.use('Agg')

# generate loose packing
from yade import pack, qt, plot

O.periodic=True
sp=pack.SpherePack()
if 0:
   ## uniform distribution
   sp.makeCloud((0,0,0),(2,2,2),rMean=.1,rRelFuzz=.3,periodic=True)
else:
   ## create packing from clumps
   # configuration of one clump
   c1=pack.SpherePack([((0,0,0),.03333),((.03,0,0),.017),((0,.03,0),.017)])
   # make cloud using the configuration c1 (there could c2, c3, ...; selection between them would be random)
   sp.makeClumpCloud((0,0,0),(2,2,2),[c1],periodic=True,num=500)

# setup periodic boundary, insert the packing
sp.toSimulation()

O.engines=[
   ForceResetter(),
   InsertionSortCollider([Bo1_Sphere_Aabb()]),
   InteractionLoop(
      [Ig2_Sphere_Sphere_ScGeom()],
      [Ip2_FrictMat_FrictMat_FrictPhys()],
```

```
      [Law2_ScGeom_FrictPhys_CundallStrack()]
   ),
   PeriTriaxController(label='triax',
      # specify target values and whether they are strains or stresses
      goal=(sigmaIso,sigmaIso,sigmaIso),stressMask=7,
      # type of servo-control
      dynCell=True,maxStrainRate=(10,10,10),
      # wait until the unbalanced force goes below this value
      maxUnbalanced=.1,relStressTol=1e-3,
      # call this function when goal is reached and the packing is stable
      doneHook='compactionFinished()'
   ),
   NewtonIntegrator(damping=.2),
   PyRunner(command='addPlotData()',iterPeriod=100),
]
O.dt=.5*PWaveTimeStep()

def addPlotData():
   plot.addData(unbalanced=unbalancedForce(),i=O.iter,
      sxx=triax.stress[0],syy=triax.stress[1],szz=triax.stress[2],
      exx=triax.strain[0],eyy=triax.strain[1],ezz=triax.strain[2],
      # save all available energy data
      Etot=O.energy.total(),**O.energy
   )

# enable energy tracking in the code
O.trackEnergy=True

# define what to plot
plot.plots={'i':('unbalanced',),'i ':('sxx','syy','szz'),' i':('exx','eyy','ezz'),
   # energy plot
   ' i ':(O.energy.keys,None,'Etot'),
}
# show the plot
plot.plot()

def compactionFinished():
   # set the current cell configuration to be the reference one
   O.cell.trsf=Matrix3.Identity
   # change control type: keep constant confinement in x,y, 20% compression in z
   triax.goal=(sigmaIso,sigmaIso,-.2)
   triax.stressMask=3
   # allow faster deformation along x,y to better maintain stresses
   triax.maxStrainRate=(1.,1.,.1)
   # next time, call triaxFinished instead of compactionFinished
   triax.doneHook='triaxFinished()'
   # do not wait for stabilization before calling triaxFinished
   triax.maxUnbalanced=10

def triaxFinished():
   print 'Finished'
   O.pause()
```

Chapter 3

User's manual

3.1 Scene construction

3.1.1 Adding particles

The *BodyContainer* holds *Body* objects in the simulation; it is accessible as `O.bodies`.

Creating Body objects

Body objects are only rarely constructed by hand by their components (*Shape*, *Bound*, *State*, *Material*); instead, convenience functions *sphere*, *facet* and *wall* are used to create them. Using these functions also ensures better future compatibility, if internals of *Body* change in some way. These functions receive geometry of the particle and several other characteristics. See their documentation for details. If the same *Material* is used for several (or many) bodies, it can be shared by adding it in `O.materials`, as explained below.

Defining materials

The `O.materials` object (instance of *Omega.materials*) holds defined shared materials for bodies. It only supports addition, and will typically hold only a few instance (though there is no limit).

`label` given to each material is optional, but can be passed to *sphere* and other functions for constructing body. The value returned by `O.materials.append` is an `id` of the material, which can be also passed to *sphere* – it is a little bit faster than using label, though not noticeable for small number of particles and perhaps less convenient.

If no *Material* is specified when calling *sphere*, the *last* defined material is used; that is a convenient default. If no material is defined yet (hence there is no last material), a default material will be created: FrictMat(density=2e3,young=30e9,poisson=.3,frictionAngle=.5236). This should not happen for serious simulations, but is handy in simple scripts, where exact material properties are more or less irrelevant.

```
Yade [238]: len(O.materials)
Out[238]: 0

Yade [239]: idConcrete=O.materials.append(FrictMat(young=30e9,poisson=.2,frictionAngle=.6,label="concrete"))

Yade [240]: O.materials[idConcrete]
Out[240]: <FrictMat instance at 0x65e7080>

# uses the last defined material
Yade [241]: O.bodies.append(sphere(center=(0,0,0),radius=1))
Out[241]: 0
```

```
# material given by id
Yade [242]: O.bodies.append(sphere((0,0,2),1,material=idConcrete))
Out[242]: 1

# material given by label
Yade [243]: O.bodies.append(sphere((0,2,0),1,material="concrete"))
Out[243]: 2

Yade [244]: idSteel=O.materials.append(FrictMat(young=210e9,poisson=.25,frictionAngle=.8,label="steel"))

Yade [245]: len(O.materials)
Out[245]: 2

# implicitly uses "steel" material, as it is the last one now
Yade [246]: O.bodies.append(facet([(1,0,0),(0,1,0),(-1,-1,0)]))
Out[246]: 3
```

Adding multiple particles

As shown above, bodies are added one by one or several at the same time using the `append` method:

```
Yade [247]: O.bodies.append(sphere((0,10,0),1))
Out[247]: 0

Yade [248]: O.bodies.append(sphere((0,0,2),1))
Out[248]: 1

# this is the same, but in one function call
Yade [249]: O.bodies.append([
   .....:   sphere((0,0,0),1),
   .....:   sphere((0,0,2),1)
   .....: ])
   .....:
Out[249]: [2, 3]
```

Many functions introduced in next sections return list of bodies which can be readily added to the simulation, including

- packing generators, such as *pack.randomDensePack*, *pack.regularHexa*
- surface function *pack.gtsSurface2Facets*
- import functions *ymport.gmsh*, *ymport.stl*, ...

As those functions use *sphere* and *facet* internally, they accept additional argument passed to those function. In particular, material for each body is selected following the rules above (last one if not specified, by label, by index, etc.).

Clumping particles together

In some cases, you might want to create rigid aggregate of individual particles (i.e. particles will retain their mutual position during simulation). This we call a *clump*. A clump is internally represented by a special *body*, referenced by *clumpId* of its members (see also *isClump*, *isClumpMember* and *isStandalone*). Like every body a clump has a *position*, which is the (mass) balance point between all members. A clump body itself has no *interactions* with other bodies. Interactions between clumps is represented by interactions between clump members. There are no interactions between clump members of the same clump.

YADE supports different ways of creating clumps:

- Create clumps and spheres (clump members) directly with one command:

The function *appendClumped()* is designed for this task. For instance, we might add 2 spheres tied together:

```
Yade [250]: O.bodies.appendClumped([
   .....:    sphere([0,0,0],1),
   .....:    sphere([0,0,2],1)
   .....: ])
   .....:
Out[250]: (2, [0, 1])

Yade [251]: len(O.bodies)
Out[251]: 3

Yade [252]: O.bodies[1].isClumpMember, O.bodies[2].clumpId
Out[252]: (True, 2)

Yade [253]: O.bodies[2].isClump, O.bodies[2].clumpId
Out[253]: (True, 2)
```

-> *appendClumped()* returns a tuple of ids `(clumpId,[memberId1,memberId2,...])`

- Use existing spheres and clump them together:

For this case the function *clump()* can be used. One way to do this is to create a list of bodies, that should be clumped before using the *clump()* command:

```
Yade [254]: bodyList = []

Yade [255]: for ii in range(0,5):
   .....:    bodyList.append(O.bodies.append(sphere([ii,0,1],.5)))#create a "chain" of 5 spheres
   .....:

Yade [256]: print bodyList
[0, 1, 2, 3, 4]

Yade [257]: idClump=O.bodies.clump(bodyList)
```

-> *clump()* returns `clumpId`

- Another option is to replace *standalone* spheres from a given packing (see *SpherePack* and *makeCloud*) by clumps using clump templates.

This is done by a function called *replaceByClumps()*. This function takes a list of *clumpTemplates()* and a list of amounts and replaces spheres by clumps. The volume of a new clump will be the same as the volume of the sphere, that was replaced (clump volume/mass/inertia is accounting for overlaps assuming that there are only pair overlaps).

-> *replaceByClumps()* returns a list of tuples: `[(clumpId1,[memberId1,memberId2,...]),(clumpId2,[memberId1,memb`

It is also possible to *add* bodies to a clump and *release* bodies from a clump. Also you can *erase* the clump (clump members will get standalone spheres).

Additionally YADE supports to achieve the *roundness* of a clump or roundness coefficient of a packing. Parts of the packing can be excluded from roundness measurement via exclude list.

```
Yade [258]: bodyList = []

Yade [259]: for ii in range(1,5):
   .....:    bodyList.append(O.bodies.append(sphere([ii,ii,ii],.5)))
   .....:

Yade [260]: O.bodies.clump(bodyList)
Out[260]: 4

Yade [261]: RC=O.bodies.getRoundness()
```

```
Yade [262]: print RC
0.256191414232
```

-> *getRoundness()* returns roundness coefficient RC of a packing or a part of the packing

Note: Have a look at examples/clumps/ folder. There you will find some examples, that show usage of different functions for clumps.

3.1.2 Sphere packings

Representing a solid of an arbitrary shape by arrangement of spheres presents the problem of sphere packing, i.e. spatial arrangement of sphere such that given solid is approximately filled with them. For the purposes of DEM simulation, there can be several requirements.

1. Distribution of spheres' radii. Arbitrary volume can be filled completely with spheres provided there are no restrictions on their radius; in such case, number of spheres can be infinite and their radii approach zero. Since both number of particles and minimum sphere radius (via critical timestep) determine computation cost, radius distribution has to be given mandatorily. The most typical distribution is uniform: mean±dispersion; if dispersion is zero, all spheres will have the same radius.
2. Smooth boundary. Some algorithms treat boundaries in such way that spheres are aligned on them, making them smoother as surface.
3. Packing density, or the ratio of spheres volume and solid size. It is closely related to radius distribution.
4. Coordination number, (average) number of contacts per sphere.
5. Isotropy (related to regularity/irregularity); packings with preferred directions are usually not desirable, unless the modeled solid also has such preference.
6. Permissible Spheres' overlap; some algorithms might create packing where spheres slightly overlap; since overlap usually causes forces in DEM, overlap-free packings are sometimes called "stress-free .

Volume representation

There are 2 methods for representing exact volume of the solid in question in Yade: boundary representation and constructive solid geometry. Despite their fundamental differences, they are abstracted in Yade in the *Predicate* class. Predicate provides the following functionality:

1. defines axis-aligned bounding box for the associated solid (optionally defines oriented bounding box);
2. can decide whether given point is inside or outside the solid; most predicates can also (exactly or approximately) tell whether the point is inside *and* satisfies some given padding distance from the represented solid boundary (so that sphere of that volume doesn't stick out of the solid).

Constructive Solid Geometry (CSG)

CSG approach describes volume by geometric *primitives* or primitive solids (sphere, cylinder, box, cone, ...) and boolean operations on them. Primitives defined in Yade include *inCylinder*, *inSphere*, *inEllipsoid*, *inHyperboloid*, *notInNotch*.

For instance, *hyperboloid* (dogbone) specimen for tension-compression test can be constructed in this way (shown at img. *img-hyperboloid*):

```
from yade import pack

## construct the predicate first
pred=pack.inHyperboloid(centerBottom=(0,0,-.1),centerTop=(0,0,.1),radius=.05,skirt=.03)
```

```
## alternatively: pack.inHyperboloid((0,0,-.1),(0,0,.1),.05,.03)

## pack the predicate with spheres (will be explained later)
spheres=pack.randomDensePack(pred,spheresInCell=2000,radius=3.5e-3)

## add spheres to simulation
O.bodies.append(spheres)
```

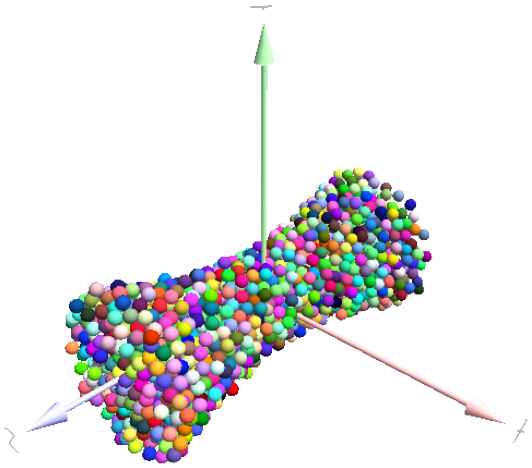


Fig. 3.1: Specimen constructed with the *pack.inHyperboloid* predicate, packed with *pack.randomDensePack*.

Boundary representation (BREP)

Representing a solid by its boundary is much more flexible than CSG volumes, but is mostly only approximate. Yade interfaces to GNU Triangulated Surface Library (GTS) to import surfaces readable by GTS, but also to construct them explicitly from within simulation scripts. This makes possible parametric construction of rather complicated shapes; there are functions to create set of 3d polylines from 2d polyline (*pack.revolutionSurfaceMeridians*), to triangulate surface between such set of 3d polylines (*pack.sweptPolylines2gtsSurface*).

For example, we can construct a simple funnel (examples/funnel.py, shown at *img-funnel*):

```
from numpy import linspace
from yade import pack

# angles for points on circles
thetas=linspace(0,2*pi,num=16,endpoint=True)

# creates list of polylines in 3d from list of 2d projections
# turned from 0 to π
meridians=pack.revolutionSurfaceMeridians(
        [[(3+rad*sin(th),10*rad+rad*cos(th)) for th in thetas] for rad in linspace(1,2,num=10)],
        linspace(0,pi,num=10)
)

# create surface
surf=pack.sweptPolylines2gtsSurface(
        meridians+
        +[[Vector3(5*sin(-th),-10+5*cos(-th),30) for th in thetas]]  # add funnel top
)

# add to simulation
O.bodies.append(pack.gtsSurface2Facets(surf))
```

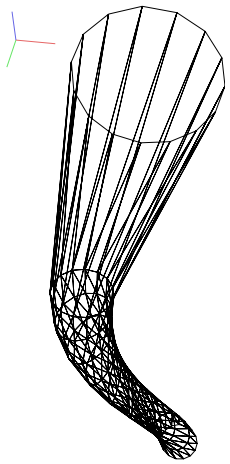


Fig. 3.2: Triangulated funnel, constructed with the examples/funnel.py script.

GTS surface objects can be used for 2 things:

1. *pack.gtsSurface2Facets* function can create the triangulated surface (from *Facet* particles) in the simulation itself, as shown in the funnel example. (Triangulated surface can also be imported directly from a STL file using *ymport.stl*.)
2. *pack.inGtsSurface* predicate can be created, using the surface as boundary representation of the enclosed volume.

The examples/gts-horse/gts-horse.py (img. *img-horse*) shows both possibilities; first, a GTS surface is imported:

```
import gts
surf=gts.read(open('horse.coarse.gts'))
```

That surface object is used as predicate for packing:

```
pred=pack.inGtsSurface(surf)
O.bodies.append(pack.regularHexa(pred,radius=radius,gap=radius/4.))
```

and then, after being translated, as base for triangulated surface in the simulation itself:

```
surf.translate(0,0,-(aabb[1][2]-aabb[0][2]))
O.bodies.append(pack.gtsSurface2Facets(surf,wire=True))
```

Boolean operations on predicates

Boolean operations on pair of predicates (noted `A` and `B`) are defined:

- *intersection* `A & B` (conjunction): point must be in both predicates involved.
- *union* `A | B` (disjunction): point must be in the first or in the second predicate.
- *difference* `A - B` (conjunction with second predicate negated): the point must be in the first predicate and not in the second one.
- *symmetric difference* `A ^ B` (exclusive disjunction): point must be in exactly one of the two predicates.

Composed predicates also properly define their bounding box. For example, we can take box and remove cylinder from inside, using the `A - B` operation (img. *img-predicate-difference*):

```
pred=pack.inAlignedBox((-2,-2,-2),(2,2,2))-pack.inCylinder((0,-2,0),(0,2,0),1)
spheres=pack.randomDensePack(pred,spheresInCell=2000,radius=.1,rRelFuzz=.4)
```

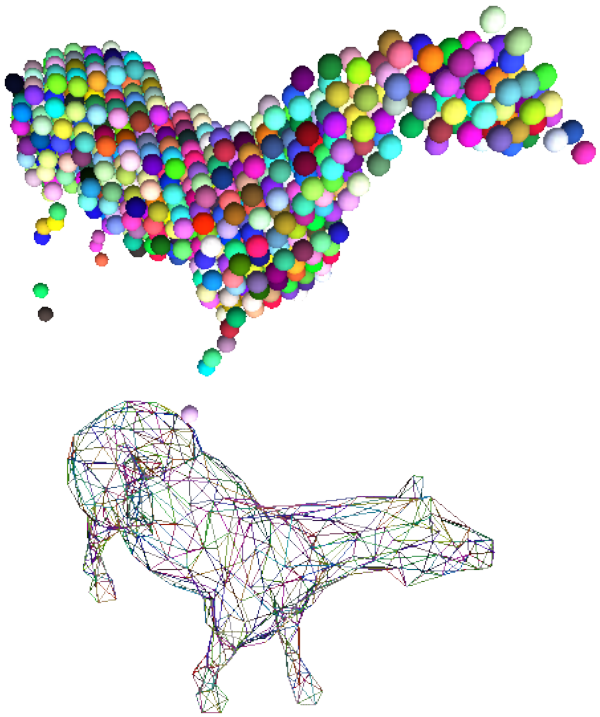


Fig. 3.3: Imported GTS surface (horse) used as packing predicate (top) and surface constructed from *facets* (bottom). See http://www.youtube.com/watch?v=PZVruIlUX1A for movie of this simulation.

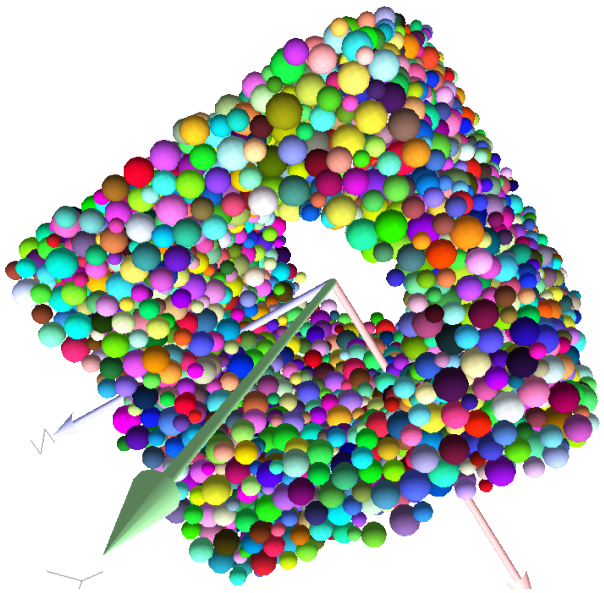


Fig. 3.4: Box with cylinder removed from inside, using difference of these two predicates.

Packing algorithms

Algorithms presented below operate on geometric spheres, defined by their center and radius. With a few exception documented below, the procedure is as follows:

1. Sphere positions and radii are computed (some functions use volume predicate for this, some do not)
2. *sphere* is called for each position and radius computed; it receives extra keyword arguments of the packing function (i.e. arguments that the packing function doesn't specify in its definition; they are noted `**kw`). Each *sphere* call creates actual *Body* objects with *Sphere shape*. List of *Body* objects is returned.
3. List returned from the packing function can be added to simulation using `O.bodies.append`.

Taking the example of pierced box:

```
pred=pack.inAlignedBox((-2,-2,-2),(2,2,2))-pack.inCylinder((0,-2,0),(0,2,0),1)
spheres=pack.randomDensePack(pred,spheresInCell=2000,radius=.1,rRelFuzz=.4,wire=True,color=(0,0,1),material=1)
```

Keyword arguments `wire`, `color` and `material` are not declared in *pack.randomDensePack*, therefore will be passed to *sphere*, where they are also documented. `spheres` is now list of *Body* objects, which we add to the simulation:

```
O.bodies.append(spheres)
```

Packing algorithms described below produce dense packings. If one needs loose packing, *pack.SpherePack* class provides functions for generating loose packing, via its *pack.SpherePack.makeCloud* method. It is used internally for generating initial configuration in dynamic algorithms. For instance:

```
from yade import pack
sp=pack.SpherePack()
sp.makeCloud(minCorner=(0,0,0),maxCorner=(3,3,3),rMean=.2,rRelFuzz=.5)
```

will fill given box with spheres, until no more spheres can be placed. The object can be used to add spheres to simulation:

```
for c,r in sp: O.bodies.append(sphere(c,r))
```

or, in a more pythonic way, with one single `O.bodies.append` call:

```
O.bodies.append([sphere(c,r) for c,r in sp])
```

Geometric

Geometric algorithms compute packing without performing dynamic simulation; among their advantages are

- speed;
- spheres touch exactly, there are no overlaps (what some people call "stress-free" packing);

their chief disadvantage is that radius distribution cannot be prescribed exactly, save in specific cases (regular packings); sphere radii are given by the algorithm, which already makes the system determined. If exact radius distribution is important for your problem, consider dynamic algorithms instead.

Regular Yade defines packing generators for spheres with constant radii, which can be used with volume predicates as described above. They are dense orthogonal packing (*pack.regularOrtho*) and dense hexagonal packing (*pack.regularHexa*). The latter creates so-called "hexagonal close packing", which achieves maximum density (http://en.wikipedia.org/wiki/Close-packing_of_spheres).

Clear disadvantage of regular packings is that they have very strong directional preferences, which might not be an issue in some cases.

Irregular Random geometric algorithms do not integrate at all with volume predicates described above; rather, they take their own boundary/volume definition, which is used during sphere positioning. On the other hand, this makes it possible for them to respect boundary in the sense of making spheres touch it at appropriate places, rather than leaving empty space in-between.

GenGeo is library (python module) for packing generation developed with ESyS-Particle. It creates packing by random insertion of spheres with given radius range. Inserted spheres touch each other exactly and, more importantly, they also touch the boundary, if in its neighbourhood. Boundary is represented as special object of the GenGeo library (Sphere, cylinder, box, convex polyhedron, ...). Therefore, GenGeo cannot be used with volume represented by yade predicates as explained above.

Packings generated by this module can be imported directly via *ymport.gengeo*, or from saved file via *ymport.gengeoFile*. There is an example script examples/test/genCylLSM.py. Full documentation for GenGeo can be found at ESyS documentation website.

To our knowledge, the GenGeo library is not currently packaged. It can be downloaded from current subversion repository

```
svn checkout https://svn.esscc.uq.edu.au/svn/esys3/lsm/contrib/LSMGenGeo
```

then following instruction in the `INSTALL` file.

Dynamic

The most versatile algorithm for random dense packing is provided by *pack.randomDensePack*. Initial loose packing of non-overlapping spheres is generated by randomly placing them in cuboid volume, with radii given by requested (currently only uniform) radius distribution. When no more spheres can be inserted, the packing is compressed and then uncompressed (see py/pack/pack.py for exact values of these "stresses") by running a DEM simulation; *Omega.switchScene* is used to not affect existing simulation). Finally, resulting packing is clipped using provided predicate, as explained above.

By its nature, this method might take relatively long; and there are 2 provisions to make the computation time shorter:

- If number of spheres using the `spheresInCell` parameter is specified, only smaller specimen with *periodic* boundary is created and then repeated as to fill the predicate. This can provide high-quality packing with low regularity, depending on the `spheresInCell` parameter (value of several thousands is recommended).
- Providing `memoizeDb` parameter will make *pack.randomDensePack* first look into provided file (SQLite database) for packings with similar parameters. On success, the packing is simply read from database and returned. If there is no similar pre-existent packing, normal procedure is run, and the result is saved in the database before being returned, so that subsequent calls with same parameters will return quickly.

If you need to obtain full periodic packing (rather than packing clipped by predicate), you can use *pack.randomPeriPack*.

In case of specific needs, you can create packing yourself, "by hand". For instance, packing boundary can be constructed from *facets*, letting randomly positioned spheres in space fall down under gravity.

3.1.3 Triangulated surfaces

Yade integrates with the the GNU Triangulated Surface library, exposed in python via GTS module. GTS provides variety of functions for surface manipulation (coarsening, tesselation, simplification, import), to be found in its documentation.

GTS surfaces are geometrical objects, which can be inserted into simulation as set of particles whose *Body.shape* is of type *Facet* – single triangulation elements. *pack.gtsSurface2Facets* can be used to convert GTS surface triangulation into list of *bodies* ready to be inserted into simulation via `O.bodies.append`.

Facet particles are created by default as non-*Body.dynamic* (they have zero inertial mass). That means that they are fixed in space and will not move if subject to forces. You can however

- prescribe arbitrary movement to facets using a *PartialEngine* (such as *TranslationEngine* or *RotationEngine*);
- assign explicitly *mass* and *inertia* to that particle;
- make that particle part of a clump and assign *mass* and *inertia* of the clump itself (described below).

Note: Facets can only (currently) interact with *spheres*, not with other facets, even if they are *dynamic*. Collision of 2 *facets* will not create interaction, therefore no forces on facets.

Import

Yade currently offers 3 formats for importing triangulated surfaces from external files, in the *ymport* module:

ymport.gts text file in native GTS format.

ymport.stl STereoLitography format, in either text or binary form; exported from Blender, but from many CAD systems as well.

***ymport.gmsh*.** text file in native format for GMSH, popular open-source meshing program.

If you need to manipulate surfaces before creating list of facets, you can study the py/ymport.py file where the import functions are defined. They are rather simple in most cases.

Parametric construction

The GTS module provides convenient way of creating surface by vertices, edges and triangles.

Frequently, though, the surface can be conveniently described as surface between polylines in space. For instance, cylinder is surface between two polygons (closed polylines). The *pack.sweptPolylines2gtsSurface* offers the functionality of connecting several polylines with triangulation.

Note: The implementation of *pack.sweptPolylines2gtsSurface* is rather simplistic: all polylines must be of the same length, and they are connected with triangles between points following their indices within each polyline (not by distance). On the other hand, points can be co-incident, if the `threshold` parameter is positive: degenerate triangles with vertices closer that `threshold` are automatically eliminated.

Manipulating lists efficiently (in terms of code length) requires being familiar with list comprehensions in python.

Another examples can be found in examples/mill.py (fully parametrized) or examples/funnel.py (with hardcoded numbers).

3.1.4 Creating interactions

In typical cases, interactions are created during simulations as particles collide. This is done by a *Collider* detecting approximate contact between particles and then an *IGeomFunctor* detecting exact collision.

Some material models (such as the *concrete model*) rely on initial interaction network which is denser than geometrical contact of spheres: sphere's radii as "enlarged" by a dimensionless factor called *interaction radius* (or *interaction ratio*) to create this initial network. This is done typically in this way (see examples/concrete/uniax.py for an example):

1. Approximate collision detection is adjusted so that approximate contacts are detected also between particles within the interaction radius. This consists in setting value of *Bo1_Sphere_Aabb.aabbEnlargeFactor* to the interaction radius value.

2. The geometry functor (`Ig2`) would normally say that "there is no contact" if given 2 spheres that are not in contact. Therefore, the same value as for *Bo1_Sphere_Aabb.aabbEnlargeFactor* must be given to it (*Ig2_Sphere_Sphere_ScGeom.interactionDetectionFactor*).

   Note that only *Sphere* + *Sphere* interactions are supported; there is no parameter analogous to *distFactor* in *Ig2_Facet_Sphere_ScGeom*. This is on purpose, since the interaction radius is meaningful in bulk material represented by sphere packing, whereas facets usually represent boundary conditions which should be exempt from this dense interaction network.

3. Run one single step of the simulation so that the initial network is created.
4. Reset interaction radius in both `Bo1` and `Ig2` functors to their default value again.
5. Continue the simulation; interactions that are already established will not be deleted (the `Law2` functor in usepermitting).

In code, such scenario might look similar to this one (labeling is explained in *Labeling things*):

```
intRadius=1.5

O.engines=[
   ForceResetter(),
   InsertionSortCollider([
      # enlarge here
      Bo1_Sphere_Aabb(aabbEnlargeFactor=intRadius,label='bo1s'),
      Bo1_Facet_Aabb(),
        ]),
   InteractionLoop(
      [
         # enlarge here
         Ig2_Sphere_Sphere_ScGeom(interactionDetectionFactor=intRadius,label='ig2ss'),
         Ig2_Facet_Sphere_ScGeom(),
      ],
      [Ip2_CpmMat_CpmMat_CpmPhys()],
      [Law2_ScGeom_CpmPhys_Cpm(epsSoft=0)], # deactivated
   ),
   NewtonIntegrator(damping=damping,label='damper'),
]

# run one single step
O.step()

# reset interaction radius to the default value
bo1s.aabbEnlargeFactor=1.0
ig2ss.interactionDetectionFactor=1.0

# now continue simulation
O.run()
```

Individual interactions on demand

It is possible to create an interaction between a pair of particles independently of collision detection using *createInteraction*. This function looks for and uses matching `Ig2` and `Ip2` functors. Interaction will be created regardless of distance between given particles (by passing a special parameter to the `Ig2` functor to force creation of the interaction even without any geometrical contact). Appropriate constitutive law should be used to avoid deletion of the interaction at the next simulation step.

```
Yade [263]: O.materials.append(FrictMat(young=3e10,poisson=.2,density=1000))
Out[263]: 0

Yade [264]: O.bodies.append([
   .....:    sphere([0,0,0],1),
   .....:    sphere([0,0,1000],1)
   .....: ])
```

```
   .....:
Out[264]: [0, 1]

# only add InteractionLoop, no other engines are needed now
Yade [265]: O.engines=[
   .....:    InteractionLoop(
   .....:        [Ig2_Sphere_Sphere_ScGeom(),],
   .....:        [Ip2_FrictMat_FrictMat_FrictPhys()],
   .....:        [] # not needed now
   .....:    )
   .....: ]
   .....:

Yade [266]: i=createInteraction(0,1)

# created by functors in InteractionLoop
Yade [267]: i.geom, i.phys
Out[267]: (<ScGeom instance at 0x67236d0>, <FrictPhys instance at 0x8522130>)
```

This method will be rather slow if many interaction are to be created (the functor lookup will be repeated for each of them). In such case, ask on yade-dev@lists.launchpad.net to have the *createInteraction* function accept list of pairs id's as well.

3.1.5 Base engines

A typical DEM simulation in Yade does at least the following at each step (see *Function components* for details):

1. Reset forces from previous step
2. Detect new collisions
3. Handle interactions
4. Apply forces and update positions of particles

Each of these points corresponds to one or several engines:

```
O.engines=[
   ForceResetter(),          # reset forces
   InsertionSortCollider([...]),  # approximate collision detection
   InteractionLoop([...],[...],[...]) # handle interactions
   NewtonIntegrator()        # apply forces and update positions
]
```

The order of engines is important. In majority of cases, you will put any additional engine after *InteractionLoop*:

- if it apply force, it should come before *NewtonIntegrator*, otherwise the force will never be effective.
- if it makes use of bodies' positions, it should also come before *NewtonIntegrator*, otherwise, positions at the next step will be used (this might not be critical in many cases, such as output for visualization with *VTKRecorder*).

The *O.engines* sequence must be always assigned at once (the reason is in the fact that although engines themselves are passed by reference, the sequence is *copied* from c++ to Python or from Python to c++). This includes modifying an existing `O.engines`; therefore

```
O.engines.append(SomeEngine()) # wrong
```

will not work;

```
O.engines=O.engines+[SomeEngine()] # ok
```

must be used instead. For inserting an engine after position #2 (for example), use python slice notation:

```
O.engines=O.engines[:2]+[SomeEngine()]+O.engines[2:]
```

Note: When Yade starts, O.engines is filled with a reasonable default list, so that it is not strictly necessary to redefine it when trying simple things. The default scene will handle spheres, boxes, and facets with *frictional* properties correctly, and adjusts the timestep dynamically. You can find an example in simple-scene-default-engines.py.

Functors choice

In the above example, we omited functors, only writing ellipses `...` instead. As explained in *Dispatchers and functors*, there are 4 kinds of functors and associated dispatchers. User can choose which ones to use, though the choice must be consistent.

Bo1 functors

`Bo1` functors must be chosen depending on the collider in use; they are given directly to the collider (which internally uses *BoundDispatcher*).

At this moment (September 2010), the most common choice is *InsertionSortCollider*, which uses *Aabb*; functors creating *Aabb* must be used in that case. Depending on particle *shapes* in your simulation, choose appropriate functors:

```
O.engines=[...,
   InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb()]),
   ...
]
```

Using more functors than necessary (such as *Bo1_Facet_Aabb* if there are no *facets* in the simulation) has no performance penalty. On the other hand, missing functors for existing *shapes* will cause those bodies to not collider with other bodies (they will freely interpenetrate).

There are other colliders as well, though their usage is only experimental:

- *SpatialQuickSortCollider* is correctness-reference collider operating on *Aabb*; it is significantly slower than *InsertionSortCollider*.
- *PersistentTriangulationCollider* only works on spheres; it does not use a *BoundDispatcher*, as it operates on spheres directly.
- *FlatGridCollider* is proof-of-concept grid-based collider, which computes grid positions internally (no *BoundDispatcher* either)

Ig2 functors

`Ig2` functor choice (all of the derive from *IGeomFunctor*) depends on

1. shape combinations that should collide; for instance:

   ```
   InteractionLoop([Ig2_Sphere_Sphere_ScGeom()],[],[])
   ```

   will handle collisions for *Sphere* + *Sphere*, but not for *Facet* + *Sphere* – if that is desired, an additional functor must be used:

   ```
   InteractionLoop([
      Ig2_Sphere_Sphere_ScGeom(),
      Ig2_Facet_Sphere_ScGeom()
   ],[],[])
   ```

   Again, missing combination will cause given shape combinations to freely interpenetrate one another.

2. *IGeom* type accepted by the `Law2` functor (below); it is the first part of functor's name after `Law2` (for instance, *Law2_ScGeom_CpmPhys_Cpm* accepts *ScGeom*).

Ip2 functors

`Ip2` functors (deriving from *IPhysFunctor*) must be chosen depending on

1. *Material* combinations within the simulation. In most cases, `Ip2` functors handle 2 instances of the same *Material* class (such as *Ip2_FrictMat_FrictMat_FrictPhys* for 2 bodies with *FrictMat*)
2. *IPhys* accepted by the constitutive law (`Law2` functor), which is the second part of the `Law2` functor's name (e.g. *Law2_ScGeom_FrictPhys_CundallStrack* accepts *FrictPhys*)

Note: Unlike with `Bo1` and `Ig2` functors, unhandled combination of *Materials* is an error condition signaled by an exception.

Law2 functor(s)

`Law2` functor was the ultimate criterion for the choice of `Ig2` and `Ip2` functors; there are no restrictions on its choice in itself, as it only applies forces without creating new objects.

In most simulations, only one `Law2` functor will be in use; it is possible, though, to have several of them, dispatched based on combination of *IGeom* and *IPhys* produced previously by `Ig2` and `Ip2` functors respectively (in turn based on combination of *Shapes* and *Materials*).

Note: As in the case of `Ip2` functors, receiving a combination of *IGeom* and *IPhys* which is not handled by any `Law2` functor is an error.

Warning: Many `Law2` exist in Yade, and new ones can appear at any time. In some cases different functors are only different implementations of the same contact law (e.g. *Law2_ScGeom_FrictPhys_CundallStrack* and *Law2_L3Geom_FrictPhys_ElPerfPl*). Also, sometimes, the peculiarity of one functor may be reproduced as a special case of a more general one. Therefore, for a given constitutive behavior, the user may have the choice between different functors. It is strongly recommended to favor the most used and most validated implementation when facing such choice. A list of available functors classified from mature to unmaintained is updated here to guide this choice.

Examples

Let us give several example of the chain of created and accepted types.

Basic DEM model

Suppose we want to use the *Law2_ScGeom_FrictPhys_CundallStrack* constitutive law. We see that

1. the `Ig2` functors must create *ScGeom*. If we have for instance *spheres* and *boxes* in the simulation, we will need functors accepting *Sphere* + *Sphere* and *Box* + *Sphere* combinations. We don't want interactions between boxes themselves (as a matter of fact, there is no such functor anyway). That gives us *Ig2_Sphere_Sphere_ScGeom* and *Ig2_Box_Sphere_ScGeom*.
2. the `Ip2` functors should create *FrictPhys*. Looking at *InteractionPhysicsFunctors*, there is only *Ip2_FrictMat_FrictMat_FrictPhys*. That obliges us to use *FrictMat* for particles.

The result will be therefore:

```
InteractionLoop(
   [Ig2_Sphere_Sphere_ScGeom(),Ig2_Box_Sphere_ScGeom()],
   [Ip2_FrictMat_FrictMat_FrictPhys()],
```

```
   [Law2_ScGeom_FrictPhys_CundallStrack()]
)
```

Concrete model

In this case, our goal is to use the *Law2_ScGeom_CpmPhys_Cpm* constitutive law.

- We use *spheres* and *facets* in the simulation, which selects `Ig2` functors accepting those types and producing *ScGeom*: *Ig2_Sphere_Sphere_ScGeom* and *Ig2_Facet_Sphere_ScGeom*.
- We have to use *Material* which can be used for creating *CpmPhys*. We find that *CpmPhys* is only created by *Ip2_CpmMat_CpmMat_CpmPhys*, which determines the choice of *CpmMat* for all particles.

Therefore, we will use:

```
InteractionLoop(
   [Ig2_Sphere_Sphere_ScGeom(),Ig2_Facet_Sphere_ScGeom()],
   [Ip2_CpmMat_CpmMat_CpmPhys()],
   [Law2_ScGeom_CpmPhys_Cpm()]
)
```

3.1.6 Imposing conditions

In most simulations, it is not desired that all particles float freely in space. There are several ways of imposing boundary conditions that block movement of all or some particles with regard to global space.

Motion constraints

- *Body.dynamic* determines whether a body will be accelerated by *NewtonIntegrator*; it is mandatory to make it false for bodies with zero mass, where applying non-zero force would result in infinite displacement.

  Facets are case in the point: *facet* makes them non-dynamic by default, as they have zero volume and zero mass (this can be changed, by passing `dynamic=True` to *facet* or setting *Body.dynamic*; setting *State.mass* to a non-zero value must be done as well). The same is true for *wall*.

  Making sphere non-dynamic is achieved simply by:

```
b = sphere([x,y,z],radius,dynamic=False)
b.dynamic=True #revert the previous
```

- *State.blockedDOFs* permits selective blocking of any of 6 degrees of freedom in global space. For instance, a sphere can be made to move only in the xy plane by saying:

```
Yade [268]: O.bodies.append(sphere((0,0,0),1))
Out[268]: 0

Yade [269]: O.bodies[0].state.blockedDOFs='zXY'
```

  In contrast to *Body.dynamic*, *blockedDOFs* will only block forces (and acceleration) in selected directions. Actually, `b.dynamic=False` is nearly only a shorthand for `b.state.blockedDOFs=='xyzXYZ'` . A subtle difference is that the former does reset the velocity components automaticaly, while the latest does not. If you prescribed linear or angular velocity, they will be applied regardless of *blockedDOFs*. It also implies that if the velocity is not zero when degrees of freedom are blocked via blockedDOFs assignements, the body will keep moving at the velocity it has at the time of blocking. The differences are shown below:

```
Yade [270]: b1 = sphere([0,0,0],1,dynamic=True)

Yade [271]: b1.state.blockedDOFs
```

```
Out[271]: ''

Yade [272]: b1.state.vel = Vector3(1,0,0) #we want it to move...

Yade [273]: b1.dynamic = False #... at a constant velocity

Yade [274]: print b1.state.blockedDOFs, b1.state.vel
xyzXYZ Vector3(0,0,0)

Yade [275]: # oops, velocity has been reset when setting dynamic=False

Yade [276]: b1.state.vel = (1,0,0) # we can still assign it now

Yade [277]: print b1.state.blockedDOFs, b1.state.vel
xyzXYZ Vector3(1,0,0)

Yade [278]: b2 = sphere([0,0,0],1,dynamic=True) #another try

Yade [279]: b2.state.vel = (1,0,0)

Yade [280]: b2.state.blockedDOFs = "xyzXYZ" #this time we assign blockedDOFs directly, velocity is unchang

Yade [281]: print b2.state.blockedDOFs, b2.state.vel
xyzXYZ Vector3(1,0,0)
```

It might be desirable to constrain motion of some particles constructed from a generated sphere packing, following some condition, such as being at the bottom of a specimen; this can be done by looping over all bodies with a conditional:

```
for b in O.bodies:
   # block all particles with z coord below .5:
   if b.state.pos[2]<.5: b.dynamic=False
```

Arbitrary spatial predicates introduced above can be expoited here as well:

```
from yade import pack
pred=pack.inAlignedBox(lowerCorner,upperCorner)
for b in O.bodies:
   if b.shape.name!=Sphere: continue # skip non-spheres
   # ask the predicate if we are inside
   if pred(b.state.pos,b.shape.radius): b.dynamic=False
```

Imposing motion and forces

Imposed velocity

If a degree of freedom is blocked and a velocity is assigned along that direction (translational or rotational velocity), then the body will move at constant velocity. This is the simpler and recommended method to impose the motion of a body. This, for instance, will result in a constant velocity along x (it can still be freely accelerated along y and z):

```
O.bodies.append(sphere((0,0,0),1))
O.bodies[0].state.blockedDOFs='x'
O.bodies[0].state.vel=(10,0,0)
```

Conversely, modifying the position directly is likely to break Yade's algorithms, especially those related to collision detection and contact laws, as they are based on bodies velocities. Therefore, unless you really know what you are doing, don't do that for imposing a motion:

```
O.bodies.append(sphere((0,0,0),1))
O.bodies[0].state.blockedDOFs='x'
O.bodies[0].state.pos=10*O.dt #REALLY BAD! Don't assign position
```

Imposed force

Applying a force or a torque on a body is done via functions of the *ForceContainer*. It is as simple as this:

```
O.forces.addF(0,(1,0,0)) #applies for one step
```

By default, the force applies for one time step only, and is resetted at the beginning of each step. For this reason, imposing a force at the begining of one step will have no effect at all, since it will be immediatly resetted. The only way is to place a *PyRunner* inside the simulation loop.

Applying the force permanently is possible with an optional argument (in this case it does not matter if the command comes at the begining of the time step):

```
O.forces.addF(0,(1,0,0),permanent=True) #applies permanently
```

The force will persist across iterations, until it is overwritten by another call to `O.forces.addF(id,f,True)` or erased by `O.forces.reset(resetAll=True)`. The permanent force on a body can be checked with `O.forces.permF(id)`.

Boundary controllers

Engines deriving from *BoundaryController* impose boundary conditions during simulation, either directly, or by influencing several bodies. You are referred to their individual documentation for details, though you might find interesting in particular

- *UniaxialStrainer* for applying strain along one axis at constant rate; useful for plotting strain-stress diagrams for uniaxial loading case. See examples/concrete/uniax.py for an example.
- *TriaxialStressController* which applies prescribed stress/strain along 3 perpendicular axes on cuboid-shaped packing using 6 walls (*Box* objects) (*ThreeDTriaxialEngine* is generalized such that it allows independent value of stress along each axis)
- *PeriTriaxController* for applying stress/strain along 3 axes independently, for simulations using periodic boundary conditions (*Cell*)

Field appliers

Engines deriving from *FieldApplier* acting on all particles. The one most used is *GravityEngine* applying uniform acceleration field (*GravityEngine* is deprecated, use *NewtonIntegrator.gravity* instead!).

Partial engines

Engines deriving from *PartialEngine* define the *ids* attribute determining bodies which will be affected. Several of them warrant explicit mention here:

- *TranslationEngine* and *RotationEngine* for applying constant speed linear and rotational motion on subscribers.
- *ForceEngine* and *TorqueEngine* applying given values of force/torque on subscribed bodies at every step.
- *StepDisplacer* for applying generalized displacement delta at every timestep; designed for precise control of motion when testing constitutive laws on 2 particles.

The real value of partial engines is if you need to prescribe complex types of force or displacement fields. For moving a body at constant velocity or for imposing a single force, the methods explained in *Imposing motion and forces* are much simpler. There are several interpolating engines (*InterpolatingDirectedForceEngine* for applying force with varying magnitude, *InterpolatingHelixEngine* for applying spiral displacement with varying angular velocity and possibly others); writing a new interpolating engine is rather simple using examples of those that already exist.

3.1.7 Convenience features

Labeling things

Engines and functors can define that `label` attribute. Whenever the `O.engines` sequence is modified, python variables of those names are created/update; since it happens in the `__builtins__` namespaces, these names are immediately accessible from anywhere. This was used in *Creating interactions* to change interaction radius in multiple functors at once.

Warning: Make sure you do not use label that will overwrite (or shadow) an object that you already use under that variable name. Take care not to use syntactically wrong names, such as "er*452" or "my engine"; only variable names permissible in Python can be used.

Simulation tags

Omega.tags is a dictionary (it behaves like a dictionary, although the implementation in c++ is different) mapping keys to labels. Contrary to regular python dictionaries that you could create,

- `O.tags` is *saved and loaded with simulation*;
- `O.tags` has some values pre-initialized.

After Yade startup, `O.tags` contains the following:

```
Yade [282]: dict(O.tags) # convert to real dictionary
Out[282]:
{'author': 'root~(root@x86-csail-02)',
 'd.id': '20151114T011329p2070',
 'id': '20151114T011329p2070',
 'id.d': '20151114T011329p2070',
 'isoTime': '20151114T011329'}
```

author Real name, username and machine as obtained from your system at simulation creation

id Unique identifier of this Yade instance (or of the instance which created a loaded simulation). It is composed of date, time and process number. Useful if you run simulations in parallel and want to avoid overwriting each other's outputs; embed `O.tags['id']` in output filenames (either as directory name, or as part of the file's name itself) to avoid it. This is explained in *Separating output files from jobs* in detail.

isoTime Time when simulation was created (with second resolution).

d.id, id.d Simulation description and id joined by period (and vice-versa). Description is used in batch jobs; in non-batch jobs, these tags are identical to id.

You can add your own tags by simply assigning value, with the restriction that the left-hand side object must be a string and must not contain `=`.

```
Yade [283]: O.tags['anythingThat I lik3']='whatever'

Yade [284]: O.tags['anythingThat I lik3']
Out[284]: 'whatever'
```

Saving python variables

Python variable lifetime is limited; in particular, if you save simulation, variables will be lost after reloading. Yade provides limited support for data persistence for this reason (internally, it uses special values of `O.tags`). The functions in question are *saveVars* and *loadVars*.

saveVars takes dictionary (variable names and their values) and a *mark* (identification string for the variable set); it saves the dictionary inside the simulation. These variables can be re-created (after the simulation was loaded from a XML file, for instance) in the `yade.params.`*mark* namespace by calling *loadVars* with the same identification *mark*:

```
Yade [285]: a=45; b=pi/3

Yade [286]: saveVars('ab',a=a,b=b)

# save simulation (we could save to disk just as well)
Yade [286]: O.saveTmp()

Yade [288]: O.loadTmp()

Yade [289]: loadVars('ab')

Yade [290]: yade.params.ab.a
Out[290]: 45

# import like this
Yade [291]: from yade.params import ab

Yade [292]: ab.a, ab.b
Out[292]: (45, 1.0471975511965976)

# also possible
Yade [293]: from yade.params import *

Yade [294]: ab.a, ab.b
Out[294]: (45, 1.0471975511965976)
```

Enumeration of variables can be tedious if they are many; creating local scope (which is a function definition in Python, for instance) can help:

```
def setGeomVars():
        radius=a*4
        thickness=22
        p_t=4/3*pi
        dim=Vector3(1.23,2.2,3)
        #
        # define as much as you want here
        # it all appears in locals() (and nothing else does)
        #
        saveVars('geom',loadNow=True,**locals())

setGeomVars()
from yade.params.geom import *
# use the variables now
```

Note: Only types that can be pickled can be passed to *saveVars*.

3.2 Controlling simulation

3.2.1 Tracking variables

Running python code

A special engine *PyRunner* can be used to periodically call python code, specified via the `command` parameter. Periodicity can be controlled by specifying computation time (`realPeriod`), virutal time (`virtPeriod`) or iteration number (`iterPeriod`).

For instance, to print kinetic energy (using *kineticEnergy*) every 5 seconds, the following engine will be put to `O.engines`:

```
PyRunner(command="print 'kinetic energy',kineticEnergy()",realPeriod=5)
```

For running more complex commands, it is convenient to define an external function and only call it from within the engine. Since the `command` is run in the script's namespace, functions defined within scripts can be called. Let us print information on interaction between bodies 0 and 1 periodically:

```
def intrInfo(id1,id2):
        try:
                i=O.interactions[id1,id2]
                # assuming it is a CpmPhys instance
                print id1,id2,i.phys.sigmaN
        except:
                # in case the interaction doesn't exist (yet?)
                print "No interaction between",id1,id2
O.engines=[...,
        PyRunner(command="intrInfo(0,1)",realPeriod=5)
]
```

More useful examples will be given below.

The *plot* module provides simple interface and storage for tracking various data. Although originally conceived for plotting only, it is widely used for tracking variables in general.

The data are in *plot.data* dictionary, which maps variable names to list of their values; the *plot.addData* function is used to add them.

```
Yade [295]: from yade import plot

Yade [296]: plot.data
Out[296]:
{'eps': [0.0001, 0.001, nan],
 'force': [nan, nan, 1000.0],
 'sigma': [12, nan, nan]}

Yade [297]: plot.addData(sigma=12,eps=1e-4)

# not adding sigma will add a NaN automatically
# this assures all variables have the same number of records
Yade [298]: plot.addData(eps=1e-3)

# adds NaNs to already existing sigma and eps columns
Yade [299]: plot.addData(force=1e3)

Yade [300]: plot.data
Out[300]:
{'eps': [0.0001, 0.001, nan, 0.0001, 0.001, nan],
 'force': [nan, nan, 1000.0, nan, nan, 1000.0],
 'sigma': [12, nan, nan, 12, nan, nan]}

# retrieve only one column
Yade [301]: plot.data['eps']
Out[301]: [0.0001, 0.001, nan, 0.0001, 0.001, nan]

# get maximum eps
Yade [302]: max(plot.data['eps'])
Out[302]: 0.001
```

New record is added to all columns at every time *plot.addData* is called; this assures that lines in different columns always match. The special value `nan` or `NaN` (Not a Number) is inserted to mark the record invalid.

Note: It is not possible to have two columns with the same name, since data are stored as a dictionary.

To record data periodically, use *PyRunner*. This will record the z coordinate and velocity of body #1, iteration number and simulation time (every 20 iterations):

```
O.engines=O.engines+[PyRunner(command='myAddData()', iterPeriod=20)]

from yade import plot
def myAddData():
        b=O.bodies[1]
        plot.addData(z1=b.state.pos[2], v1=b.state.vel.norm(), i=O.iter, t=O.time)
```

Note: Arbitrary string can be used as column label for *plot.data*. If it cannot be used as keyword name for *plot.addData* (since it is a python keyword (`for`), or has spaces inside (`my funny column`), you can pass dictionary to *plot.addData* instead:

```
plot.addData(z=b.state.pos[2],**{'my funny column':b.state.vel.norm()})
```

An exception are columns having leading of trailing whitespaces. They are handled specially in *plot.plots* and should not be used (see below).

Labels can be conveniently used to access engines in the `myAddData` function:

```
O.engines=[...,
        UniaxialStrainer(...,label='strainer')
]
def myAddData():
        plot.addData(sigma=strainer.avgStress,eps=strainer.strain)
```

In that case, naturally, the labeled object must define attributes which are used (*UniaxialStrainer.strain* and *UniaxialStrainer.avgStress* in this case).

Plotting variables

Above, we explained how to track variables by storing them using *plot.addData*. These data can be readily used for plotting. Yade provides a simple, quick to use, plotting in the *plot* module. Naturally, since direct access to underlying data is possible via *plot.data*, these data can be processed in any way.

The *plot.plots* dictionary is a simple specification of plots. Keys are x-axis variable, and values are tuple of y-axis variables, given as strings that were used for *plot.addData*; each entry in the dictionary represents a separate figure:

```
plot.plots={
        'i':('t',),     # plot t(i)
        't':('z1','v1') # z1(t) and v1(t)
}
```

Actual plot using data in *plot.data* and plot specification of *plot.plots* can be triggered by invoking the *plot.plot* function.

Live updates of plots

Yade features live-updates of figures during calculations. It is controlled by following settings:

- *plot.live* - By setting `yade.plot.live=True` you can watch the plot being updated while the calculations run. Set to `False` otherwise.
- *plot.liveInterval* - This is the interval in seconds between the plot updates.
- *plot.autozoom* - When set to `True` the plot will be automatically rezoomed.

Controlling line properties

In this subsection let us use a *basic complete script* like examples/simple-scene/simple-scene-plot.py, which we will later modify to make the plots prettier. Line of interest from that file is, and generates a picture presented below:

```
plot.plots={'i':('t'),'t':('z_sph',None,('v_sph','go-'),'z_sph_half')}
```

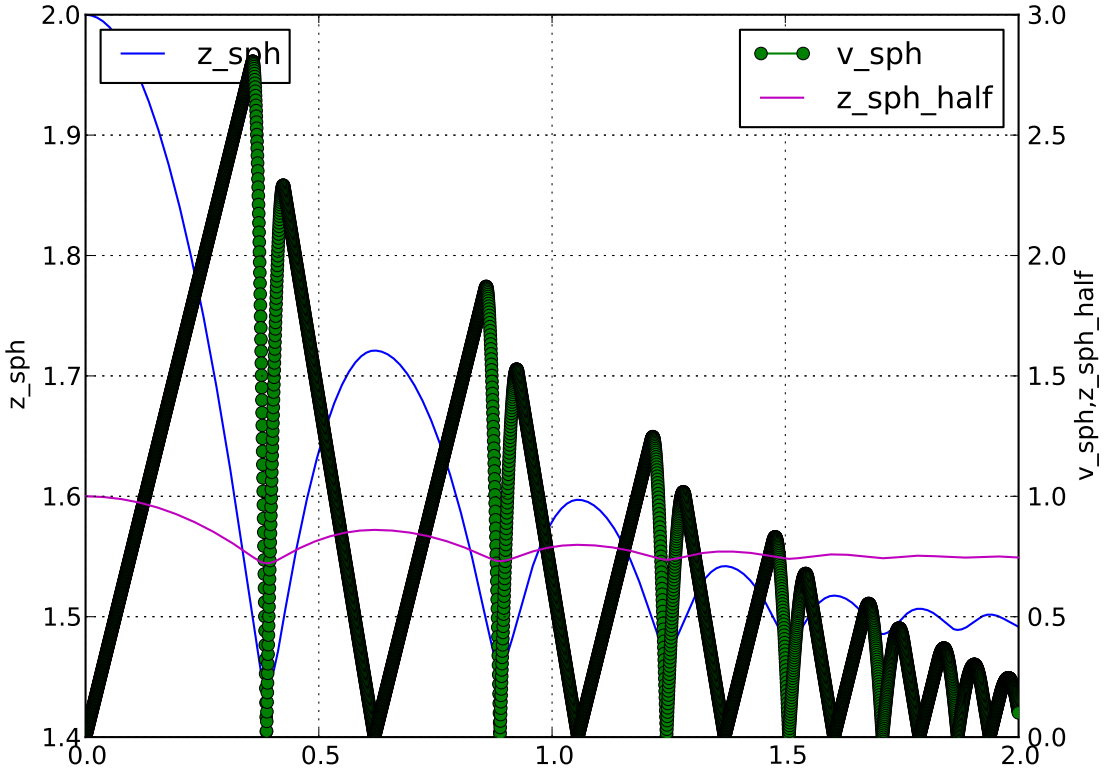


Fig. 3.5: Figure generated by examples/simple-scene/simple-scene-plot.py.

The line plots take an optional second string argument composed of a line color (eg. `'r'`, `'g'` or `'b'`), a line style (eg. `'-'`, `'--'` or `':'`) and a line marker (`'o'`, `'s'` or `'d'`). A red dotted line with circle markers is created with 'ro:' argument. For a listing of all options please have a look at http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot

For example using following plot.plots() command, will produce a following graph:

```
plot.plots={'i':(('t','xr:'),),'t':(('z_sph','r:'),None,('v_sph','g--'),('z_sph_half','b-.'))}
```

And this one will produce a following graph:

```
plot.plots={'i':(('t','xr:'),),'t':(('z_sph','Hr:'),None,('v_sph','+g--'),('z_sph_half','*b-.'))}
```

Note: You can learn more in matplotlib tutorial http://matplotlib.sourceforge.net/users/pyplot_tutorial.html and documentation http://matplotlib.sourceforge.net/users/pyplot_tutorial.html#controlling-line-properties

Note: Please note that there is an extra `,` in `'i':(('t','xr:'),)`, otherwise the `'xr:'` wouldn't be recognized as a line style parameter, but would be treated as an extra data to plot.

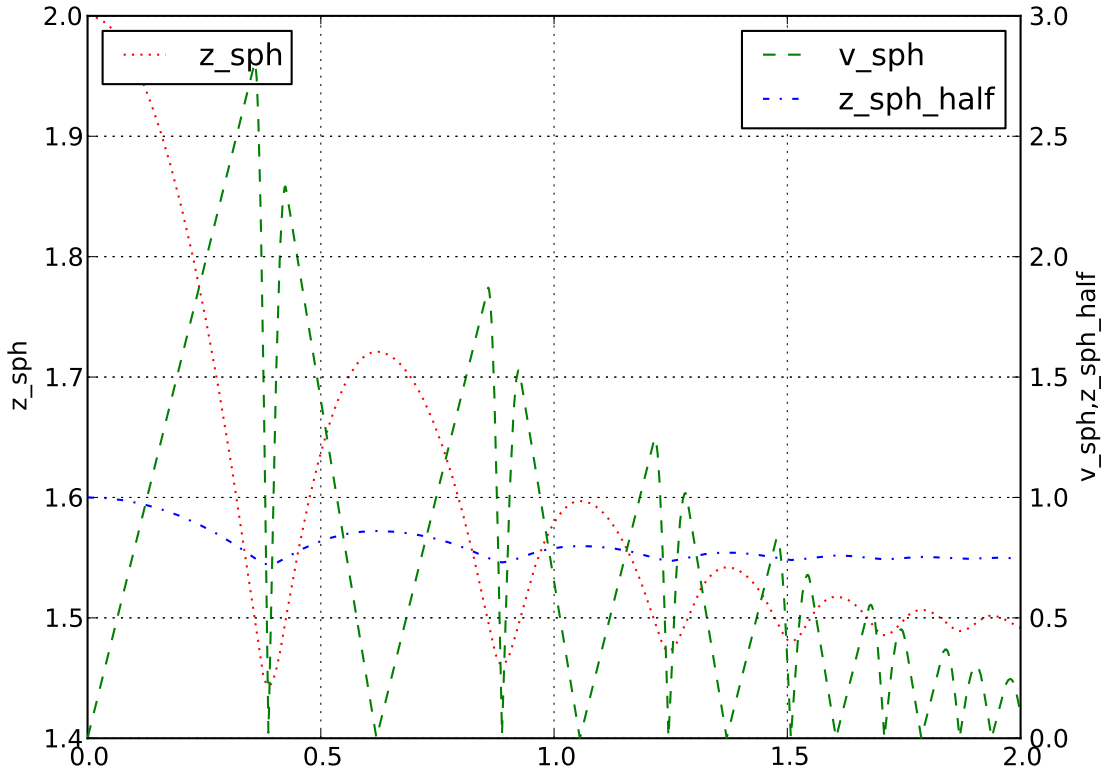


Fig. 3.6: Figure generated by changing parameters to plot.plots as above.

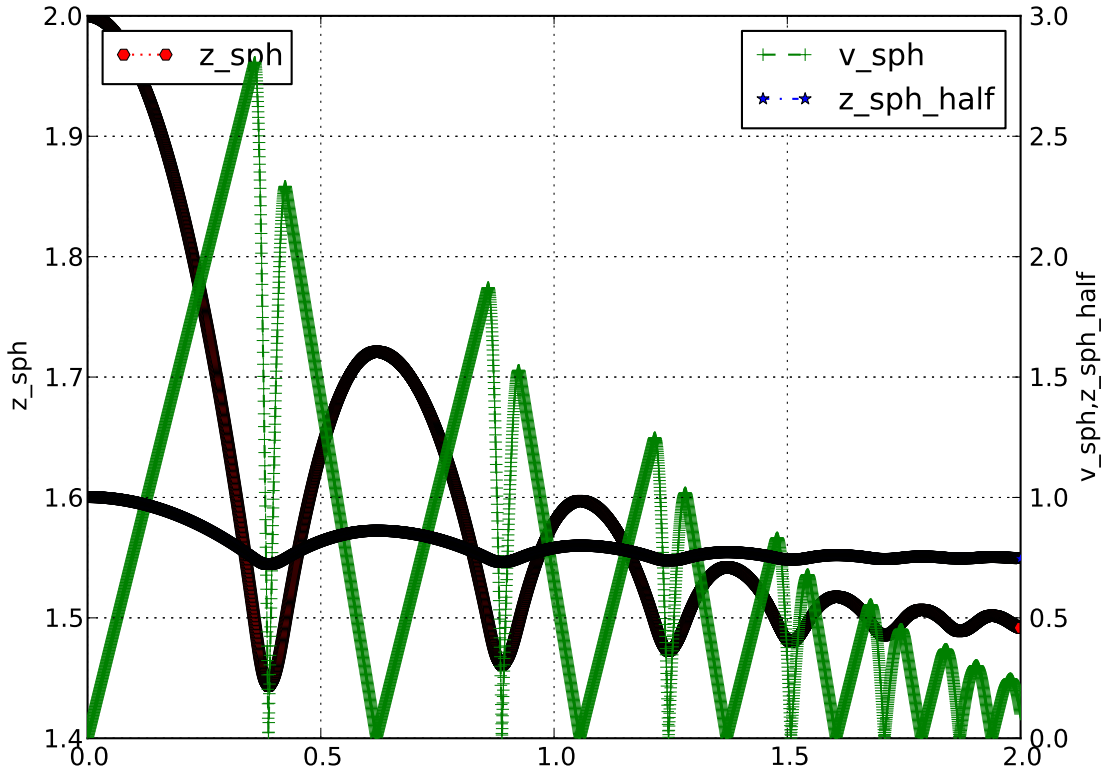


Fig. 3.7: Figure generated by changing parameters to plot.plots as above.

Controlling text labels

It is possible to use TeX syntax in plot labels. For example using following two lines in examples/simple-scene/simple-scene-plot.py, will produce a following picture:

```
plot.plots={'i':(('t','xr:'),),'t':(('z_sph','r:'),None,('v_sph','g--'),('z_sph_half','b-.'))}
plot.labels={'z_sph':'$z_{sph}$' , 'v_sph':'$v_{sph}$' , 'z_sph_half':'$z_{sph}/2$'}
```

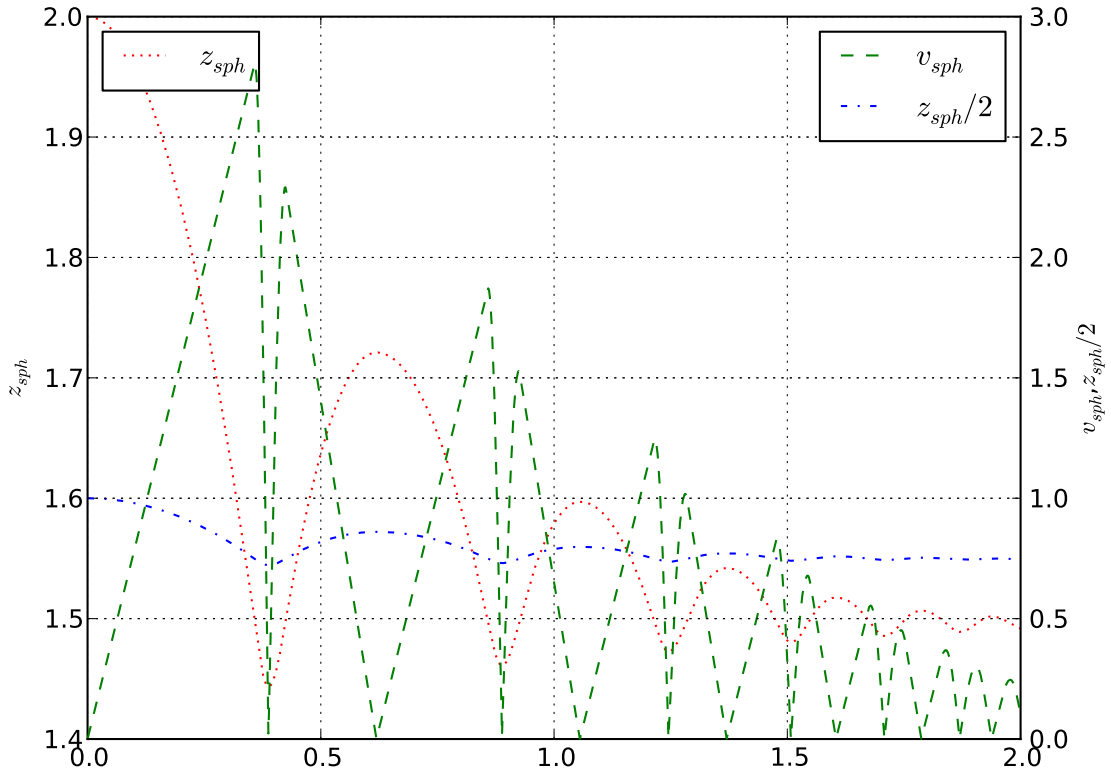


Fig. 3.8: Figure generated by examples/simple-scene/simple-scene-plot.py, with TeX labels.

Greek letters are simply a `'$\alpha$'`, `'$\beta$'` etc. in those labels. To change the font style a following command could be used:

```
yade.plot.matplotlib.rc('mathtext', fontset='stixsans')
```

But this is not part of yade, but a part of matplotlib, and if you want something more complex you really should have a look at matplotlib users manual http://matplotlib.sourceforge.net/users/index.html

Multiple figures

Since *plot.plots* is a dictionary, multiple entries with the same key (x-axis variable) would not be possible, since they overwrite each other:

```
Yade [303]: plot.plots={
   .....:    'i':('t',),
   .....:    'i':('z1','v1')
   .....: }
   .....:
```

```
Yade [304]: plot.plots
Out[304]: {'i': ('z1', 'v1')}
```

You can, however, distinguish them by prepending/appending space to the x-axis variable, which will be removed automatically when looking for the variable in *plot.data* – both x-axes will use the `i` column:

```
Yade [305]: plot.plots={
   .....:    'i':('t',),
   .....:    'i ':('z1','v1') # note the space in 'i '
   .....: }
   .....:

Yade [306]: plot.plots
Out[306]: {'i': ('t',), 'i ': ('z1', 'v1')}
```

Split y1 y2 axes

To avoid big range differences on the `y` axis, it is possible to have left and right `y` axes separate (like `axes x1y2` in gnuplot). This is achieved by inserting `None` to the plot specifier; variables coming before will be plot normally (on the left y-axis), while those after will appear on the right:

```
plot.plots={'i':('z1',None,'v1')}
```

Exporting

Plots can be exported to external files for later post-processing via that *plot.saveGnuplot* function. Note that all data you added via plot.addData is saved - even data that you don't plot live during simulation. By editing the generated .gnuplot file you can plot any of the added Data afterwards.

- Data file is saved (compressed using bzip2) separately from the gnuplot file, so any other programs can be used to process them. In particular, the `numpy.genfromtxt` (documented here) can be useful to import those data back to python; the decompression happens automatically.
- The gnuplot file can be run through gnuplot to produce the figure; see *plot.saveGnuplot* documentation for details.

3.2.2 Stop conditions

For simulations with pre-determined number of steps, number of steps can be prescribed:

```
# absolute iteration number O.stopAtIter=35466 O.run() O.wait()
```

or

```
# number of iterations to run from now
O.run(35466,True) # wait=True
```

causes the simulation to run 35466 iterations, then stopping.

Frequently, decisions have to be made based on evolution of the simulation itself, which is not yet known. In such case, a function checking some specific condition is called periodically; if the condition is satisfied, `O.pause` or other functions can be called to stop the stimulation. See documentation for *Omega.run*, *Omega.pause*, *Omega.step*, *Omega.stopAtIter* for details.

For simulations that seek static equilibrium, the *unbalancedForce* can provide a useful metrics (see its documentation for details); for a desired value of `1e-2` or less, for instance, we can use:

```
def checkUnbalanced():
        if unbalancedForce<1e-2: O.pause()

O.engines=O.engines+[PyRunner(command="checkUnbalanced()",iterPeriod=100)]
```

```
# this would work as well, without the function defined apart:
#   PyRunner(command="if unablancedForce<1e-2: O.pause()",iterPeriod=100)

O.run(); O.wait()
# will continue after O.pause() will have been called
```

Arbitrary functions can be periodically checked, and they can also use history of variables tracked via *plot.addData*. For example, this is a simplified version of damage control in examples/concrete/uniax.py; it stops when current stress is lower than half of the peak stress:

```
O.engines=[...,
        UniaxialStrainer=(...,label='strainer'),
        PyRunner(command='myAddData()',iterPeriod=100),
        PyRunner(command='stopIfDamaged()',iterPeriod=100)
]

def myAddData():
        plot.addData(t=O.time,eps=strainer.strain,sigma=strainer.stress)

def stopIfDamaged():
        currSig=plot.data['sigma'][-1] # last sigma value
        maxSig=max(plot.data['sigma']) # maximum sigma value
        # print something in any case, so that we know what is happening
        print plot.data['eps'][-1],currSig
        if currSig<.5*maxSig:
                print "Damaged, stopping"
                print 'gnuplot',plot.saveGnuplot(O.tags['id'])
                import sys
                sys.exit(0)

O.run(); O.wait()
# this place is never reached, since we call sys.exit(0) directly
```

Checkpoints

Occasionally, it is useful to revert to simulation at some past point and continue from it with different parameters. For instance, tension/compression test will use the same initial state but load it in 2 different directions. Two functions, *Omega.saveTmp* and *Omega.loadTmp* are provided for this purpose; *memory* is used as storage medium, which means that saving is faster, and also that the simulation will disappear when Yade finishes.

```
O.saveTmp()
# do something
O.saveTmp('foo')
O.loadTmp()      # loads the first state
O.loadTmp('foo') # loads the second state
```

Warning: `O.loadTmp` cannot be called from inside an engine, since *before* loading a simulation, the old one must finish the current iteration; it would lead to deadlock, since `O.loadTmp` would wait for the current iteration to finish, while the current iteration would be blocked on `O.loadTmp`.
A special trick must be used: a separate function to be run after the current iteration is defined and is invoked from an independent thread launched only for that purpose:

```
O.engines=[...,PyRunner('myFunc()',iterPeriod=345)]

def myFunc():
        if someCondition:
                import thread
                # the () are arguments passed to the function
                thread.start_new_thread(afterIterFunc,())
def afterIterFunc():
        O.pause(); O.wait() # wait till the iteration really finishes
        O.loadTmp()

O.saveTmp()
O.run()
```

3.2.3 Remote control

Yade can be controlled remotely over network. At yade startup, the following lines appear, among other messages:

```
TCP python prompt on localhost:9000, auth cookie `dcekyu'
TCP info provider on localhost:21000
```

They inform about 2 ports on which connection of 2 different kind is accepted.

Python prompt

`TCP python prompt` is telnet server with authenticated connection, providing full python command-line. It listens on port 9000, or higher if already occupied (by another yade instance, for example).

Using the authentication cookie, connection can be made using telnet:

```
$ telnet localhost 9000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Enter auth cookie: dcekyu
__   __    ____                  __  _____ ____ ____
\ \ / /_ _|  _ \  ___    ___   / / |_   _/ ___|  _ \
 \ V / _` | | | |/ _ \  / _ \ / /    | || |   | |_) |
  | | (_| | |_| |  __/ | (_) / /     | || |___|  __/
  |_|\__,_|____/ \___|  \___/_/      |_| \____|_|

(connected from 127.0.0.1:40372)
>>>
```

The python pseudo-prompt `>>>` lets you write commands to manipulate simulation in variety of ways as usual. Two things to notice:

1. The new python interpreter (`>>>`) lives in a namespace separate from `Yade [1]:` command-line. For your convenience, `from yade import *` is run in the new python instance first, but local and global variables are not accessible (only builtins are).
2. The (fake) `>>>` interpreter does not have rich interactive feature of IPython, which handles the usual command-line `Yade [1]:`; therefore, you will have no command history, `?` help and so on.

Note: By giving access to python interpreter, full control of the system (including reading user's files) is possible. For this reason, **connection are only allowed from localhost**, not over network remotely. Of course you can log into the system via SSH over network to get remote access.

Warning: Authentication cookie is trivial to crack via bruteforce attack. Although the listener stalls for 5 seconds after every failed login attempt (and disconnects), the cookie could be guessed by trial-and-error during very long simulations on a shared computer.

Info provider

`TCP Info provider` listens at port 21000 (or higher) and returns some basic information about current simulation upon connection; the connection terminates immediately afterwards. The information is python dictionary represented as string (serialized) using standard pickle module.

This functionality is used by the batch system (described below) to be informed about individual simulation progress and estimated times. If you want to access this information yourself, you can study core/main/yade-batch.in for details.

3.2.4 Batch queuing and execution (yade-batch)

Yade features light-weight system for running one simulation with different parameters; it handles assignment of parameter values to python variables in simulation script, scheduling jobs based on number of available and required cores and more. The whole batch consists of 2 files:

simulation script regular Yade script, which calls *readParamsFromTable* to obtain parameters from parameter table. In order to make the script runnable outside the batch, *readParamsFromTable* takes default values of parameters, which might be overridden from the parameter table.

readParamsFromTable knows which parameter file and which line to read by inspecting the `PARAM_TABLE` environment variable, set by the batch system.

parameter table simple text file, each line representing one parameter set. This file is read by *readParamsFromTable* (using *TableParamReader* class), called from simulation script, as explained above. For better reading of the text file you can make use of tabulators, these will be ignored by *readParamsFromTable*. Parameters are not restricted to numerical values. You can also make use of strings by "quoting" them (' ' may also be used instead of " "). This can be useful for nominal parameters.

The batch can be run as

```
yade-batch parameters.table simulation.py
```

and it will intelligently run one simulation for each parameter table line. A minimal example is found in examples/test/batch/params.table and examples/test/batch/sim.py, another example follows.

Example

Suppose we want to study influence of parameters *density* and *initialVelocity* on position of a sphere falling on fixed box. We create parameter table like this:

```
description density initialVelocity # first non-empty line are column headings
reference   2400    10
hi_v           =    20              # = to use value from previous line
lo_v           =     5
# comments are allowed
hi_rho      5000    10
# blank lines as well:

```

```
hi_rho_v       =    20
hi_rh0_lo_v    =     5
```

Each line give one combination of these 2 parameters and assigns (which is optional) a *description* of this simulation.

In the simulation file, we read parameters from table, at the beginning of the script; each parameter has default value, which is used if not specified in the parameters file:

```
readParamsFromTable(
        gravity=-9.81,
        density=2400,
        initialVelocity=20,
        noTableOk=True      # use default values if not run in batch
)
from yade.params.table import *
print gravity, density, initialVelocity
```

after the call to *readParamsFromTable*, corresponding python variables are created in the `yade.params.table` module and can be readily used in the script, e.g.

```
GravityEngine(gravity=(0,0,gravity))
```

Let us see what happens when running the batch:

```
$ yade-batch batch.table batch.py
Will run `/usr/local/bin/yade-trunk' on `batch.py' with nice value 10, output redirected to `batch.@.log', 4 jo
Will use table `batch.table', with available lines 2, 3, 4, 5, 6, 7.
Will use lines  2 (reference), 3 (hi_v), 4 (lo_v), 5 (hi_rho), 6 (hi_rho_v), 7 (hi_rh0_lo_v).
Master process pid 7030
```

These lines inform us about general batch information: nice level, log file names, how many cores will be used (4); table name, and line numbers that contain parameters; finally, which lines will be used; master PID is useful for killing (stopping) the whole batch with the `kill` command.

```
Job summary:
   #0 (reference/4): PARAM_TABLE=batch.table:2 DISPLAY=  /usr/local/bin/yade-trunk --threads=4 --nice=10 -x bat
   #1 (hi_v/4): PARAM_TABLE=batch.table:3 DISPLAY=  /usr/local/bin/yade-trunk --threads=4 --nice=10 -x batch.py
   #2 (lo_v/4): PARAM_TABLE=batch.table:4 DISPLAY=  /usr/local/bin/yade-trunk --threads=4 --nice=10 -x batch.py
   #3 (hi_rho/4): PARAM_TABLE=batch.table:5 DISPLAY=  /usr/local/bin/yade-trunk --threads=4 --nice=10 -x batch.
   #4 (hi_rho_v/4): PARAM_TABLE=batch.table:6 DISPLAY=  /usr/local/bin/yade-trunk --threads=4 --nice=10 -x batc
   #5 (hi_rh0_lo_v/4): PARAM_TABLE=batch.table:7 DISPLAY=  /usr/local/bin/yade-trunk --threads=4 --nice=10 -x ba
```

displays all jobs with command-lines that will be run for each of them. At this moment, the batch starts to be run.

```
#0 (reference/4) started on Tue Apr 13 13:59:32 2010
#0 (reference/4) done    (exit status 0), duration 00:00:01, log batch.reference.log
#1 (hi_v/4) started on Tue Apr 13 13:59:34 2010
#1 (hi_v/4) done    (exit status 0), duration 00:00:01, log batch.hi_v.log
#2 (lo_v/4) started on Tue Apr 13 13:59:35 2010
#2 (lo_v/4) done    (exit status 0), duration 00:00:01, log batch.lo_v.log
#3 (hi_rho/4) started on Tue Apr 13 13:59:37 2010
#3 (hi_rho/4) done    (exit status 0), duration 00:00:01, log batch.hi_rho.log
#4 (hi_rho_v/4) started on Tue Apr 13 13:59:38 2010
#4 (hi_rho_v/4) done    (exit status 0), duration 00:00:01, log batch.hi_rho_v.log
#5 (hi_rh0_lo_v/4) started on Tue Apr 13 13:59:40 2010
#5 (hi_rh0_lo_v/4) done    (exit status 0), duration 00:00:01, log batch.hi_rh0_lo_v.log
```

information about job status changes is being printed, until:

```
All jobs finished, total time  00:00:08
Log files:
batch.reference.log batch.hi_v.log batch.lo_v.log batch.hi_rho.log batch.hi_rho_v.log batch.hi_rh0_lo_v.log
Bye.
```

Separating output files from jobs

As one might output data to external files during simulation (using classes such as *VTKRecorder*), it is important to name files in such way that they are not overwritten by next (or concurrent) job in the same batch. A special tag `O.tags['id']` is provided for such purposes: it is comprised of date, time and PID, which makes it always unique (e.g. `20100413T144723p7625`); additional advantage is that alphabetical order of the `id` tag is also chronological. To add the used parameter set or the description of the job, if set, you could add O.tags['params'] to the filename.

For smaller simulations, prepending all output file names with `O.tags['id']` can be sufficient:

```
saveGnuplot(O.tags['id'])
```

For larger simulations, it is advisable to create separate directory of that name first, putting all files inside afterwards:

```
os.mkdir(O.tags['id'])
O.engines=[
        # …
        VTKRecorder(fileName=O.tags['id']+'/'+'vtk'),
        # …
]
# …
O.saveGnuplot(O.tags['id']+'/'+'graph1')
```

Controlling parallel computation

Default total number of available cores is determined from `/proc/cpuinfo` (provided by Linux kernel); in addition, if `OMP_NUM_THREADS` environment variable is set, minimum of these two is taken. The `-j`/`--jobs` option can be used to override this number.

By default, each job uses all available cores for itself, which causes jobs to be effectively run in parallel. Number of cores per job can be globally changed via the `--job-threads` option.

Table column named `!OMP_NUM_THREADS` (`!` prepended to column generally means to assign *environment variable*, rather than python variable) controls number of threads for each job separately, if it exists.

If number of cores for a job exceeds total number of cores, warning is issued and only the total number of cores is used instead.

Merging gnuplot from individual jobs

Frequently, it is desirable to obtain single figure for all jobs in the batch, for comparison purposes. Somewhat heiristic way for this functionality is provided by the batch system. `yade-batch` must be run with the `--gnuplot` option, specifying some file name that will be used for the merged figure:

```
yade-trunk --gnuplot merged.gnuplot batch.table batch.py
```

Data are collected in usual way during the simulation (using *plot.addData*) and saved to gnuplot file via *plot.saveGnuplot* (it creates 2 files: gnuplot command file and compressed data file). The batch system *scans*, once the job is finished, log file for line of the form `gnuplot [something]`. Therefore, in order to print this *magic line* we put:

```
print 'gnuplot',plot.saveGnuplot(O.tags['id'])
```

and the end of the script (even after waitIfBatch()) , which prints:

```
gnuplot 20100413T144723p7625.gnuplot
```

to the output (redirected to log file).

This file itself contains single graph:

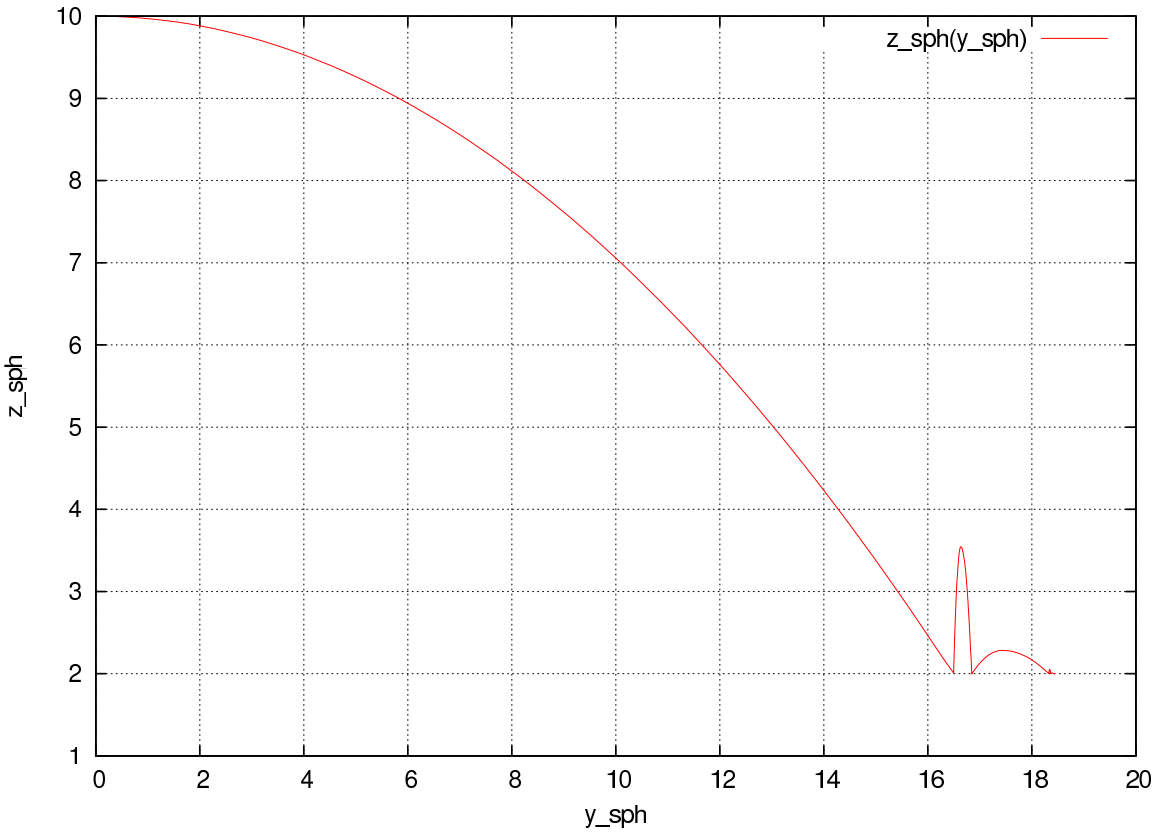


Fig. 3.9: Figure from single job in the batch.

At the end, the batch system knows about all gnuplot files and tries to merge them together, by assembling the `merged.gnuplot` file.

HTTP overview

While job is running, the batch system presents progress via simple HTTP server running at port 9080, which can be acessed from regular web browser by requesting the `http://localhost:9080` URL. This page can be accessed remotely over network as well.

3.3 Postprocessing

3.3.1 3d rendering & videos

There are multiple ways to produce a video of simulation:

1. Capture screen output (the 3d rendering window) during the simulation — there are tools available for that (such as Istanbul or RecordMyDesktop, which are also packaged for most Linux distributions). The output is "what you see is what you get", with all the advantages and disadvantages.
2. Periodic frame snapshot using *SnapshotEngine* (see examples/bulldozer/bulldozer.py for a full example):

```
O.engines=[
  #...
  SnapshotEngine(iterPeriod=100,fileBase='/tmp/bulldozer-',viewNo=0,label='snapshooter')
]
```

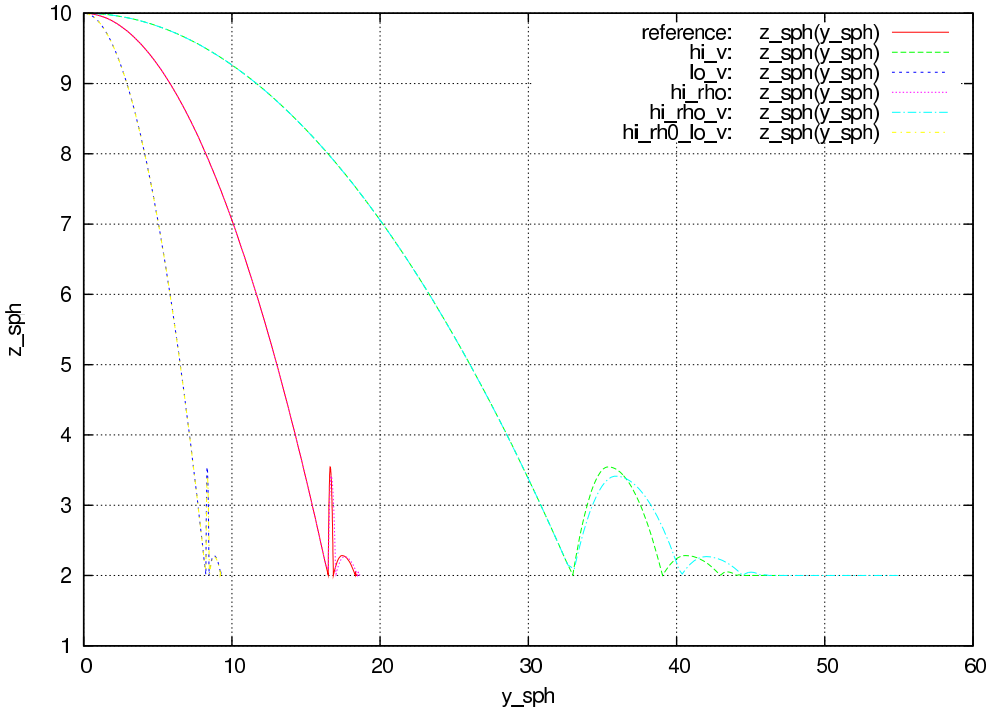


Fig. 3.10: Merged figure from all jobs in the batch. Note that labels are prepended by job description to make lines distinguishable.

Running for 00:10:19, since Tue Apr 13 16:17:11 2010.

Pid 9873

4 slots available, 4 used, 0 free.

Jobs

4 total, **2** running, **1** done

id	status	info	slots	command
_geomType=B	00:10:19	96.33% done step 9180/9530 avg 14.9596/sec 10267 bodies 65506 intrs	2	PARAM_TABLE=iParams.table:2 DISPLAY= /usr/local/bin/yade-trunk --threads=2 --nice=10 -x indent.py > indent._geomType=B.log 2>&1
_geomType=smallA	00:09:53	(no info)	2	PARAM_TABLE=iParams.table:3 DISPLAY= /usr/local/bin/yade-trunk --threads=2 --nice=10 -x indent.py > indent._geomType=smallA.log 2>&1
_geomType=smallB	00:00:24	6.95% done step 694/9985 avg 35.8212/sec 9021 bodies 58352 intrs	2	PARAM_TABLE=iParams.table:4 DISPLAY= /usr/local/bin/yade-trunk --threads=2 --nice=10 -x indent.py > indent._geomType=smallB.log 2>&1
_geomType=smallC	(pending)	(no info)	2	PARAM_TABLE=iParams.table:5 DISPLAY= /usr/local/bin/yade-trunk --threads=2 --nice=10 -x indent.py > indent._geomType=smallC.log 2>&1

Fig. 3.11: Summary page available at port 9080 as batch is processed (updates every 5 seconds automatically). Possible job statuses are pending, running, done, failed.

which will save numbered files like `/tmp/bulldozer-0000.png`. These files can be processed externally (with mencoder and similar tools) or directly with the *makeVideo*:

```
makeVideo(frameSpec,out,renameNotOverwrite=True,fps=24,kbps=6000,bps=None)
```

The video is encoded using the default mencoder codec (mpeg4).

3. Specialized post-processing tools, notably Paraview. This is described in more detail in the following section.

Paraview

Saving data during the simulation

Paraview is based on the Visualization Toolkit, which defines formats for saving various types of data. One of them (with the `.vtu` extension) can be written by a special engine *VTKRecorder*. It is added to the simulation loop:

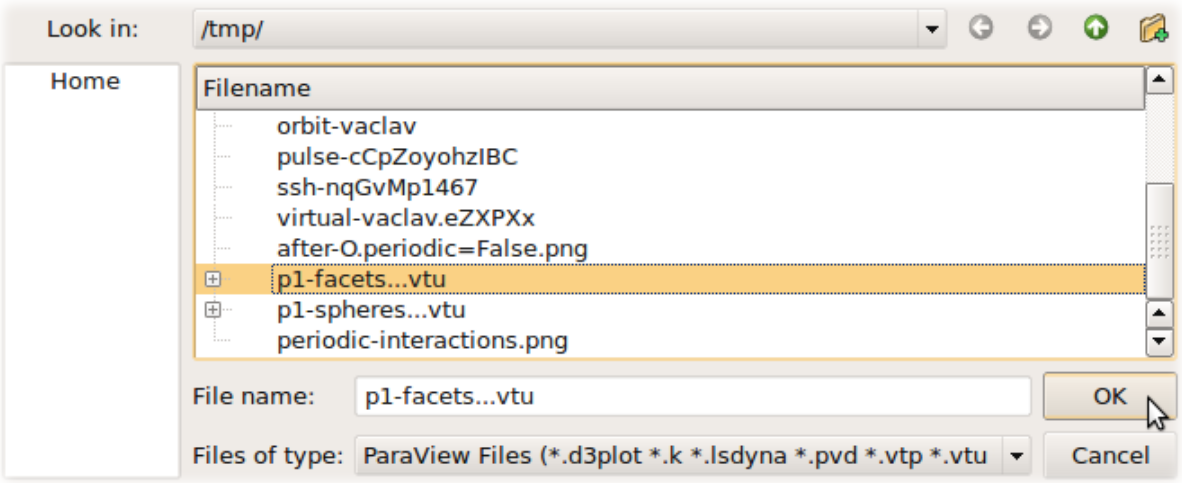
```
O.engines=[
        # ...
        VTKRecorder(iterPeriod=100,recorders=['spheres','facets','colors'],fileName='/tmp/p1-')
]
```

- *iterPeriod* determines how often to save simulation data (besides *iterPeriod*, you can also use *virtPeriod* or *realPeriod*). If the period is too high (and data are saved only few times), the video will have few frames.
- *fileName* is the prefix for files being saved. In this case, output files will be named `/tmp/p1-spheres.0.vtu` and `/tmp/p1-facets.0.vtu`, where the number is the number of iteration; many files are created, putting them in a separate directory is advisable.
- *recorders* determines what data to save

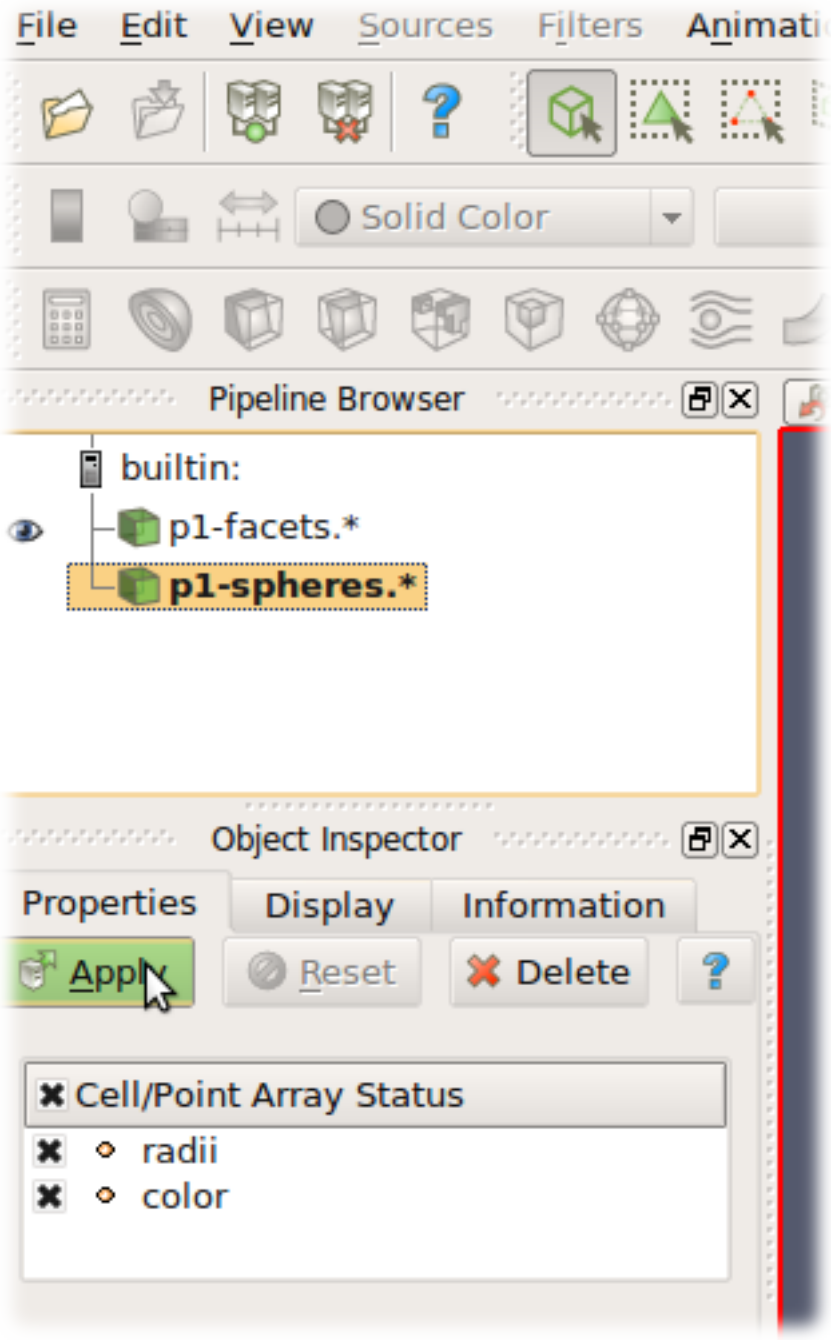
exporter.VTKExporter plays a similar role, with the difference that it is more flexible. It will save any user defined variable associated to the bodies.

Loading data into Paraview

All sets of files (`spheres`, `facets`, ...) must be opened one-by-one in Paraview. The open dialogue automatically collapses numbered files in one, making it easy to select all of them:



Click on the "Apply" button in the "Object inspector" sub-window to make loaded objects visible. You can see tree of displayed objects in the "Pipeline browser":


File
Edit
View
Sources
Filters
Animati
Solid Color
Pipeline Browser
builtin:
p1-facets.*
p1-spheres.*
Object Inspector
Properties
Display
Information
Apply
Reset
Delete
Cell/Point Array Status
radii
color

Rendering spherical particles. Glyphs Spheres will only appear as points. To make them look as spheres, you have to add "glyph" to the `p1-spheres.*` item in the pipeline using the  icon. Then set (in the Object inspector)

- "Glyph type" to *Sphere*
- "Radius" to *1*
- "Scale mode" to *Scalar* (*Scalar* is set above to be the *radii* value saved in the file, therefore spheres with radius *1* will be scaled by their true radius)
- "Set scale factor" to *1*
- optionally uncheck "Mask points" and "Random mode" (they make some particles not to be rendered for performance reasons, controlled by the "Maximum Number of Points")

After clicking "Apply", spheres will appear. They will be rendered over the original white points, which you can disable by clicking on the eye icon next to `p1-spheres.*` in the Pipeline browser.

Rendering spherical particles. PointSprite Another opportunity to display spheres is an using *PointSprite* plugin. This technique requires much less RAM in comparison to Glyphs.

- "Tools -> Manage Plugins"
- "PointSprite_Plugin -> Load selected -> Close"
- Load VTU-files
- "Representation -> Point Sprite"
- "Point Sprite -> Scale By -> radii"
- "Edit Radius Transfer Function -> Proportional -> Multiplier = 1.0 -> Close"

Facet transparency If you want to make facet objects transparent, select `p1-facets.*` in the Pipeline browser, then go to the Object inspector on the Display tab. Under "Style", you can set the "Opacity" value to something smaller than 1.

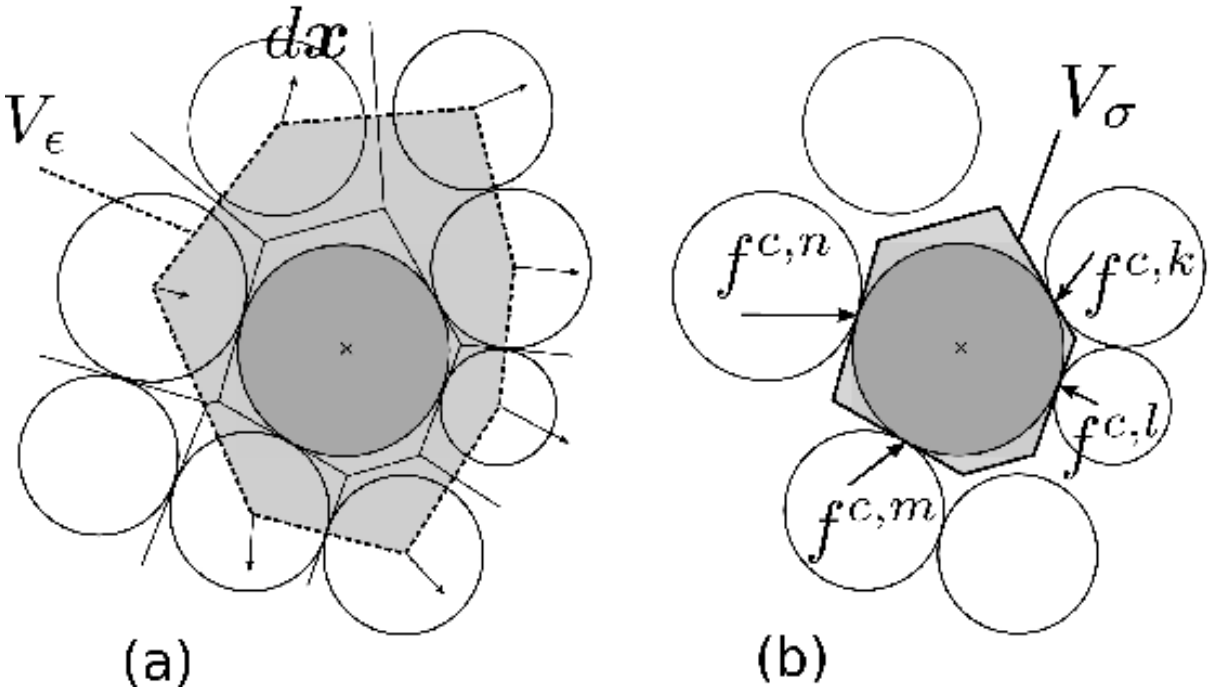
Animation You can move between frames (snapshots that were saved) via the "Animation" menu. After setting the view angle, zoom etc to your satisfaction, the animation can be saved with *File/Save animation*.

3.3.2 Micro-stress and micro-strain

It is sometimes useful to visualize a DEM simulation through equivalent strain fields or stress fields. This is possible with *TesselationWrapper*. This class handles the triangulation of spheres in a scene, build tesselation on request, and give access to computed quantities: volume, porosity and local deformation for each sphere. The definition of microstrain and microstress is at the scale of particle-centered subdomains shown below, as explained in *[Catalano2014a]* .

Micro-strain

Below is an output of the *defToVtk* function visualized with paraview (in this case Yade's TesselationWrapper was used to process experimental data obtained on sand by Edward Ando at Grenoble University, 3SR lab.). The output is visualized with paraview, as explained in the previous section. Similar results can be generated from simulations:



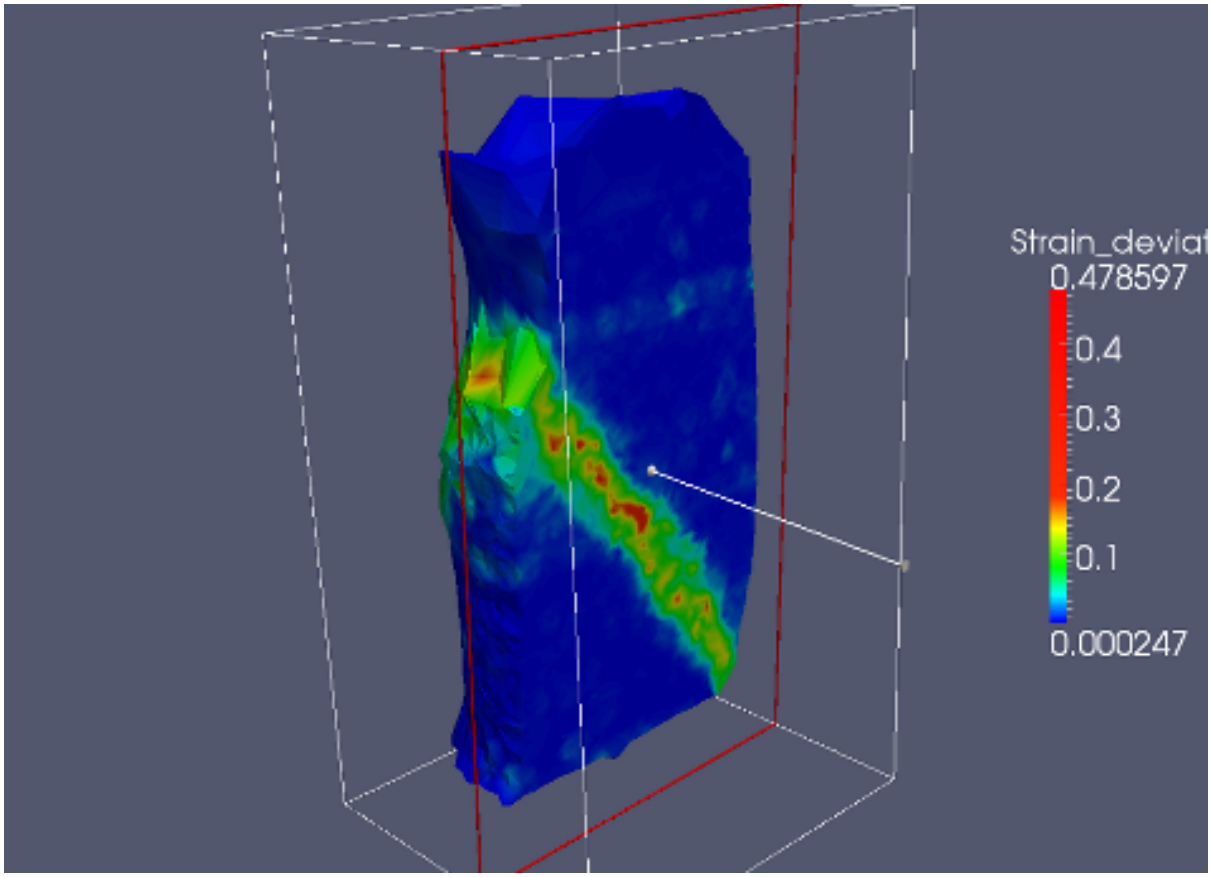

```
tt=TriaxialTest()
tt.generate("test.yade")
O.load("test.yade")
O.run(100,True)
TW=TesselationWrapper()
TW.triangulate()        #compute regular Delaunay triangulation, don't construct tesselation
TW.computeVolumes()     #will silently tesselate the packing, then compute volume of each Voronoi cell
TW.volume(10)           #get volume associated to sphere of id 10
TW.setState(0)          #store current positions internaly for later use as the "0" state
O.run(100,True)         #make particles move a little (let's hope they will!)
TW.setState(1)          #store current positions internaly in the "1" (deformed) state
#Now we can define strain by comparing states 0 and 1, and average them at the particles scale
TW.defToVtk("strain.vtk")
```

Micro-stress

Stress fields can be generated by combining the volume returned by TesselationWrapper to per-particle stress given by *bodyStressTensors*. Since the stress σ from bodyStressTensor implies a division by the volume V_b of the solid particle, one has to re-normalize it in order to obtain the micro-stress as defined in *[Catalano2014a]* (equation 39 therein), i.e. $\overline{\sigma}^k = \sigma^k \times V_b^k/V_\sigma^k$ where V_σ^k is the volume assigned to particle k in the tesselation. For instance:

```
#"b" being a body
TW=TesselationWrapper()
TW.computeVolumes()
s=bodyStressTensors()
stress = s[b.id]**4.*pi/3.*b.shape.radius**3/TW.volume(b.id)
```

As any other value, the stress can be exported to a vtk file for display in Paraview using *export.VTKExporter*.



3.4 Python specialties and tricks

3.4.1 Importing Yade in other Python applications

Yade can be imported in other Python applications. To do so, you need somehow to make yade executable .py extended. The easiest way is to create a symbolic link, i.e. (suppose your Yade executable file is called "yade-trunk" and you want make it "yadeimport.py"):

```
$ cd /path/where/you/want/yadeimport
$ ln -s /path/to/yade/executable/yade-trunk yadeimport.py
```

Then you need to make your yadeimport.py findable by Python. You can export PYTHONPATH environment variable, or simply use sys.path directly in Python script:

```
import sys
sys.path.append('/path/where/you/want/yadeimport')
from yadeimport import *

print Matrix3(1,2,3, 4,5,6, 7,8,9)
print O.bodies
# any other Yade code
```

3.5 Extending Yade

- new particle shape
- new constitutive law

3.6 Troubleshooting

3.6.1 Crashes

It is possible that you encounter crash of Yade, i.e. Yade terminates with error message such as

```
Segmentation fault (core dumped)
```

without further explanation. Frequent causes of such conditions are

- program error in Yade itself;
- fatal condition in your particular simulation (such as impossible dispatch);
- problem with graphics card driver.

Try to reproduce the error (run the same script) with debug-enabled version of Yade. Debugger will be automatically launched at crash, showing backtrace of the code (in this case, we triggered crash by hand):

```
Yade [1]: import os,signal
Yade [2]: os.kill(os.getpid(),signal.SIGSEGV)
SIGSEGV/SIGABRT handler called; gdb batch file is `/tmp/yade-YwtfRY/tmp-0'
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
[Thread debugging using libthread_db enabled]
[New Thread 0x7f0fb1268710 (LWP 16471)]
[New Thread 0x7f0fb29f2710 (LWP 16470)]
[New Thread 0x7f0fb31f3710 (LWP 16469)]


…
```

What looks as cryptic message is valuable information for developers to locate source of the bug. In particular, there is (usually) line `<signal handler called>`; lines below it are source of the bug (at least very likely so):

```
Thread 1 (Thread 0x7f0fcee53700 (LWP 16465)):
#0  0x00007f0fcd8f4f7d in __libc_waitpid (pid=16497, stat_loc=<value optimized out>, options=0) at ../sysdeps/u
#1  0x00007f0fcd88c7e9 in do_system (line=<value optimized out>) at ../sysdeps/posix/system.c:149
#2  0x00007f0fcd88cb20 in __libc_system (line=<value optimized out>) at ../sysdeps/posix/system.c:190
#3  0x00007f0fcd0b4b23 in crashHandler (sig=11) at core/main/pyboot.cpp:45
#4  <signal handler called>
#5  0x00007f0fcd87ed57 in kill () at ../sysdeps/unix/syscall-template.S:82
#6  0x000000000051336d in posix_kill (self=<value optimized out>, args=<value optimized out>) at ../Modules/pos
#7  0x00000000004a7c5e in call_function (f=Frame 0x1c54620, for file <ipython console>, line 1, in <module> (),
#8  PyEval_EvalFrameEx (f=Frame 0x1c54620, for file <ipython console>, line 1, in <module> (), throwflag=<value
```

If you think this might be error in Yade, file a bug report as explained below. Do not forget to attach *full* yade output from terminal, including startup messages and debugger output – select with right mouse button, with middle button paste the bugreport to a file and attach it. Attach your simulation script as well.

3.6.2 Reporting bugs

Bugs are general name for defects (functionality shortcomings, misdocumentation, crashes) or feature requests. They are tracked at http://bugs.launchpad.net/yade.

When reporting a new bug, be as specific as possible; state version of yade you use, system version and so on, as explained in the above section on crashes.

3.6.3 Getting help

Mailing lists

Yade has two mailing-lists. Both are hosted at http://www.launchpad.net and before posting, you must register to Launchpad and subscribe to the list by adding yourself to "team" of the same name running the list.

yade-users@lists.launchpad.net is general help list for Yade users. Add yourself to yade-users team so that you can post messages. List archive is available.

yade-dev@lists.launchpad.net is for discussions about Yade development; you must be member of yade-dev team to post. This list is archived as well.

Read How To Ask Questions The Smart Way before posting. Do not forget to state what *version* of yade you use (shown when you start yade), what operating system (such as Ubuntu 10.04), and if you have done any local modifications to source code.

Questions and answers

Launchpad provides interface for giving questions at https://answers.launchpad.net/yade/ which you can use instead of mailing lists; at the moment, it functionality somewhat overlaps with yade-users, but has the advantage of tracking whether a particular question has already been answered.

Wiki

http://www.yade-dem.org/wiki/

Private and/or paid support

You might contact developers by their private mail (rather than by mailing list) if you do not want to disclose details on the mailing list. This is also a suitable method for proposing financial reward for implementation of a substantial feature that is not yet in Yade – typically, though, we will request this feature to be part of the public codebase once completed, so that the rest of the community can benefit from it as well.

Chapter 4

Programmer's manual

4.1 Build system

Yade uses cmake the cross-platform, open-source build system for managing the build process. It takes care of configuration, compilation and installation. CMake is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice.

4.1.1 Building

Yade source tree has the following structure (omiting, `doc`, `examples` and `scripts` which don't participate in the build process); we shall call each top-level component *module*:

```
core/          ## core simulation building blocks
extra/         ## miscillanea
gui/           ## user interfaces
   qt4/           ## graphical user interface based on qt3 and OpenGL
   py/            ## python console interface (phased out)
lib/           ## support libraries, not specific to simulations
pkg/           ## simulation-specific files
   common/        ## generally useful classes
   dem/           ## classes for Discrete Element Method
py/            ## python modules
```

Header installation

To allow flexibility in source layout, CMAKE will copy (symlink) all headers into flattened structure within the build directory. First 2 components of the original directory are joind by dash, deeper levels are discarded (in case of `core` and `extra`, only 1 level is used). The following table makes gives a few examples:

Original header location	Included as
`core/Scene.hpp`	`<core/Scene.hpp>`
`lib/base/Logging.hpp`	`<lib-base/Logging.hpp>`
`lib/serialization/Serializable.hpp`	`<lib-serialization/Serializable.hpp>`
`pkg/dem/DataClass/SpherePack.hpp`	`<pkg-dem/SpherePack.hpp>`
`gui/qt3/QtGUI.hpp`	`<gui-qt3/QtGUI.hpp>`

It is advised to use `#include<module/Class.hpp>` style of inclusion rather than `#include"Class.hpp` even if you are in the same directory.

Automatic compilation

In the `pkg/` directory, situation is different. In order to maximally ease addition of modules to yade, all `*.cpp` files are *automatically scanned* by CMAKE and considered for compilation. Each file may contain multiple lines that declare features that are necessary for this file to be compiled:

```
YADE_REQUIRE_FEATURE(vtk);
YADE_REQUIRE_FEATURE(gts);
```

This file will be compiled only if *both* `VTK` and `GTS` features are enabled. Depending on current feature set, only selection of plugins will be compiled.

It is possible to disable compilation of a file by requiring any non-existent feature, such as:

```
YADE_REQUIRE_FEATURE(temporarily disabled 345uiysdijkn);
```

The `YADE_REQUIRE_FEATURE` macro expands to nothing during actual compilation.

Linking

The order in which modules might depend on each other is given as follows:

module	resulting shared library	dependencies
lib	`libyade-support.so`	can depend on external libraries, may **not** depend on any other part of Yade.
core	`libcore.so`	`yade-support`; *may* depend on external libraries.
pkg	`libplugins.so`	`core`, `yade-support`
gui	`libQtGUI.so`, `libPythonUI.so`	`lib`, `core`, `pkg`
py	(many files)	`lib`, `core`, `pkg`, external

4.2 Development tools

4.2.1 Integrated Development Environment and other tools

A frequently used IDE is Kdevelop. We recommend using this software for navigating in the sources, compiling and debugging. Other useful tools for debugging and profiling are Valgrind and KCachegrind. A series of wiki pages is dedicated to these tools in the development section of the wiki.

4.2.2 Hosting and versioning

The Yade project is kindly hosted at launchpad, which is used for source code, bug tracking, planning, package downloads and more.

The versioning software used is GIT, for which a short tutorial can be found in *Yade on GitHub*. GIT is a distributed revision control system. It is available packaged for all major linux distributions.

The source code is hosted on GitHub , which is periodically imported to Launchpad for building PPA-packages. The repository can be http-browsed.

4.2.3 Build robot

A build robot hosted at 3SR lab. is tracking souce code changes. Each time a change in the source code is commited to the main development branch via GIT, the "buildbot" downloads and compiles the new version, and start a series of tests.

If a compilation error has been introduced, it will be notified to the yade-dev mailing list and to the commiter, thus helping to fix problems quickly. If the compilation is successfull, the buildbot starts unit regression tests and "check tests" (see below) and report the results. If all tests are passed, a new version of the documentation is generated and uploaded to the website in html and pdf formats. As a consequence, those two links always point to the documentation (the one you are reading now) of the last successfull build, and the delay between commits and documentation updates are very short (minutes). The buildbot activity and logs can be browsed online.

4.2.4 Regression tests

Yade contains two types of regression tests, some are unit tests while others are testing more complex simulations. Altough both types can be considered regression tests, the usage is that we name the first simply "regression tests", while the latest are called "check tests". Both series of tests can be ran at yade startup by passing the options "test" or "check"

```
yade --test
yade --check
```

Unit regression tests

Unit regression tests are testing the output of individual functors and engines in well defined conditions. They are defined in the folder py/tests/. The purpose of unit testing is to make sure that the behaviour of the most important classes remains correct during code development. Since they test classes one by one, unit tests can't detect problems coming from the interaction between different engines in a typical simulation. That is why check tests have been introduced.

Check tests

Check tests perform comparisons of simulation results between different versions of yade, as discussed here. They differ with regression tests in the sense that they simulate more complex situations and combinations of different engines, and usually don't have a mathematical proof (though there is no restriction on the latest). They compare the values obtained in version N with values obtained in a previous version or any other "expected" results. The reference values must be hardcoded in the script itself or in data files provided with the script. Check tests are based on regular yade scripts, so that users can easily commit their own scripts to trunk in order to get some automatized testing after commits from other developers.

Since the check tests history will be mostly based on standard output generated by "yade —check", a meaningfull checkTest should include some "print" command telling if something went wrong. If the script itself fails for some reason and can't generate an output, the log will contain "scriptName failure". If the script defines differences on obtained and awaited data, it should print some useful information about the problem and increase the value of global variable resultStatus. After this occurs, the automatic test will stop the execution with error message.

An example check test can be found in checkTestTriax.py. It shows results comparison, output, and how to define the path to data files using "checksPath". Users are encouraged to add their own scripts into the scripts/test/checks/ folder. Discussion of some specific checktests design in users question is welcome. Note that re-compiling is required before that added scripts can be launched by "yade —check" (or direct changes have to be performed in "lib" subfolders). A check test should never need more than a few seconds to run. If your typical script needs more, try and reduce the number of element or the number of steps.

4.3 Conventions

The following rules that should be respected; documentation is treated separately.

- general

  - C++ source files have `.hpp` and `.cpp` extensions (for headers and implementation, respectively).
  - All header files should have the `#pragma once` multiple-inclusion guard.
  - Try to avoid `using namespace …` in header files.
  - Use tabs for indentation. While this is merely visual in `c++`, it has semantic meaning in python; inadverently mixing tabs and spaces can result in syntax errors.
- capitalization style
  - Types should be always capitalized. Use CamelCase for composed names (`GlobalEngine`). Underscores should be used only in special cases, such as functor names.
  - Class data members and methods must not be capitalized, composed names should use use lowercased camelCase (`glutSlices`). The same applies for functions in python modules.
  - Preprocessor macros are uppercase, separated by underscores; those that are used outside the core take (with exceptions) the form `YADE_*`, such as *YADE_CLASS_BASE_DOC_* macro family*.
- programming style
  - Be defensive, if it has no significant performance impact. Use assertions abundantly: they don't affect performance (in the optimized build) and make spotting error conditions much easier.
  - Use `YADE_CAST` and `YADE_PTR_CAST` where you want type-check during debug builds, but fast casting in optimized build.
  - Initialize all class variables in the default constructor. This avoids bugs that may manifest randomly and are difficult to fix. Initializing with NaN's will help you find otherwise unitialized variable. (This is taken care of by *YADE_CLASS_BASE_DOC_* macro family* macros for user classes)

4.3.1 Class naming

Although for historial reasons the naming scheme is not completely consistent, these rules should be obeyed especially when adding a new class.

GlobalEngines and PartialEngines GlobalEngines should be named in a way suggesting that it is a performer of certain action (like *ForceResetter*, *InsertionSortCollider*, *Recorder*); if this is not appropriate, append the `Engine` to the characteristics (*GravityEngine*). *PartialEngines* have no special naming convention different from *GlobalEngines*.

Dispatchers Names of all dispatchers end in `Dispatcher`. The name is composed of type it creates or, in case it doesn't create any objects, its main characteristics. Currently, the following dispatchers [1] are defined:

dispatcher	arity	dispatch types	created type	functor type	functor prefix
BoundDispatcher	1	*Shape*	*Bound*	*BoundFunctor*	`Bo1`
IGeomDispatcher	2 (symetric)	2 × *Shape*	*IGeom*	*IGeomFunctor*	`Ig2`
IPhysDispatcher	2 (symetric)	2 × *Material*	*IPhys*	*IPhysFunctor*	`Ip2`
LawDispatcher	2 (asymetric)	*IGeom* *IPhys*	*(none)*	*LawFunctor*	`Law2`

Respective abstract functors for each dispatchers are *BoundFunctor*, *IGeomFunctor*, *IPhysFunctor* and *LawFunctor*.

Functors Functor name is composed of 3 parts, separated by underscore.

[1] Not considering OpenGL dispatchers, which might be replaced by regular virtual functions in the future.

1. prefix, composed of abbreviated functor type and arity (see table above)
2. Types entering the dispatcher logic (1 for unary and 2 for binary functors)
3. Return type for functors that create instances, simple characteristics for functors that don't create instances.

To give a few examples:

- *Bo1_Sphere_Aabb* is a *BoundFunctor* which is called for *Sphere*, creating an instance of *Aabb*.
- *Ig2_Facet_Sphere_ScGeom* is binary functor called for *Facet* and *Sphere*, creating and instace of *ScGeom*.
- *Law2_ScGeom_CpmPhys_Cpm* is binary functor (*LawFunctor*) called for types *ScGeom (Geom)* and *CpmPhys*.

4.3.2 Documentation

Documenting code properly is one of the most important aspects of sustained development.

Read it again.

Most code in research software like Yade is not only used, but also read, by developers or even by regular users. Therefore, when adding new class, always mention the following in the documentation:

- purpose
- details of the functionality, unless obvious (algorithms, internal logic)
- limitations (by design, by implementation), bugs
- bibliographical reference, if using non-trivial published algorithms (see below)
- references to other related classes
- hyperlinks to bugs, blueprints, wiki or mailing list about this particular feature.

As much as it is meaningful, you should also

- update any other documentation affected
- provide a simple python script demonstrating the new functionality in `scripts/test`.

Sphinx documentation

Most c++ classes are wrapped in Python, which provides good introspection and interactive documentation (try writing `Material?` in the ipython prompt; or `help(CpmState)`).

Syntax of documentation is ReST (reStructuredText, see reStructuredText Primer). It is the same for c++ and python code.

- Documentation of c++ classes exposed to python is given as 3rd argument to *YADE_CLASS_BASE_DOC_* macro family* introduced below.
- Python classes/functions are documented using regular python docstrings. Besides explaining functionality, meaning and types of all arguments should also be documented. Short pieces of code might be very helpful. See the *utils* module for an example.

In addition to standard ReST syntax, yade provides several shorthand macros:

`:yref:` creates hyperlink to referenced term, for instance:

```
:yref:`CpmMat`
```

becomes *CpmMat*; link name and target can be different:

```
:yref:`Material used in the CPM model<CpmMat>`
```

yielding *Material used in the CPM model*.

`:ysrc:` creates hyperlink to file within the source tree (to its latest version in the repository), for instance core/Cell.hpp. Just like with *:yref:*, alternate text can be used with

```
:ysrc:`Link text<target/file>`
```

like this.

`|ycomp|` is used in attribute description for those that should not be provided by the used, but are auto-computed instead; `|ycomp|` expands to *(auto-computed)*.

`|yupdate|` marks attributes that are periodically update, being subset of the previous. `|yupdate|` expands to *(auto-updated)*.

`$...$` delimits inline math expressions; they will be replaced by:

```
:math:`...`
```

and rendered via LaTeX. To write a single dollar sign, escape it with backslash `\$`.

Displayed mathematics (standalone equations) can be inserted as explained in Math support in Sphinx.

Bibliographical references

As in any scientific documentation, references to publications are very important. To cite an article, add it to BibTeX file in doc/references.bib, using the BibTeX format. Please adhere to the following conventions:

1. Keep entries in the form `Author2008` (`Author` is the first author), `Author2008b` etc if multiple articles from one author;
2. Try to fill mandatory fields for given type of citation;
3. Do not use `\'{i}` funny escapes for accents, since they will not work with the HTML output; put everything in straight utf-8.

In your docstring, the `Author2008` article can be cited by `[Author2008]_`; for example:

```
According to [Allen1989]_, the integration scheme …
```

will be rendered as

According to *[Allen1989]*, the integration scheme …

Separate class/function documentation

Some c++ might have long or content-rich documentation, which is rather inconvenient to type in the c++ source itself as string literals. Yade provides a way to write documentation separately in py/_extraDocs.py file: it is executed after loading c++ plugins and can set `__doc__` attribute of any object directly, overwriting docstring from c++. In such (exceptional) cases:

1. Provide at least a brief description of the class in the c++ code nevertheless, for people only reading the code.
2. Add notice saying "This class is documented in detail in the py/_extraDocs.py file".
3. Add documentation to py/_etraDocs.py in this way:

```
module.YourClass.__doc__='''
        This is the docstring for YourClass.

        Class, methods and functions can be documented this way.

        .. note:: It can use any syntax features you like.

'''
```

Note: Boost::python embeds function signatures in the docstring (before the one provided by the user). Therefore, before creating separate documentation of your function, have a look at its `__doc__` attribute and copy the first line (and the blank lie afterwards) in the separate docstring. The first line is then used to create the function signature (arguments and return value).

Internal c++ documentation

doxygen was used for automatic generation of c++ code. Since user-visible classes are defined with sphinx now, it is not meaningful to use doxygen to generate overall documentation. However, take care to document well internal parts of code using regular comments, including public and private data members.

4.4 Support framework

Besides the framework provided by the c++ standard library (including STL), boost and other dependencies, yade provides its own specific services.

4.4.1 Pointers

Shared pointers

Yade makes extensive use of shared pointers `shared_ptr`. [2] Although it probably has some performance impacts, it greatly simplifies memory management, ownership management of c++ objects in python and so forth. To obtain raw pointer from a `shared_ptr`, use its `get()` method; raw pointers should be used in case the object will be used only for short time (during a function call, for instance) and not stored anywhere.

Python defines thin wrappers for most c++ Yade classes (for all those registered with *YADE_CLASS_BASE_DOC_* macro family* and several others), which can be constructed from `shared_ptr`; in this way, Python reference counting blends with the `shared_ptr` reference counting model, preventing crashes due to python objects pointing to c++ objects that were destructed in the meantime.

Typecasting

Frequently, pointers have to be typecast; there is choice between static and dynamic casting.

- `dynamic_cast` (`dynamic_pointer_cast` for a `shared_ptr`) assures cast admissibility by checking runtime type of its argument and returns NULL if the cast is invalid; such check obviously costs time. Invalid cast is easily caught by checking whether the pointer is NULL or not; even if such check (e.g. `assert`) is absent, dereferencing NULL pointer is easily spotted from the stacktrace (debugger output) after crash. Moreover, `shared_ptr` checks that the pointer is non-NULL before dereferencing in debug build and aborts with "Assertion 'px!=0' failed." if the check fails.
- `static_cast` is fast but potentially dangerous (`static_pointer_cast` for `shared_ptr`). Static cast will return non-NULL pointer even if types don't allow the cast (such as casting from `State*` to `Material*`); the consequence of such cast is interpreting garbage data as instance of the class cast to, leading very likely to invalid memory access (segmentation fault, "crash" for short).

To have both speed and safety, Yade provides 2 macros:

`YADE_CAST` expands to `static_cast` in optimized builds and to `dynamic_cast` in debug builds.

`YADE_PTR_CAST` expands to `static_pointer_cast` in optimized builds and to `dynamic_pointer_cast` in debug builds.

[2] Either `boost::shared_ptr` or `tr1::shared_ptr` is used, but it is always imported with the `using` statement so that unqualified `shared_ptr` can be used.

4.4.2 Basic numerics

The floating point type to use in Yade `Real`, which is by default typedef for `double`. [3]

Yade uses the Eigen library for computations. It provides classes for 2d and 3d vectors, quaternions and 3x3 matrices templated by number type; their specialization for the `Real` type are typedef'ed with the "r" suffix, and occasionally useful integer types with the "i" suffix:

- `Vector2r`, `Vector2i`
- `Vector3r`, `Vector3i`
- `Quaternionr`
- `Matrix3r`

Yade additionally defines a class named *Se3r*, which contains spatial position (`Vector3r Se3r::position`) and orientation (`Quaternionr Se3r::orientation`), since they are frequently used one with another, and it is convenient to pass them as single parameter to functions.

Eigen provides full rich linear algebra functionality. Some code firther uses the *[cgal]* library for computational geometry.

In Python, basic numeric types are wrapped and imported from the `minieigen` module; the types drop the `r` type qualifier at the end, the syntax is otherwise similar. `Se3r` is not wrapped at all, only converted automatically, rarely as it is needed, from/to a `(Vector3,Quaternion)` tuple/list.

```
# cross product
Yade [116]: Vector3(1,2,3).cross(Vector3(0,0,1))
Out[116]: Vector3(2,-1,0)

# construct quaternion from axis and angle
Yade [117]: Quaternion(Vector3(0,0,1),pi/2)
Out[117]: Quaternion((0,0,1),1.5707963267948966)
```

Note: Quaternions are internally stored as 4 numbers. Their usual human-readable representation is, however, (normalized) axis and angle of rotation around that axis, and it is also how they are input/output in Python. Raw internal values can be accessed using the `[0]` … `[3]` element access (or `.W()`, `.X()`, `.Y()` and `.Z()` methods), in both c++ and Python.

4.4.3 Run-time type identification (RTTI)

Since serialization and dispatchers need extended type and inheritance information, which is not sufficiently provided by standard RTTI. Each yade class is therefore derived from `Factorable` and it must use macro to override its virtual functions providing this extended RTTI:

`YADE_CLASS_BASE_DOC(Foo,Bar Baz,"Docstring)` creates the following virtual methods (mediated via the `REGISTER_CLASS_AND_BASE` macro, which is not user-visible and should not be used directly):

- `std::string getClassName()` returning class name (`Foo`) as string. (There is the `typeid(instanceOrType).name()` standard c++ construct, but the name returned is compiler-dependent.)
- `unsigned getBaseClassNumber()` returning number of base classes (in this case, 2).
- `std::string getBaseClassName(unsigned i=0)` returning name of i-th base class (here, `Bar` for i=0 and `Baz` for i=1).

[3] Historically, it was thought that Yade could be also run with single precision based on build-time parameter; it turned out however that the impact on numerical stability was such disastrous that this option is not available now. There is, however, `QUAD_PRECISION` parameter to scons, which will make `Real` a typedef for `long double` (extended precision; quad precision in the proper sense on IA64 processors); this option is experimental and is unlikely to be used in near future, though.

Warning: RTTI relies on virtual functions; in order for virtual functions to work, at least one virtual method must be present in the implementation (`.cpp`) file. Otherwise, virtual method table (vtable) will not be generated for this class by the compiler, preventing virtual methods from functioning properly.

Some RTTI information can be accessed from python:

```
Yade [118]: yade.system.childClasses('Shape')
Out[118]:
{'Box',
 'ChainedCylinder',
 'Clump',
 'Cylinder',
 'Facet',
 'GridConnection',
 'GridNode',
 'Polyhedra',
 'Sphere',
 'Tetra',
 'Wall'}

Yade [119]: Sphere().name            ## getClassName()
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/build/yade-Swrxdd/yade-1.20.0/debian/tmp/usr/lib/x86_64-linux-gnu/yade/py/yade/__init__.pyc in <module>()
----> 1 Sphere().name            ## getClassName()

AttributeError: 'Sphere' object has no attribute 'name'
```

4.4.4 Serialization

Serialization serves to save simulation to file and restore it later. This process has several necessary conditions:

- classes know which attributes (data members) they have and what are their names (as strings);
- creating class instances based solely on its name;
- knowing what classes are defined inside a particular shared library (plugin).

This functionality is provided by 3 macros and 4 optional methods; details are provided below.

`Serializable::preLoad`, `Serializable::preSave`, `Serializable::postLoad`, `Serializable::postSave`
: Prepare attributes before serialization (saving) or deserialization (loading) or process them after serialization or deserialization.

  See *Attribute registration*.

`YADE_CLASS_BASE_DOC_*` Inside the class declaration (i.e. in the `.hpp` file within the `class Foo { /* … */};` block). See *Attribute registration*.

: Enumerate class attributes that should be saved and loaded; associate each attribute with its literal name, which can be used to retrieve it. See *YADE_CLASS_BASE_DOC_* macro family*.

  Additionally documents the class in python, adds methods for attribute access from python, and documents each attribute.

`REGISTER_SERIALIZABLE` In header file, but *after* the class declaration block. See *Class factory*.

: Associate literal name of the class with functions that will create its new instance (`ClassFactory`).

`YADE_PLUGIN` In the implementation `.cpp` file. See *Plugin registration*.

: Declare what classes are declared inside a particular plugin at time the plugin is being loaded (yade startup).

Attribute registration

All (serializable) types in Yade are one of the following:

- Type deriving from *Serializable*, which provide information on how to serialize themselves via overriding the `Serializable::registerAttributes` method; it declares data members that should be serialzed along with their literal names, by which they are identified. This method then invokes `registerAttributes` of its base class (until `Serializable` itself is reached); in this way, derived classes properly serialize data of their base classes.

  This funcionality is hidden behind the macro *YADE_CLASS_BASE_DOC_* * *macro family* used in class declaration body (header file), which takes base class and list of attributes:

```
YADE_CLASS_BASE_DOC_ATTRS(ThisClass,BaseClass,"class documentation",((type1,attribute1,initValue1,,"Docume
```

  Note that attributes are encodes in double parentheses, not separated by commas. Empty attribute list can be given simply by `YADE_CLASS_BASE_DOC_ATTRS(ThisClass,BaseClass,"documentation",)` (the last comma is mandatory), or by omiting `ATTRS` from macro name and last parameter altogether.

- Fundamental type: strings, various number types, booleans, `Vector3r` and others. Their "handlers" (serializers and deserializers) are defined in `lib/serialization`.
- Standard container of any serializable objects.
- Shared pointer to serializable object.

Yade uses the excellent boost::serialization library internally for serialization of data.

Note: `YADE_CLASS_BASE_DOC_ATTRS` also generates code for attribute access from python; this will be discussed later. Since this macro serves both purposes, the consequence is that attributes that are serialized can always be accessed from python.

Yade also provides callback for before/after (de) serialization, virtual functions *Serializable::preProcessAttributes* and *Serializable::postProcessAttributes*, which receive one `bool deserializing` argument (`true` when deserializing, `false` when serializing). Their default implementation in *Serializable* doesn't do anything, but their typical use is:

- converting some non-serializable internal data structure of the class (such as multi-dimensional array, hash table, array of pointers) into a serializable one (pre-processing) and fill this non-serializable structure back after deserialization (post-processing); for instance, InteractionContainer uses these hooks to ask its concrete implementation to store its contents to a unified storage (`vector<shared_ptr<Interaction> >`) before serialization and to restore from it after deserialization.
- precomputing non-serialized attributes from the serialized values; e.g. *Facet* computes its (local) edge normals and edge lengths from vertices' coordinates.

Class factory

Each serializable class must use `REGISTER_SERIALIZABLE`, which defines function to create that class by `ClassFactory`. `ClassFactory` is able to instantiate a class given its name (as string), which is necessary for deserialization.

Although mostly used internally by the serialization framework, programmer can ask for a class instantiation using `shared_ptr<Factorable> f=ClassFactory::instance().createShared("ClassName");`, casting the returned `shared_ptr<Factorable>` to desired type afterwards. *Serializable* itself derives from `Factorable`, i.e. all serializable types are also factorable (It is possible that different mechanism will be in place if boost::serialization is used, though.)

Plugin registration

Yade loads dynamic libraries containing all its functionality at startup. ClassFactory must be taught about classes each particular file provides. `YADE_PLUGIN` serves this purpose and, contrary to *YADE_CLASS_BASE_DOC_* macro family*, must be place in the implementation (.cpp) file. It simple enumerates classes that are provided by this file:

```
YADE_PLUGIN((ClassFoo)(ClassBar));
```

Note: You must use parentheses around the class name even if there is only one (preprocessor limitation): `YADE_PLUGIN((classFoo));`. If there is no class in this file, do not use this macro at all.

Internally, this macro creates function `registerThisPluginClasses_` declared specially as `__attribute__((constructor))` (see GCC Function Attributes); this attributes makes the function being executed when the plugin is loaded via `dlopen` from `ClassFactory::load(...)`. It registers all factorable classes from that file in the *Class factory*.

Note: Classes that do not derive from `Factorable`, such as `Shop` or `SpherePack`, are not declared with `YADE_PLUGIN`.

This is an example of a serializable class header:

```
/*! Homogeneous gravity field; applies gravity×mass force on all bodies. */
class GravityEngine: public GlobalEngine{
        public:
                virtual void action();
        // registering class and its base for the RTTI system
        YADE_CLASS_BASE_DOC_ATTRS(GravityEngine,GlobalEngine,
                // documentation visible from python and generated reference documentation
                "Homogeneous gravity field; applies gravity×mass force on all bodies.",
                // enumerating attributes here, include documentation
                ((Vector3r,gravity,Vector3r::ZERO,"acceleration, zero by default [kgms ²]"))
        );
};
// registration function for ClassFactory
REGISTER_SERIALIZABLE(GravityEngine);
```

and this is the implementation:

```
#include<pkg-common/GravityEngine.hpp>
#include<core/Scene.hpp>

// registering the plugin
YADE_PLUGIN((GravityEngine));

void GravityEngine::action(){
        /* do the work here */
}
```

We can create a mini-simulation (with only one GravityEngine):

```
Yade [120]: O.engines=[GravityEngine(gravity=Vector3(0,0,-9.81))]

Yade [121]: O.save('abc.xml')
```

and the XML looks like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="12">
<scene class_id="0" tracking_level="0" version="1">
```

```
<px class_id="1" tracking_level="1" version="0" object_id="_0">
        <Serializable class_id="2" tracking_level="1" version="0" object_id="_1"></Serializable>
        <dt>1.0000000000000002e-08</dt>
        <iter>0</iter>
        <subStepping>0</subStepping>
        <subStep>-1</subStep>
        <time>0.0000000000000000e+00</time>
        <speed>0.0000000000000000e+00</speed>
        <stopAtIter>0</stopAtIter>
        <stopAtTime>0.0000000000000000e+00</stopAtTime>
        <isPeriodic>0</isPeriodic>
        <trackEnergy>0</trackEnergy>
        <doSort>0</doSort>
        <runInternalConsistencyChecks>1</runInternalConsistencyChecks>
        <selectedBody>-1</selectedBody>
        <flags>0</flags>
        <tags class_id="3" tracking_level="0" version="0">
                <count>5</count>
                <item_version>0</item_version>
                <item>author=root~(root@x86-csail-02)</item>
                <item>isoTime=20151114T011325</item>
                <item>id=20151114T011325p2070</item>
                <item>d.id=20151114T011325p2070</item>
                <item>id.d=20151114T011325p2070</item>
        </tags>
        <engines class_id="4" tracking_level="0" version="0">
                <count>1</count>
                <item_version>1</item_version>
                <item class_id="5" tracking_level="0" version="1">
                        <px class_id="7" class_name="GravityEngine" tracking_level="1" version="0" obje
                                <FieldApplier class_id="8" tracking_level="1" version="0" object_id="_3
                                        <GlobalEngine class_id="9" tracking_level="1" version="0" objec
                                                <Engine class_id="6" tracking_level="1" version="0" obj
                                                        <Serializable object_id="_6"></Serializable>
                                                        <dead>0</dead>
                                                        <ompThreads>-1</ompThreads>
                                                        <label></label>
                                                </Engine>
                                        </GlobalEngine>
                                </FieldApplier>
                                <gravity class_id="10" tracking_level="0" version="0">
                                        <x>0.0000000000000000e+00</x>
                                        <y>0.0000000000000000e+00</y>
                                        <z>-9.8100000000000050e+00</z>
                                </gravity>
                                <mask>0</mask>
                                <warnOnce>1</warnOnce>
                        </px>
                </item>
        </engines>
        <_nextEngines>
                <count>0</count>
                <item_version>1</item_version>
        </_nextEngines>
        <bodies class_id="11" tracking_level="0" version="1">
                <px class_id="12" tracking_level="1" version="0" object_id="_7">
                        <Serializable object_id="_8"></Serializable>
                        <body class_id="13" tracking_level="0" version="0">
                                <count>0</count>
                                <item_version>1</item_version>
                        </body>
                </px>
        </bodies>
```

```
<interactions class_id="14" tracking_level="0" version="1">
        <px class_id="15" tracking_level="1" version="0" object_id="_9">
                <Serializable object_id="_10"></Serializable>
                <interaction class_id="16" tracking_level="0" version="0">
                        <count>0</count>
                        <item_version>1</item_version>
                </interaction>
                <serializeSorted>0</serializeSorted>
                <dirty>1</dirty>
        </px>
</interactions>
<energy class_id="17" tracking_level="0" version="1">
        <px class_id="18" tracking_level="1" version="0" object_id="_11">
                <Serializable object_id="_12"></Serializable>
                <energies class_id="19" tracking_level="0" version="0">
                        <size>0</size>
                </energies>
                <names class_id="20" tracking_level="0" version="0">
                        <count>0</count>
                        <item_version>0</item_version>
                </names>
                <resetStep>
                        <count>0</count>
                </resetStep>
        </px>
</energy>
<materials class_id="22" tracking_level="0" version="0">
        <count>0</count>
        <item_version>1</item_version>
</materials>
<bound class_id="23" tracking_level="0" version="1">
        <px class_id="-1"></px>
</bound>
<cell class_id="25" tracking_level="0" version="1">
        <px class_id="26" tracking_level="1" version="0" object_id="_13">
                <Serializable object_id="_14"></Serializable>
                <trsf class_id="27" tracking_level="0" version="0">
                        <m00>1.0000000000000000e+00</m00>
                        <m01>0.0000000000000000e+00</m01>
                        <m02>0.0000000000000000e+00</m02>
                        <m10>0.0000000000000000e+00</m10>
                        <m11>1.0000000000000000e+00</m11>
                        <m12>0.0000000000000000e+00</m12>
                        <m20>0.0000000000000000e+00</m20>
                        <m21>0.0000000000000000e+00</m21>
                        <m22>1.0000000000000000e+00</m22>
                </trsf>
                <refHSize>
                        <m00>1.0000000000000000e+00</m00>
                        <m01>0.0000000000000000e+00</m01>
                        <m02>0.0000000000000000e+00</m02>
                        <m10>0.0000000000000000e+00</m10>
                        <m11>1.0000000000000000e+00</m11>
                        <m12>0.0000000000000000e+00</m12>
                        <m20>0.0000000000000000e+00</m20>
                        <m21>0.0000000000000000e+00</m21>
                        <m22>1.0000000000000000e+00</m22>
                </refHSize>
                <hSize>
                        <m00>1.0000000000000000e+00</m00>
                        <m01>0.0000000000000000e+00</m01>
                        <m02>0.0000000000000000e+00</m02>
                        <m10>0.0000000000000000e+00</m10>
```

```
                                        <m11>1.0000000000000000e+00</m11>
                                        <m12>0.0000000000000000e+00</m12>
                                        <m20>0.0000000000000000e+00</m20>
                                        <m21>0.0000000000000000e+00</m21>
                                        <m22>1.0000000000000000e+00</m22>
                                </hSize>
                                <prevHSize>
                                        <m00>1.0000000000000000e+00</m00>
                                        <m01>0.0000000000000000e+00</m01>
                                        <m02>0.0000000000000000e+00</m02>
                                        <m10>0.0000000000000000e+00</m10>
                                        <m11>1.0000000000000000e+00</m11>
                                        <m12>0.0000000000000000e+00</m12>
                                        <m20>0.0000000000000000e+00</m20>
                                        <m21>0.0000000000000000e+00</m21>
                                        <m22>1.0000000000000000e+00</m22>
                                </prevHSize>
                                <velGrad>
                                        <m00>0.0000000000000000e+00</m00>
                                        <m01>0.0000000000000000e+00</m01>
                                        <m02>0.0000000000000000e+00</m02>
                                        <m10>0.0000000000000000e+00</m10>
                                        <m11>0.0000000000000000e+00</m11>
                                        <m12>0.0000000000000000e+00</m12>
                                        <m20>0.0000000000000000e+00</m20>
                                        <m21>0.0000000000000000e+00</m21>
                                        <m22>0.0000000000000000e+00</m22>
                                </velGrad>
                                <nextVelGrad>
                                        <m00>0.0000000000000000e+00</m00>
                                        <m01>0.0000000000000000e+00</m01>
                                        <m02>0.0000000000000000e+00</m02>
                                        <m10>0.0000000000000000e+00</m10>
                                        <m11>0.0000000000000000e+00</m11>
                                        <m12>0.0000000000000000e+00</m12>
                                        <m20>0.0000000000000000e+00</m20>
                                        <m21>0.0000000000000000e+00</m21>
                                        <m22>0.0000000000000000e+00</m22>
                                </nextVelGrad>
                                <prevVelGrad>
                                        <m00>0.0000000000000000e+00</m00>
                                        <m01>0.0000000000000000e+00</m01>
                                        <m02>0.0000000000000000e+00</m02>
                                        <m10>0.0000000000000000e+00</m10>
                                        <m11>0.0000000000000000e+00</m11>
                                        <m12>0.0000000000000000e+00</m12>
                                        <m20>0.0000000000000000e+00</m20>
                                        <m21>0.0000000000000000e+00</m21>
                                        <m22>0.0000000000000000e+00</m22>
                                </prevVelGrad>
                                <homoDeform>2</homoDeform>
                                <velGradChanged>0</velGradChanged>
                        </px>
                </cell>
                <miscParams class_id="28" tracking_level="0" version="0">
                        <count>0</count>
                        <item_version>1</item_version>
                </miscParams>
                <dispParams class_id="29" tracking_level="0" version="0">
                        <count>0</count>
                        <item_version>1</item_version>
                </dispParams>
        </px>
```

```
</scene>
</boost_serialization>
```

Warning: Since XML files closely reflect implementation details of Yade, they will not be compatible between different versions. Use them only for short-term saving of scenes. Python is *the* high-level description Yade uses.

Python attribute access

The macro *YADE_CLASS_BASE_DOC_* macro family* introduced above is (behind the scenes) also used to create functions for accessing attributes from Python. As already noted, set of serialized attributes and set of attributes accessible from Python are identical. Besides attribute access, these wrapper classes imitate also some functionality of regular python dictionaries:

```
Yade [122]: s=Sphere()

Yade [123]: s.radius              ## read-access
Out[123]: nan

Yade [124]: s.radius=4.           ## write access

Yade [125]: s.dict().keys()              ## show all available keys
Out[125]: ['color', 'highlight', 'wire', 'radius']

Yade [126]: for k in s.dict().keys(): print s.dict()[k]  ## iterate over keys, print their values
   .....:
Vector3(1,1,1)
False
False
4.0

Yade [127]: s.dict()['radius']            ## same as: 'radius' in s.keys()
Out[127]: 4.0

Yade [128]: s.dict()                      ## show dictionary of both attributes and values
Out[128]: {'color': Vector3(1,1,1), 'highlight': False, 'radius': 4.0, 'wire': False}
```

4.4.5 YADE_CLASS_BASE_DOC_* macro family

There is several macros that hide behind them the functionality of *Sphinx documentation*, *Run-time type identification (RTTI)*, *Attribute registration*, *Python attribute access*, plus automatic attribute initialization and documentation. They are all defined as shorthands for base macro `YADE_CLASS_BASE_DOC_ATTRS_INIT_CTOR_PY` with some arguments left out. They must be placed in class declaration's body (`.hpp` file):

```
#define YADE_CLASS_BASE_DOC(klass,base,doc) \
        YADE_CLASS_BASE_DOC_ATTRS(klass,base,doc,)
#define YADE_CLASS_BASE_DOC_ATTRS(klass,base,doc,attrs) \
        YADE_CLASS_BASE_DOC_ATTRS_CTOR(klass,base,doc,attrs,)
#define YADE_CLASS_BASE_DOC_ATTRS_CTOR(klass,base,doc,attrs,ctor) \
        YADE_CLASS_BASE_DOC_ATTRS_CTOR_PY(klass,base,doc,attrs,ctor,)
#define YADE_CLASS_BASE_DOC_ATTRS_CTOR_PY(klass,base,doc,attrs,ctor,py) \
        YADE_CLASS_BASE_DOC_ATTRS_INIT_CTOR_PY(klass,base,doc,attrs,,ctor,py)
#define YADE_CLASS_BASE_DOC_ATTRS_INIT_CTOR_PY(klass,base,doc,attrs,init,ctor,py) \
        YADE_CLASS_BASE_DOC_ATTRS_INIT_CTOR_PY(klass,base,doc,attrs,inits,ctor,py)
```

Expected parameters are indicated by macro name components separated with underscores. Their meaning is as follows:

klass (unquoted) name of this class (used for RTTI and python)

base (unquoted) name of the base class (used for RTTI and python)

doc docstring of this class, written in the ReST syntax. This docstring will appear in generated documentation (such as *CpmMat*). It can be as long as necessary, but sequences interpreted by c++ compiler must be properly escaped (therefore some backslashes must be doubled, like in $\sigma = \varepsilon E$:

```
":math:`\\sigma=\\epsilon E"
```

Use `\n` and `\t` for indentation inside the docstring. Hyperlink the documentation abundantly with `yref` (all references to other classes should be hyperlinks).

See *Sphinx documentation* for syntax details.

attrs Attribute must be written in the form of parethesized list:

```
((type1,attr1,initValue1,attrFlags,"Attribute 1 documentation"))
((type2,attr2,,,"Attribute 2 documentation"))  // initValue and attrFlags unspecified
```

This will expand to

1. data members declaration in c++ (note that all attributes are *public*):

```
public: type1 attr1;
        type2 attr2;
```

2. Initializers of the default (argument-less) constructor, for attributes that have non-empty `initValue`:

```
Klass(): attr1(initValue1), attr2() { /* constructor body */ }
```

No initial value will be assigned for attribute of which initial value is left empty (as is for attr2 in the above example). Note that you still have to write the commas.

3. Registration of the attribute in the serialization system (unless disabled by attrFlags – see below)
4. **Registration of the attribute in python (unless disabled by attrFlags), so that it can be accesse**
   The attribute is read-write by default, see attrFlags to change that.

   This attribute will carry the docstring provided, along with knowledge of the initial value. You can add text description to the default value using the comma operator of c++ and casting the char* to (void):

```
((Real,dmgTau,((void)"deactivated if negative",-1),,"Characteristic time for normal viscosity. [s]
```

   leading to *CpmMat::dmgTau*.

   The attribute is registered via `boost::python::add_property` specifying `return_by_value` policy rather than `return_internal_reference`, which is the default when using `def_readwrite`. The reason is that we need to honor custom converters for those values; see note in *Custom converters* for details.

Attribute flags

By default, an attribute will be serialized and will be read-write from python. There is a number of flags that can be passed as the 4th argument (empty by default) to change that:

- `Attr::noSave` avoids serialization of the attribute (while still keeping its accessibility from Python)
- `Attr::readonly` makes the attribute read-only from Python
- `Attr::triggerPostLoad` will trigger call to `postLoad` function to handle attribute change after its value is set from Python; this is to ensure consistency of other precomputed data which depend on this value (such as `Cell.trsf` and such)
- `Attr::hidden` will not expose the attribute to Python at all

- `Attr::noResize` will not permit changing size of the array from Python [not yet used]

Flags can be combined as usual using bitwise disjunction `|` (such as `Attr::noSave | Attr::readonly`), though in such case the value should be parenthesized to avoid a warning with some compilers (g++ specifically), i.e. `(Attr::noSave | Attr::readonly)`.

Currently, the flags logic handled at runtime; that means that even for attributes with `Attr::noSave`, their serialization template must be defined (although it will never be used). In the future, the implementation might be template-based, avoiding this necessity.

---

deprec List of deprecated attribute names. The syntax is

```
((oldName1,newName1,"Explanation why renamed etc."))
((oldName2,newName2,"! Explanation why removed and what to do instaed."))
```

This will make accessing `oldName1` attribute *from Python* return value of `newName`, but displaying warning message about the attribute name change, displaying provided explanation. This happens whether the access is read or write.

If the explanation's first character is `!` (*bang*), the message will be displayed upon attribute access, but exception will be thrown immediately. Use this in cases where attribute is no longer meaningful or was not straightforwardly replaced by another, but more complex adaptation of user's script is needed. You still have to give `newName2`, although its value will never be used – you can use any variable you like, but something must be given for syntax reasons).

> **Warning:** Due to compiler limitations, this feature only works if Yade is compiled with gcc >= 4.4. In the contrary case, deprecated attribute functionality is disabled, even if such attributes are declared.

init Parethesized list of the form:

```
((attr3,value3)) ((attr4,value4))
```

which will be expanded to initializers in the default ctor:

```
Klass(): /* attributes declared with the attrs argument */ attr4(value4), attr5(value5) { /* constructor b
```

The purpose of this argument is to make it possible to initialize constants and references (which are not declared as attributes using this macro themselves, but separately), as that cannot be done in constructor body. This argument is rarely used, though.

ctor will be put directly into the generated constructor's body. Mostly used for calling createIndex(); in the constructor.

Note: The code must not contain commas ouside parentheses (since preprocessor uses commas to separate macro arguments). If you need complex things at construction time, create a separate init() function and call it from the constructor instead.

---

py will be appeneded directly after generated python code that registers the class and all its attributes. You can use it to access class methods from python, for instance, to override an existing attribute with the same name etc:

```
.def_readonly("omega",&CpmPhys::omega,"Damage internal variable")
.def_readonly("Fn",&CpmPhys::Fn,"Magnitude of normal force.")
```

`def_readonly` will not work for custom types (such as std::vector), as it bypasses conversion registry; see *Custom converters* for details.

Special python constructors

The Python wrapper automatically create constructor that takes keyword (named) arguments corresponding to instance attributes; those attributes are set to values provided in the constructor. In some

cases, more flexibility is desired (such as *InteractionLoop*, which takes 3 lists of functors). For such cases, you can override the function `Serializable::pyHandleCustomCtorArgs`, which can arbitrarily modify the new (already existing) instance. It should modify in-place arguments given to it, as they will be passed further down to the routine which sets attribute values. In such cases, you should document the constructor:

```
.. admonition:: Special constructor

        Constructs from lists of …
```

which then appears in the documentation similar to *InteractionLoop*.

Static attributes

Some classes (such as OpenGL functors) are instantiated automatically; since we want their attributes to be persistent throughout the session, they are static. To expose class with static attributes, use the `YADE_CLASS_BASE_DOC_STATICATTRS` macro. Attribute syntax is the same as for `YADE_CLASS_BASE_DOC_ATTRS`:

```
class SomeClass: public BaseClass{
        YADE_CLASS_BASE_DOC_STATICATTRS(SomeClass,BaseClass,"Documentation of SomeClass",
                ((Type1,attr1,default1,"doc for attr1"))
                ((Type2,attr2,default2,"doc for attr2"))
        );
};
```

additionally, you *have* to allocate memory for static data members in the `.cpp` file (otherwise, error about undefined symbol will appear when the plugin is loaded):

There is no way to expose class that has both static and non-static attributes using `YADE_CLASS_BASE_*` macros. You have to expose non-static attributes normally and wrap static attributes separately in the `py` parameter.

Returning attribute by value or by reference

When attribute is passed from c++ to python, it can be passed either as

- value: new python object representing the original c++ object is constructed, but not bound to it; changing the python object doesn't modify the c++ object, unless explicitly assigned back to it, where inverse conversion takes place and the c++ object is replaced.
- reference: only reference to the underlying c++ object is given back to python; modifying python object will make the c++ object modified automatically.

The way of passing attributes given to `YADE_CLASS_BASE_DOC_ATTRS` in the `attrs` parameter is determined automatically in the following manner:

- **`Vector3`, `Vector3i`, `Vector2`, `Vector2i`, `Matrix3` and `Quaternion` objects are passed by *reference*. For in**
  O.bodies[0].state.pos[0]=1.33

  will assign correct value to `x` component of position, without changing the other ones.
- **Yade classes (all that use `shared_ptr` when declared in python: all classes deriving from *Serializable***
  O.engines[4].damping=.3

  will change *damping* parameter on the original engine object, not on its copy.
- **All other types are passed by *value*. This includes, most importantly, sequence types declared in *C***
  O.engines[4]=NewtonIntegrator()

  will *not* work as expected; it will replace 5th element of a *copy* of the sequence, and this change will not propagate back to c++.

4.4.6 Multiple dispatch

Multiple dispatch is generalization of virtual methods: a *Dispatcher* decides based on type(s) of its argument(s) which of its *Functors* to call. Numer of arguments (currently 1 or 2) determines *arity* of the dispatcher (and of the functor): unary or binary. For example:

```
InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb()])
```

creates *InsertionSortCollider*, which internally contains *Collider.boundDispatcher*, a *BoundDispatcher* (a *Dispatcher*), with 2 functors; they receive `Sphere` or `Facet` instances and create `Aabb`. This code would look like this in c++:

```
shared_ptr<InsertionSortCollider> collider=(new InsertionSortCollider);
collider->boundDispatcher->add(new Bo1_Sphere_Aabb());
collider->boundDispatcher->add(new Bo1_Facet_Aabb());
```

There are currenly 4 predefined dispatchers (see *dispatcher-names*) and corresponding functor types. They are inherit from template instantiations of `Dispatcher1D` or `Dispatcher2D` (for functors, `Functor1D` or `Functor2D`). These templates themselves derive from `DynlibDispatcher` (for dispatchers) and `FunctorWrapper` (for functors).

Example: IGeomDispatcher

Let's take (the most complicated perhaps) *IGeomDispatcher*. *IGeomFunctor*, which is dispatched based on types of 2 *Shape* instances (a *Functor*), takes a number of arguments and returns bool. The functor "call" is always provided by its overridden `Functor::go` method; it always receives the dispatched instances as first argument(s) (2 × `const shared_ptr<Shape>&`) and a number of other arguments it needs:

```
class IGeomFunctor: public Functor2D<
   bool,                                  //return type
   TYPELIST_7(const shared_ptr<Shape>&,   // 1st class for dispatch
      const shared_ptr<Shape>&,           // 2nd class for dispatch
      const State&,                       // other arguments passed to ::go
      const State&,                       // …
      const Vector3r&,                    // …
      const bool&,                        // …
      const shared_ptr<Interaction>&      // …
   )
>
```

The dispatcher is declared as follows:

```
class IGeomDispatcher: public Dispatcher2D<
   Shape,                        // 1st class for dispatch
   Shape,                        // 2nd class for dispatch
   IGeomFunctor,  // functor type
   bool,                         // return type of the functor

   // follow argument types for functor call
   // they must be exactly the same as types
   // given to the IGeomFunctor above.
   TYPELIST_7(const shared_ptr<Shape>&,
      const shared_ptr<Shape>&,
      const State&,
      const State&,
      const Vector3r&,
      const bool &,
      const shared_ptr<Interaction>&
   ),

   // handle symetry automatically
```

```
   // (if the dispatcher receives Sphere+Facet,
   // the dispatcher might call functor for Facet+Sphere,
   // reversing the arguments)
   false
>
{ /* … */ }
```

Functor derived from IGeomFunctor must then

- override the ::go method with appropriate arguments (they must match exactly types given to `TYPELIST_*` macro);
- declare what types they should be dispatched for, and in what order if they are not the same.

```
class Ig2_Facet_Sphere_ScGeom: public IGeomFunctor{
   public:

   // override the IGeomFunctor::go
   //   (it is really inherited from FunctorWrapper template,
   //    therefore not declare explicitly in the
   //    IGeomFunctor declaration as such)
   // since dispatcher dispatches only for declared types
   //   (or types derived from them), we can do
   //   static_cast<Facet>(shape1) and static_cast<Sphere>(shape2)
   //   in the ::go body, without worrying about types being wrong.
   virtual bool go(
      // objects for dispatch
      const shared_ptr<Shape>& shape1, const shared_ptr<Shape>& shape2,
      // other arguments
      const State& state1, const State& state2, const Vector3r& shift2,
      const bool& force, const shared_ptr<Interaction>& c
   );
   /* … */

   // this declares the type we want to be dispatched for, matching
   //   first 2 arguments to ::go and first 2 classes in TYPELIST_7 above
   //   shape1 is a Facet and shape2 is a Sphere
   //   (or vice versa, see lines below)
   FUNCTOR2D(Facet,Sphere);

   // declare how to swap the arguments
   //   so that we can receive those as well
   DEFINE_FUNCTOR_ORDER_2D(Facet,Sphere);
   /* … */
};
```

Dispatch resolution

The dispatcher doesn't always have functors that exactly match the actual types it receives. In the same way as virtual methods, it tries to find the closest match in such way that:

1. the actual instances are derived types of those the functor accepts, or exactly the accepted types;
2. sum of distances from actual to accepted types is sharp-minimized (each step up in the class hierarchy counts as 1)

If no functor is able to accept given types (first condition violated) or multiple functors have the same distance (in condition 2), an exception is thrown.

This resolution mechanism makes it possible, for instance, to have a hierarchy of *ScGeom* classes (for different combination of shapes), but only provide a *LawFunctor* accepting `ScGeom`, rather than having different laws for each shape combination.

Note: Performance implications of dispatch resolution are relatively low. The dispatcher lookup is only

done once, and uses fast lookup matrix (1D or 2D); then, the functor found for this type(s) is cached within the `Interaction` (or `Body`) instance. Thus, regular functor call costs the same as dereferencing pointer and calling virtual method. There is blueprint to avoid virtual function call as well.

Note: At the beginning, the dispatch matrix contains just entries exactly matching given functors. Only when necessary (by passing other types), appropriate entries are filled in as well.

Indexing dispatch types

Classes entering the dispatch mechanism must provide for fast identification of themselves and of their parent class. [4] This is called class indexing and all such classes derive from *Indexable*. There are `top-level` Indexables (types that the dispatchers accept) and each derived class registers its index related to this top-level Indexable. Currently, there are:

Top-level Indexable	used by
Shape	*BoundFunctor*, *IGeomDispatcher*
Material	*IPhysDispatcher*
IPhys	*LawDispatcher*
IGeom	*LawDispatcher*

The top-level Indexable must use the `REGISTER_INDEX_COUNTER` macro, which sets up the machinery for identifying types of derived classes; they must then use the `REGISTER_CLASS_INDEX` macro *and* call `createIndex()` in their constructor. For instance, taking the *Shape* class (which is a top-level Indexable):

```
// derive from Indexable
class Shape: public Serializable, public Indexable {
   // never call createIndex() in the top-level Indexable ctor!
   /* … */

   // allow index registration for classes deriving from ``Shape``
   REGISTER_INDEX_COUNTER(Shape);
};
```

Now, all derived classes (such as *Sphere* or *Facet*) use this:

```
class Sphere: public Shape{
   /* … */
   YADE_CLASS_BASE_DOC_ATTRS_CTOR(Sphere,Shape,"docstring",
      ((Type1,attr1,default1,"docstring1"))
      /* … */,
      // this is the CTOR argument
         // important; assigns index to the class at runtime
         createIndex();
   );
   // register index for this class, and give name of the immediate parent class
   //    (i.e. if there were a class deriving from Sphere, it would use
   //     REGISTER_CLASS_INDEX(SpecialSphere,Sphere),
   //     not REGISTER_CLASS_INDEX(SpecialSphere,Shape)!)
   REGISTER_CLASS_INDEX(Sphere,Shape);
};
```

At runtime, each class within the top-level Indexable hierarchy has its own unique numerical index. These indices serve to build the dispatch matrix for each dispatcher.

Inspecting dispatch in python

If there is a need to debug/study multiple dispatch, python provides convenient interface for this low-level functionality.

[4] The functionality described in *Run-time type identification (RTTI)* serves a different purpose (serialization) and would hurt the performance here. For this reason, classes provide numbers (indices) in addition to strings.

We can inspect indices with the `dispIndex` property (note that the top-level indexable `Shape` has negative (invalid) class index; we purposively didn't call `createIndex` in its constructor):

```
Yade [129]: Sphere().dispIndex, Facet().dispIndex, Wall().dispIndex
Out[129]: (1, 5, 10)

Yade [130]: Shape().dispIndex                    # top-level indexable
Out[130]: -1
```

Dispatch hierarchy for a particular class can be shown with the `dispHierarchy()` function, returning list of class names: 0th element is the instance itself, last element is the top-level indexable (again, with invalid index); for instance:

```
Yade [131]: ScGeom().dispHierarchy()        # parent class of all other ScGeom_ classes
Out[131]: ['ScGeom', 'GenericSpheresContact', 'IGeom']

Yade [132]: ScGridCoGeom().dispHierarchy(), ScGeom6D().dispHierarchy(), CylScGeom().dispHierarchy()
Out[132]:
(['ScGridCoGeom', 'ScGeom6D', 'ScGeom', 'GenericSpheresContact', 'IGeom'],
 ['ScGeom6D', 'ScGeom', 'GenericSpheresContact', 'IGeom'],
 ['CylScGeom', 'ScGeom', 'GenericSpheresContact', 'IGeom'])

Yade [133]: CylScGeom().dispHierarchy(names=False)    # show numeric indices instead
Out[133]: [4, 1, 0, -1]
```

Dispatchers can also be inspected, using the .dispMatrix() method:

```
Yade [134]: ig=IGeomDispatcher([
   .....:    Ig2_Sphere_Sphere_ScGeom(),
   .....:    Ig2_Facet_Sphere_ScGeom(),
   .....:    Ig2_Wall_Sphere_ScGeom()
   .....: ])
   .....:

Yade [135]: ig.dispMatrix()
Out[135]:
{('Facet', 'Sphere'): 'Ig2_Facet_Sphere_ScGeom',
 ('Sphere', 'Facet'): 'Ig2_Facet_Sphere_ScGeom',
 ('Sphere', 'Sphere'): 'Ig2_Sphere_Sphere_ScGeom',
 ('Sphere', 'Wall'): 'Ig2_Wall_Sphere_ScGeom',
 ('Wall', 'Sphere'): 'Ig2_Wall_Sphere_ScGeom'}

Yade [136]: ig.dispMatrix(False)          # don't convert to class names
Out[136]:
{(1, 1): 'Ig2_Sphere_Sphere_ScGeom',
 (1, 5): 'Ig2_Facet_Sphere_ScGeom',
 (1, 10): 'Ig2_Wall_Sphere_ScGeom',
 (5, 1): 'Ig2_Facet_Sphere_ScGeom',
 (10, 1): 'Ig2_Wall_Sphere_ScGeom'}
```

We can see that functors make use of symmetry (i.e. that Sphere+Wall are dispatched to the same functor as Wall+Sphere).

Finally, dispatcher can be asked to return functor suitable for given argument(s):

```
Yade [137]: ld=LawDispatcher([Law2_ScGeom_CpmPhys_Cpm()])

Yade [138]: ld.dispMatrix()
Out[138]: {('GenericSpheresContact', 'CpmPhys'): 'Law2_ScGeom_CpmPhys_Cpm'}

# see how the entry for ScGridCoGeom will be filled after this request
Yade [139]: ld.dispFunctor(ScGridCoGeom(),CpmPhys())
Out[139]: <Law2_ScGeom_CpmPhys_Cpm instance at 0x8597160>

Yade [140]: ld.dispMatrix()
```

```
Out[140]:
{('GenericSpheresContact', 'CpmPhys'): 'Law2_ScGeom_CpmPhys_Cpm',
 ('ScGridCoGeom', 'CpmPhys'): 'Law2_ScGeom_CpmPhys_Cpm'}
```

OpenGL functors

OpenGL rendering is being done also by 1D functors (dispatched for the type to be rendered). Since it is sufficient to have exactly one class for each rendered type, the functors are found automatically. Their base functor types are `GlShapeFunctor`, `GlBoundFunctor`, `GlIGeomFunctor` and so on. These classes register the type they render using the `RENDERS` macro:

```
class Gl1_Sphere: public GlShapeFunctor {
   public :
      virtual void go(const shared_ptr<Shape>&,
         const shared_ptr<State>&,
         bool wire,
         const GLViewInfo&
      );
   RENDERS(Sphere);
   YADE_CLASS_BASE_DOC_STATICATTRS(Gl1_Sphere,GlShapeFunctor,"docstring",
      ((Type1,staticAttr1,informativeDefault,"docstring"))
      /* … */
   );
};
REGISTER_SERIALIZABLE(Gl1_Sphere);
```

You can list available functors of a particular type by querying child classes of the base functor:

```
Yade [141]: yade.system.childClasses('GlShapeFunctor')
Out[141]:
{'Gl1_Box',
 'Gl1_ChainedCylinder',
 'Gl1_Cylinder',
 'Gl1_Facet',
 'Gl1_GridConnection',
 'Gl1_Polyhedra',
 'Gl1_Sphere',
 'Gl1_Tetra',
 'Gl1_Wall'}
```

Note: OpenGL functors may disappear in the future, being replaced by virtual functions of each class that can be rendered.

4.4.7 Parallel execution

Yade was originally not designed with parallel computation in mind, but rather with maximum flexibility (for good or for bad). Parallel execution was added later; in order to not have to rewrite whole Yade from scratch, relatively non-instrusive way of parallelizing was used: OpenMP. OpenMP is standartized shared-memory parallel execution environment, where parallel sections are marked by special `#pragma` in the code (which means that they can compile with compiler that doesn't support OpenMP) and a few functions to query/manipulate OpenMP runtime if necessary.

There is parallelism at 3 levels:

- Computation, interaction (python, GUI) and rendering threads are separate. This is done via regular threads (boost::threads) and is not related to OpenMP.
- *ParallelEngine* can run multiple engine groups (which are themselves run serially) in parallel; it rarely finds use in regular simulations, but it could be used for example when coupling with an independent expensive computation:

```
ParallelEngine([
        [Engine1(),Engine2()],    # Engine1 will run before Engine2
        [Engine3()]               # Engine3() will run in parallel with the group [Engine1(),Engine2()]
                                  # arbitrary number of groups can be used
])
```

`Engine2` will be run after `Engine1`, but in parallel with `Engine3`.

> **Warning:** It is your reponsibility to avoid concurrent access to data when using ParallelEngine. Make sure you understand *very well* what the engines run in parallel do.

- Parallelism inside Engines. Some loops over bodies or interactions are parallelized (notably *InteractionLoop* and *NewtonIntegrator*, which are treated in detail later (FIXME: link)):

```
#pragma omp parallel for
for(long id=0; id<size; id++){
   const shared_ptr<Body>& b(scene->bodies[id]);
   /* … */
}
```

> **Note:** OpenMP requires loops over contiguous range of integers (OpenMP 3 also accepts containers with random-access iterators).

If you consider running parallelized loop in your engine, always evalue its benefits. OpenMP has some overhead fo creating threads and distributing workload, which is proportionally more expensive if the loop body execution is fast. The results are highly hardware-dependent (CPU caches, RAM controller).

Maximum number of OpenMP threads is determined by the `OMP_NUM_THREADS` environment variable and is constant throughout the program run. Yade main program also sets this variable (before loading OpenMP libraries) if you use the `-j`/`--threads` option. It can be queried at runtime with the `omp_get_max_threads` function.

At places which are susceptible of being accessed concurrently from multiple threads, Yade provides some mutual exclusion mechanisms, discussed elsewhere (FIXME):

- simultaneously writeable container for *ForceContainer*,
- mutex for *Body::state*.

4.4.8 Timing

Yade provides 2 services for measuring time spent in different pars of the code. One has the granularity of engine and can be enabled at runtime. The other one is finer, but requires adjusting and recompiling the code being measured.

Per-engine timing

The coarser timing works by merely accumulating numebr of invocations and time (with the precision of the `clock_gettime` function) spent in each engine, which can be then post-processed by associated Python module `yade.timing`. There is a static bool variable controlling whether such measurements take place (disabled by default), which you can change

```
TimingInfo::enabled=True;            // in c++
```

```
O.timingEnabled=True                 ## in python
```

After running the simulation, `yade.timing.stats()` function will show table with the results and percentages:

```
Yade [142]: TriaxialTest(numberOfGrains=100).load()

Yade [143]: O.engines[0].label='firstEngine'    ## labeled engines will show by labels in the stats table

Yade [144]: import yade.timing;

Yade [145]: O.timingEnabled=True

Yade [146]: yade.timing.reset()                    ## not necessary if used for the first time

Yade [147]: O.run(50); O.wait()

Yade [148]: yade.timing.stats()
Name                                                    Count                Time            Rel. time
-------------------------------------------------------------------------------------------------------
"firstEngine"                                              50              126us                0.87%
InsertionSortCollider                                      27             3653us               25.17%
InteractionLoop                                            50             7074us               48.73%
GlobalStiffnessTimeStepper                                  2               34us                0.24%
TriaxialCompressionEngine                                  50             1147us                7.90%
TriaxialStateRecorder                                       3              372us                2.56%
NewtonIntegrator                                           50             2109us               14.53%
TOTAL                                                                     14517us              100.00%
```

Exec count and time can be accessed and manipulated through `Engine::timingInfo` from c++ or `Engine().execCount` and `Engine().execTime` properties in Python.

In-engine and in-functor timing

Timing within engines (and functors) is based on *TimingDeltas* class. It is made for timing loops (functors' loop is in their respective dispatcher) and stores cummulatively time differences between *checkpoints*.

Note: Fine timing with `TimingDeltas` will only work if timing is enabled globally (see previous section). The code would still run, but giving zero times and exec counts.

---

1. Engine::timingDeltas must point to an instance of *TimingDeltas* (prefferably instantiate *TimingDeltas* in the constructor):

```
// header file
class Law2_ScGeom_CpmPhys_Cpm: public LawFunctor {
   /* … */
   YADE_CLASS_BASE_DOC_ATTRS_CTOR(Law2_ScGeom_CpmPhys_Cpm,LawFunctor,"docstring",
      /* attrs */,
      /* constructor */
      timingDeltas=shared_ptr<TimingDeltas>(new TimingDeltas); // timingDeltas object is automaticall
   );
   // ...
};
```

2. Inside the loop, start the timing by calling `timingDeltas->start();`
3. At places of interest, call `timingDeltas->checkpoint("label")`. The label is used only for post-processing, data are stored based on the checkpoint position, not the label.

> **Warning:** Checkpoints must be always reached in the same order, otherwise the timing data will be garbage. Your code can still branch, but you have to put checkpoints to places which are in common.

```
void Law2_ScGeom_CpmPhys_Cpm::go(shared_ptr<IGeom>& _geom,
                                 shared_ptr<IPhys>& _phys,
```

```
                                        Interaction* I,
                                        Scene* scene)
        {
           timingDeltas->start();                        // the point at which the first timing starts
           // prepare some variables etc here
           timingDeltas->checkpoint("setup");
           // find geometrical data (deformations) here
           timingDeltas->checkpoint("geom");
           // compute forces here
           timingDeltas->checkpoint("material");
           // apply forces, cleanup here
           timingDeltas->checkpoint("rest");
        }
```

4. **Alternatively, you can compile Yade using -DENABLE_PROFILING=1 cmake option and use pre**

```
        void Law2_ScGeom_CpmPhys_Cpm::go(shared_ptr<IGeom>& _geom,
                                        shared_ptr<IPhys>& _phys,
                                        Interaction* I,
                                        Scene* scene)
        {
           TIMING_DELTAS_START();
           // prepare some variables etc here
           TIMING_DELTAS_CHECKPOINT("setup")
           // find geometrical data (deformations) here
           TIMING_DELTAS_CHECKPOINT("geom")
           // compute forces here
           TIMING_DELTAS_CHECKPOINT("material")
           // apply forces, cleanup here
           TIMING_DELTAS_CHECKPOINT("rest")
        }
```

The output might look like this (note that functors are nested inside dispatchers and `TimingDeltas` inside their engine/functor):

```
Name                                  Count                 Time          Rel. time
-------------------------------------------------------------------------------------
ForceReseter                          400                 9449µs               0.01%
BoundDispatcher                       400               1171770µs              1.15%
InsertionSortCollider                 400               9433093µs              9.24%
IGeomDispatcher       400             15177607µs              14.87%
IPhysDispatcher        400              9518738µs               9.33%
LawDispatcher                         400              64810867µs             63.49%
  Law2_ScGeom_CpmPhys_Cpm
    setup                             4926145             7649131µs             15.25%
    geom                              4926145            23216292µs             46.28%
    material                          4926145             8595686µs             17.14%
    rest                              4926145            10700007µs             21.33%
    TOTAL                                                50161117µs            100.00%
NewtonIntegrator                      400               1866816µs              1.83%
"strainer"                            400                 21589µs              0.02%
"plotDataCollector"                   160                 64284µs              0.06%
"damageChecker"                         9                  3272µs              0.00%
TOTAL                                                   102077490µs            100.00%
```

Warning: Do not use *TimingDeltas* in parallel sections, results might not be meaningful. In particular, avoid timing functors inside *InteractionLoop* when running with multiple OpenMP threads.

`TimingDeltas` data are accessible from Python as list of (*label*,*time*,*count*) tuples, one tuple representing each checkpoint:

```
deltas=someEngineOrFunctor.timingDeltas.data()
deltas[0][0] # 0th checkpoint label
deltas[0][1] # 0th checkpoint time in nanoseconds
deltas[0][2] # 0th checkpoint execution count
deltas[1][0] # 1st checkpoint label
             # …
deltas.reset()
```

Timing overhead

The overhead of the coarser, per-engine timing, is very small. For simulations with at least several hundreds of elements, they are below the usual time variance (a few percent).

The finer *TimingDeltas* timing can have major performance impact and should be only used during debugging and performance-tuning phase. The parts that are file-timed will take disproportionally longer time that the rest of engine; in the output presented above, *LawDispatcher* takes almost of total simulation time in average, but the number would be twice of thrice lower typically (note that each checkpoint was timed almost 5 million times in this particular case).

4.4.9 OpenGL Rendering

Yade provides 3d rendering based on QGLViewer. It is not meant to be full-featured rendering and post-processing, but rather a way to quickly check that scene is as intended or that simulation behaves sanely.

Note: Although 3d rendering runs in a separate thread, it has performance impact on the computation itself, since interaction container requires mutual exclusion for interaction creation/deletion. The `InteractionContainer::drawloopmutex` is either held by the renderer (*OpenGLRenderingEngine*) or by the insertion/deletion routine.

Warning: There are 2 possible causes of crash, which are not prevented because of serious performance penalty that would result:

1. access to *BodyContainer*, in particular deleting bodies from simulation; this is a rare operation, though.
2. deleting Interaction::phys or Interaction::geom.

Renderable entities (*Shape*, *State*, *Bound*, *IGeom*, *IPhys*) have their associated *OpenGL functors*. An entity is rendered if

1. Rendering such entities is enabled by appropriate attribute in *OpenGLRenderingEngine*
2. Functor for that particular entity type is found via the *dispatch mechanism*.

`Gl1_*` functors operating on Body's attributes (*Shape*, *State*, *Bound*) are called with the OpenGL context translated and rotated according to *State::pos* and *State::ori*. Interaction functors work in global coordinates.

4.5 Simulation framework

Besides the support framework mentioned in the previous section, some functionality pertaining to simulation itself is also provided.

There are special containers for storing bodies, interactions and (generalized) forces. Their internal functioning is normally opaque to the programmer, but should be understood as it can influence performance.

4.5.1 Scene

`Scene` is the object containing the whole simulation. Although multiple scenes can be present in the memory, only one of them is active. Saving and loading (serializing and deserializing) the `Scene` object should make the simulation run from the point where it left off.

Note: All *Engines* and functors have interally a `Scene* scene` pointer which is updated regularly by engine/functor callers; this ensures that the current scene can be accessed from within user code.

For outside functions (such as those called from python, or static functions in `Shop`), you can use `Omega::instance().getScene()` to retrieve a `shared_ptr<Scene>` of the current scene.

4.5.2 Body container

Body container is linear storage of bodies. Each body in the simulation has its unique *id*, under which it must be found in the *BodyContainer*. Body that is not yet part of the simulation typically has id equal to invalid value `Body::ID_NONE`, and will have its `id` assigned upon insertion into the container. The requirements on *BodyContainer* are

- O(1) access to elements,
- linear-addressability (0…n indexability),
- store `shared_ptr`, not objects themselves,
- *no* mutual exclusion for insertion/removal (this must be assured by the called, if desired),
- intelligent allocation of `id` for new bodies (tracking removed bodies),
- easy iteration over all bodies.

Note: Currently, there is "abstract" class `BodyContainer`, from which derive concrete implementations; the initial idea was the ability to select at runtime which implementation to use (to find one that performs the best for given simulation). This incurs the penalty of many virtual function calls, and will probably change in the future. All implementations of BodyContainer were removed in the meantime, except `BodyVector` (internally a `vector<shared_ptr<Body> >` plus a few methods around), which is the fastest.

Insertion/deletion

Body insertion is typically used in *FileGenerator*'s:

```
shared_ptr<Body> body(new Body);
// … (body setup)
scene->bodies->insert(body); // assigns the id
```

Bodies are deleted only rarely:

```
scene->bodies->erase(id);
```

> **Warning:** Since mutual exclusion is not assured, never insert/erase bodies from parallel sections, unless you explicitly assure there will be no concurrent access.

Iteration

The container can be iterated over using `FOREACH` macro (shorthand for `BOOST_FOREACH`):

```
FOREACH(const shared_ptr<Body>& b, *scene->bodies){
   if(!b) continue;                        // skip deleted bodies
   /* do something here */
}
```

Note a few important things:

1. Always use `const shared_ptr<Body>&` (const reference); that avoids incrementing and decrementing the reference count on each `shared_ptr`.
2. Take care to skip NULL bodies (`if(!b) continue`): deleted bodies are deallocated from the container, but since body id's must be persistent, their place is simply held by an empty `shared_ptr<Body>()` object, which is implicitly convertible to `false`.

In python, the BodyContainer wrapper also has iteration capabilities; for convenience (which is different from the c++ iterator), NULL bodies as silently skipped:

```
Yade [149]: O.bodies.append([Body(),Body(),Body()])
Out[149]: [0, 1, 2]

Yade [150]: O.bodies.erase(1)
Out[150]: True

Yade [151]: [b.id for b in O.bodies]
Out[151]: [0, 2]
```

In loops parallelized using OpenMP, the loop must traverse integer interval (rather than using iterators):

```
const long size=(long)bodies.size();       // store this value, since it doesn't change during the loop
#pragma omp parallel for
for(long _id=0; _id<size; _id++){
   const shared_ptr<Body>& b(bodies[_id]);
   if(!b) continue;
   /* … */
}
```

4.5.3 InteractionContainer

Interactions are stored in special container, and each interaction must be uniquely identified by pair of ids (id1,id2).

- O(1) access to elements,
- linear-addressability (0…n indexability),
- store `shared_ptr`, not objects themselves,
- mutual exclusion for insertion/removal,
- easy iteration over all interactions,
- addressing symmetry, i.e. interaction(id1,id2) interaction(id2,id1)

Note: As with BodyContainer, there is "abstract" class InteractionContainer, and then its concrete implementations. Currently, only InteractionVecMap implementation is used and all the other were removed. Therefore, the abstract InteractionContainer class may disappear in the future, to avoid unnecessary virtual calls.

Further, there is a blueprint for storing interactions inside bodies, as that would give extra advantage of quickly getting all interactions of one particular body (currently, this necessitates loop over all interactions); in that case, InteractionContainer would disappear.

Insert/erase

Creating new interactions and deleting them is delicate topic, since many eleents of simulation must be synchronized; the exact workflow is described in *Handling interactions*. You will almost certainly never need to insert/delete an interaction manually from the container; if you do, consider designing your code differently.

```
// both insertion and erase are internally protected by a mutex,
// and can be done from parallel sections safely
scene->interactions->insert(shared_ptr<Interaction>(new Interactions(id1,id2)));
scene->interactions->erase(id1,id2);
```

Iteration

As with BodyContainer, iteration over interactions should use the `FOREACH` macro:

```
FOREACH(const shared_ptr<Interaction>& i, *scene->interactions){
   if(!i->isReal()) continue;
   /* … */
}
```

Again, note the usage const reference for `i`. The check `if(!i->isReal())` filters away interactions that exist only *potentially*, i.e. there is only *Bound* overlap of the two bodies, but not (yet) overlap of bodies themselves. The `i->isReal()` function is equivalent to `i->geom && i->phys`. Details are again explained in *Handling interactions*.

In some cases, such as OpenMP-loops requiring integral index (OpenMP >= 3.0 allows parallelization using random-access iterator as well), you need to iterate over interaction indices instead:

```
inr nIntr=(int)scene->interactions->size(); // hoist container size
#pragma omp parallel for
for(int j=0; j<nIntr, j++){
   const shared_ptr<Interaction>& i(scene->interactions[j]);
   if(!->isReal()) continue;
   /* … */
}
```

4.5.4 ForceContainer

ForceContainer holds "generalized forces", i.e. forces, torques, (explicit) dispalcements and rotations for each body.

During each computation step, there are typically 3 phases pertaining to forces:

1. Resetting forces to zero (usually done by the *ForceResetter* engine)
2. Incrementing forces from parallel sections (solving interactions – from *LawFunctor*)
3. Reading absolute force values sequentially for each body: forces applied from different interactions are summed together to give overall force applied on that body (*NewtonIntegrator*, but also various other engine that read forces)

This scenario leads to special design, which allows fast parallel write access:

- each thread has its own storage (zeroed upon request), and only writes to its own storage; this avoids concurrency issues. Each thread identifies itself by the omp_get_thread_num() function provided by the OpenMP runtime.
- before reading absolute values, the container must be synchronized, i.e. values from all threads are summed up and stored separately. This is a relatively slow operation and we provide ForceContainer::syncCount that you might check to find cummulative number of synchronizations and compare it against number of steps. Ideally, ForceContainer is only synchronized once at each step.
- the container is resized whenever an element outside the current range is read/written to (the read returns zero in that case); this avoids the necessity of tracking number of bodies, but also is potential danger (such as `scene->forces.getForce(1000000000)`, which will probably exhaust your RAM). Unlike c++, Python does check given id against number of bodies.

```
// resetting forces (inside ForceResetter)
scene->forces.reset()

// in a parallel section
scene->forces.addForce(id,force); // add force

// container is not synced after we wrote to it, sync before reading
scene->forces.sync();
const Vector3r& f=scene->forces.getForce(id);
```

Synchronization is handled automatically if values are read from python:

```
Yade [152]: O.bodies.append(Body())
Out[152]: 3

Yade [153]: O.forces.addF(0,Vector3(1,2,3))

Yade [154]: O.forces.f(0)
Out[154]: Vector3(1,2,3)

Yade [155]: O.forces.f(100)
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/build/yade-Swrxdd/yade-1.20.0/debian/tmp/usr/lib/x86_64-linux-gnu/yade/py/yade/__init__.pyc in <module>()
----> 1 O.forces.f(100)

IndexError: Body id out of range.
```

4.5.5 Handling interactions

Creating and removing interactions is a rather delicate topic and number of components must cooperate so that the whole behaves as expected.

Terminologically, we distinguish

potential interactions, having neither *geometry* nor *physics*. *Interaction.isReal* can be used to query the status (`Interaction::isReal()` in c++).

real interactions, having both *geometry* and *physics*. Below, we shall discuss the possibility of interactions that only have geometry but no physics.

During each step in the simulation, the following operations are performed on interactions in a typical simulation:

1. Collider creates potential interactions based on spatial proximity. Not all pairs of bodies are susceptible of entering interaction; the decision is done in Collider::mayCollide:
   - clumps may not enter interactions (only their members can)
   - clump members may not interact if they belong to the same clump
   - bitwise AND on both bodies' *masks* must be non-zero (i.e. there must be at least one bit set in common)
2. Collider erases interactions that were requested for being erased (see below).
3. *InteractionLoop* (via *IGeomDispatcher*) calls appropriate *IGeomFunctor* based on *Shape* combination of both bodies, if such functor exists. For real interactions, the functor updates associated *IGeom*. For potential interactions, the functor returns

   `false` if there is no geometrical overlap, and the interaction will stillremain potential-only

   `true` if there is geometrical overlap; the functor will have created an *IGeom* in such case.

   Note: For *real* interactions, the functor *must* return `true`, even if there is no more

spatial overlap between bodies. If you wish to delete an interaction without geometrical overlap, you have to do this in the *LawFunctor*.

This behavior is deliberate, since different *laws* have different requirements, though ideally using relatively small number of generally useful *geometry functors*.

Note: If there is no functor suitable to handle given combination of *shapes*, the interaction will be left in potential state, without raising any error.

---

4. For real interactions (already existing or just created in last step), *InteractionLoop* (via *IPhysDispatcher*) calls appropriate *IPhysFunctor* based on *Material* combination of both bodies. The functor *must* update (or create, if it doesn't exist yet) associated *IPhys* instance. It is an error if no suitable functor is found, and an exception will be thrown.
5. For real interactions, *InteractionLoop* (via *LawDispatcher*) calls appropriate *LawFunctor* based on combination of *IGeom* and *IPhys* of the interaction. Again, it is an error if no functor capable of handling it is found.
6. *LawDispatcher* takes care of erasing those interactions that are no longer active (such as if bodies get too far apart for non-cohesive laws; or in case of complete damage for damage models). This is triggered by the *LawFunctor* returning false. For this reason it is of upmost importance for the *LawFunctor* to return consistently.

Such interaction will not be deleted immediately, but will be reset to potential state. At the next execution of the collider `InteractionContainer::conditionalyEraseNonReal` will be called, which will completely erase interactions only if the bounding boxes ceased to overlap; the rest will be kept in potential state.

Creating interactions explicitly

Interactions may still be created explicitly with *utils.createInteraction*, without any spatial requirements. This function searches current engines for dispatchers and uses them. *IGeomFunctor* is called with the `force` parameter, obliging it to return `true` even if there is no spatial overlap.

4.5.6 Associating Material and State types

Some models keep extra *state* information in the *Body.state* object, therefore requiring strict association of a *Material* with a certain *State* (for instance, *CpmMat* is associated to *CpmState* and this combination is supposed by engines such as *CpmStateUpdater*).

If a *Material* has such a requirement, it must override 2 virtual methods:

1. *Material.newAssocState*, which returns a new *State* object of the corresponding type. The default implementation returns *State* itself.
2. *Material.stateTypeOk*, which checks whether a given *State* object is of the corresponding type (this check is run at the beginning of the simulation for all particles).

In c++, the code looks like this (for *CpmMat*):

```
class CpmMat: public FrictMat {
   public:
      virtual shared_ptr<State> newAssocState() const { return shared_ptr<State>(new CpmState); }
      virtual bool stateTypeOk(State* s) const { return (bool)dynamic_cast<CpmState*>(s); }
   /* ... */
};
```

This allows one to construct *Body* objects from functions such as *utils.sphere* only by knowing the requires *Material* type, enforcing the expectation of the model implementor.

4.6 Runtime structure

4.6.1 Startup sequence

Yade's main program is python script in core/main/main.py.in; the build system replaces a few `${variables}` in that file before copying it to its install location. It does the following:

1. Process command-line options, set environment variables based on those options.
2. Import main yade module (`import yade`), residing in py/__init__.py.in. This module locates plugins (recursive search for files `lib*.so` in the `lib` installation directory). yade.boot module is used to setup temporary directory, … and, most importantly, loads plugins.
3. Manage further actions, such as running scripts given at command line, opening *qt.Controller* (if desired), launching the `ipython` prompt.

4.6.2 Singletons

There are several "global variables" that are always accessible from c++ code; properly speaking, they are Singletons, classes of which exactly one instance always exists. The interest is to have some general functionality acessible from anywhere in the code, without the necessity of passing pointers to such objects everywhere. The instance is created at startup and can be always retrieved (as non-const reference) using the `instance()` static method (e.g. `Omega::instance().getScene()`).

There are 3 singletons:

`SerializableSingleton` Handles serialization/deserialization; it is not used anywhere except for the serialization code proper.

`ClassFactory` Registers classes from plugins and able to factor instance of a class given its name as string (the class must derive from `Factorable`). Not exposed to python.

`Omega` Access to simulation(s); deserves separate section due to its importance.

Omega

The *Omega* class handles all simulation-related functionality: loading/saving, running, pausing.

In python, the wrapper class to the singleton is instantiated [5] as global variable `O`. For convenience, *Omega* is used as proxy for scene's attribute: although multiple `Scene` objects may be instantiated in c++, it is always the current scene that *Omega* represents.

The correspondence of data is literal: *Omega.materials* corresponds to `Scene::materials` of the current scene; likewise for *materials*, *bodies*, *interactions*, *tags*, *cell*, *engines*, *initializers*, *miscParams*.

To give an overview of (some) variables:

Python	c++
Omega.iter	`Scene::iter`
Omega.dt	`Scene::dt`
Omega.time	`Scene::time`
Omega.realtime	`Omega::getRealTime()`
Omega.stopAtIter	`Scene::stopAtIter`

`Omega` in c++ contains pointer to the current scene (`Omega::scene`, retrieved by `Omega::instance().getScene()`). Using *Omega.switchScene*, it is possible to swap this pointer with `Omega::sceneAnother`, a completely independent simulation. This can be useful for example (and this motivated this functionality) if while constructing simulation, another simulation has to be run to dynamically generate (i.e. by running simulation) packing of spheres.

[5] It is understood that instantiating `Omega()` in python only instantiates the wrapper class, not the singleton itself.

4.6.3 Engine loop

Running simulation consists in looping over *Engines* and calling them in sequence. This loop is defined in `Scene::moveToNextTimeStep` function in core/Scene.cpp. Before the loop starts, *O.initializers* are called; they are only run once. The engine loop does the following in each iteration over *O.engines*:

1. set `Engine::scene` pointer to point to the current `Scene`.
2. Call `Engine::isActivated()`; if it returns `false`, the engine is skipped.
3. Call `Engine::action()`
4. If *O.timingEnabled*, increment *Engine::execTime* by the difference from the last time reading (either after the previous engine was run, or immediately before the loop started, if this engine comes first). Increment *Engine::execCount* by 1.

After engines are processed, *virtual time* is incremented by *timestep* and *iteration number* is incremented by 1.

Background execution

The engine loop is (normally) executed in background thread (handled by SimulationFlow class), leaving foreground thread free to manage user interaction or running python script. The background thread is managed by *O.run()* and *O.pause()* commands. Foreground thread can be blocked until the loop finishes using *O.wait()*.

Single iteration can be run without spawning additional thread using *O.step()*.

4.7 Python framework

4.7.1 Wrapping c++ classes

Each class deriving from *Serializable* is automatically exposed to python, with access to its (registered) attributes. This is achieved via *YADE_CLASS_BASE_DOC_* macro family*. All classes registered in class factory are default-constructed in `Omega::buildDynlibDatabase`. Then, each serializable class calls `Serializable::pyRegisterClass` virtual method, which injects the class wrapper into (initially empty) `yade.wrapper` module. `pyRegisterClass` is defined by `YADE_CLASS_BASE_DOC` and knows about class, base class, docstring, attributes, which subsequently all appear in boost::python class definition.

Wrapped classes define special constructor taking keyword arguments corresponding to class attributes; therefore, it is the same to write:

```
Yade [156]: f1=ForceEngine()

Yade [157]: f1.ids=[0,4,5]

Yade [158]: f1.force=Vector3(0,-1,-2)
```

and

```
Yade [159]: f2=ForceEngine(ids=[0,4,5],force=Vector3(0,-1,-2))

Yade [160]: print f1.dict()
{'ompThreads': -1, 'force': Vector3(0,-1,-2), 'ids': [0, 4, 5], 'dead': False, 'label': ''}

Yade [161]: print f2.dict()
{'ompThreads': -1, 'force': Vector3(0,-1,-2), 'ids': [0, 4, 5], 'dead': False, 'label': ''}
```

Wrapped classes also inherit from *Serializable* several special virtual methods: *dict()* returning all registered class attributes as dictionary (shown above), *clone()* returning copy of instance (by copying attribute values), *updateAttrs()* and *updateExistingAttrs()* assigning attributes from given dictionary (the former thrown for unknown attribute, the latter doesn't).

Read-only property `name` wraps c++ method `getClassName()` returning class name as string. (Since c++ class and the wrapper class always have the same name, getting python type using `__class__` and its property `__name__` will give the same value).

```
Yade [162]: s=Sphere()

Yade [163]: s.__class__.__name__
Out[163]: 'Sphere'
```

4.7.2 Subclassing c++ types in python

In some (rare) cases, it can be useful to derive new class from wrapped c++ type in pure python. This is done in the *yade.pack module* module: *Predicate* is c++ base class; from this class, several c++ classes are derived (such as *inGtsSurface*), but also python classes (such as the trivial *inSpace* predicate). `inSpace` derives from python class `Predicate`; it is, however, not direct wrapper of the c++ `Predicate` class, since virtual methods would not work.

`boost::python` provides special `boost::python::wrapper` template for such cases, where each overridable virtual method has to be declared explicitly, requesting python override of that method, if present. See Overridable virtual functions for more details.

4.7.3 Reference counting

Python internally uses reference counting on all its objects, which is not visible to casual user. It has to be handled explicitly if using pure Python/C API with `Py_INCREF` and similar functions.

`boost::python` used in Yade fortunately handles reference counting internally. Additionally, it automatically integrates reference counting for `shared_ptr` and python objects, if class `A` is wrapped as `boost::python::class_<A,shared_ptr<A>>`. Since *all* Yade classes wrapped using *YADE_CLASS_BASE_DOC_* macro family* are wrapped in this way, returning `shared_ptr<…>` objects from is the preferred way of passing objects from c++ to python.

Returning `shared_ptr` is much more efficient, since only one pointer is returned and reference count internally incremented. Modifying the object from python will modify the (same) object in c++ and vice versa. It also makes sure that the c++ object will not be deleted as long as it is used somewhere in python, preventing (important) source of crashes.

4.7.4 Custom converters

When an object is passed from c++ to python or vice versa, then either

1. the type is basic type which is transparently passed between c++ and python (int, bool, std::string etc)
2. the type is wrapped by boost::python (such as Yade classes, `Vector3` and so on), in which case wrapped object is returned; [6]

Other classes, including template containers such as `std::vector` must have their custom converters written separately. Some of them are provided in py/wrapper/customConverters.cpp, notably converters between python (homogeneous, i.e. with all elements of the same type) sequences and c++ `std::vector` of corresponding type; look in that source file to add your own converter or for inspiration.

When an object is crossing c++/python boundary, boost::python's global "converters registry" is searched for class that can perform conversion between corresponding c++ and python types. The

[6] Wrapped classes are automatically registered when the class wrapper is created. If wrapped class derives from another wrapped class (and if this dependency is declared with the `boost::python::bases` template, which Yade's classes do automatically), parent class must be registered before derived class, however. (This is handled via loop in `Omega::buildDynlibDatabase`, which reiterates over classes, skipping failures, until they all successfully register) Math classes (Vector3, Matrix3, Quaternion) are wrapped in minieigen, which is available as a separate package. Use your package manager to install it.

"converters registry" is common for the whole program instance: there is no need to register converters in each script (by importing `_customConverters`, for instance), as that is done by yade at startup already.

Note: Custom converters only work for value that are passed by value to python (not "by reference"): some attributes defined using *YADE_CLASS_BASE_DOC_* macro family* are passed by value, but if you define your own, make sure that you read and understand Why is my automatic to-python conversion not being found?.

In short, the default for `def_readwrite` and `def_readonly` is to return references to underlying c++ objects, which avoids performing conversion on them. For that reason, return value policy must be set to `return_by_value` explicitly, using slighly more complicated `add_property` syntax, as explained at the page referenced.

4.8 Maintaining compatibility

In Yade development, we identified compatibility to be very strong desire of users. Compatibility concerns python scripts, *not* simulations saved in XML or old c++ code.

4.8.1 Renaming class

Script scripts/rename-class.py should be used to rename class in `c++` code. It takes 2 parameters (old name and new name) and must be run from top-level source directory:

```
$ scripts/rename-class.py OldClassName NewClassName
Replaced 4 occurences, moved 0 files and 0 directories
Update python scripts (if wanted) by running: perl -pi -e 's/\bOldClassName\b/NewClassName/g' `ls **/*.py |grep
```

This has the following effects:

1. If file or directory has basename `OldClassName` (plus extension), it will be renamed using `bzr`.
2. All occurences of whole word `OldClassName` will be replaced by `NewClassName` in c++ sources.
3. An extry is added to py/system.py, which contains map of deprecated class names. At yade startup, proxy class with `OldClassName` will be created, which issues a `DeprecationWarning` when being instantiated, informing you of the new name you should use; it creates an instance of `NewClassName`, hence not disruting your script's functioning:

```
Yade [3]: SimpleViscoelasticMat()
/usr/local/lib/yade-trunk/py/yade/__init__.py:1: DeprecationWarning: Class `SimpleViscoelasticMat' was ren
->  [3]: <ViscElMat instance at 0x2d06770>
```

As you have just been informed, you can run `yade --update` to all old names with their new names in scripts you provide:

```
$ yade-trunk --update script1.py some/where/script2.py
```

This gives you enough freedom to make your class name descriptive and intuitive.

4.8.2 Renaming class attribute

Renaming class attribute is handled from c++ code. You have the choice of merely warning at accessing old attribute (giving the new name), or of throwing exception in addition, both with provided explanation. See `deprec` parameter to *YADE_CLASS_BASE_DOC_* macro family* for details.

4.9 Debian packaging instructions

In order to make parallel installation of several Yade version possible, we adopted similar strategy as e.g. `gcc` packagers in Debian did:

1. Real Yade packages are named `yade-0.30` (for stable versions) or `yade-bzr2341` (for snapshots).
2. They provide `yade` or `yade-snapshot` virtual packages respectively.
3. Each source package creates several installable packages (using `bzr2341` as example version):
   (a) `yade-bzr2341` with the optimized binaries; the executable binary is `yade-bzr2341` (`yade-bzr2341-multi`, …)
   (b) `yade-bzr2341-dbg` with debug binaries (debugging symbols, non-optimized, and with crash handlers); the executable binary is `yade-bzr2341-dbg`
   (c) `yade-bzr2341-doc` with sample scripts and some documentation (see bug #398176 however)
   (d) (future?) `yade-bzr2341-reference` with reference documentation (see bug #401004)
4. Using Debian alternatives, the highest installed package provides additionally commands without the version specification like `yade`, `yade-multi`, … as aliases to that version's binaries. (`yade-dbg`, … for the debuggin packages). The exact rule is:
   (a) Stable releases have always higher priority than snapshots
   (b) Higher versions/revisions have higher pripority than lower versions/revisions.

4.9.1 Prepare source package

Debian packaging files are located in debian/ directory. They contain build recipe debian/rules, dependecy and package declarations debian/control and maintainer scripts. Some of those files are only provided as templates, where some variables (such as version number) are replaced by special script.

The script scripts/debian-prep processes templates in debian/ and creates files which can be used by debian packaging system. Before running this script:

1. If you are releasing stable version, make sure there is file named `RELEASE` containing single line with version number (such as `0.30`). This will make scripts/debian-prep create release packages. In absence of this file, snapshots packaging will be created instead. Release or revision number (as detected by running `bzr revno` in the source tree) is stored in `VERSION` file, where it is picked up during package build and embedded in the binary.
2. Find out for which debian/ubuntu series your package will be built. This is the name that will appear on the top of (newly created) `debian/changelog` file. This name will be usually `unstable`, `testing` or `stable` for debian and `karmic`, `lucid` etc for ubuntu. WHen package is uploaded to Launchpad's build service, the package will be built for this specified release.

Then run the script from the top-level directory, giving series name as its first (only) argument:

```
$ scripts/debian-prep lucid
```

After this, signed debian source package can be created:

```
$ debuild -S -sa -k62A21250 -I -Iattic
```

(`-k` gives GPG key identifier, `-I` skips `.bzr` and similar directories, `-Iattic` will skip the useless `attic` directory).

4.9.2 Create binary package

Local in-tree build Once files in `debian/` are prepared, packages can be build by issuing:: $ fakeroot debian/rules binary

Clean system build Using `pbuilder` system, package can be built in a chroot containing clean debian/ubuntu system, as if freshly installed. Package dependencies are automatically installed and package build attempted. This is a good way of testing packaging before having the package built remotely at Launchpad. Details are provided at wiki page.

Launchpad build service Launchpad provides service to compile package for different ubuntu releases (series), for all supported architectures, and host archive of those packages for download via APT. Having appropriate permissions at Launchpad (verified GPG key), source package can be uploaded to yade's archive by:

```
$ dput ppa:yade-users/ppa ../yade-bzr2341_1_source.changes
```

After several hours (depending on load of Launchpad build farm), new binary packages will be published at https://launchpad.net/~yade-users/+archive/ppa.

This process is well documented at https://help.launchpad.net/Packaging/PPA.

Chapter 5

Installation

Yade can be installed from packages (pre-compiled binaries) or source code. The choice depends on what you need: if you don't plan to modify Yade itself, package installation is easier. In the contrary case, you must download and install the source code.

5.1 Packages

Pre-built packages are provided for all currently supported Debian and Ubuntu versions of distributions and available on yade-dem.org/packages .

These are `daily` versions of packages and are updating regularly and include all the newly added features.

To install daily-version one needs to add this repository to your /etc/apt/sources.list, add a PGP-key AA915EEB as a trusted and install yadedaily

```
sudo bash -c 'echo "deb http://www.yade-dem.org/packages/ trusty/" >> /etc/apt/sources.list'
wget -O - http://www.yade-dem.org/packages/yadedev_pub.gpg | sudo apt-key add -
sudo apt-get update
sudo apt-get install yadedaily
```

If you have another distribution, not Ubuntu Trusty (Version 14.04 LTS), be sure to use the correct name in the first line (for instance, trusty, jessie or wheezy). For the list of currently supported distributions, please visit yade-dem.org/packages.

After that you can normally start Yade using "yadedaily" or "yadedaily-batch" command. yadedaily on older distributions can have some disabled features due to older library versions, shipped with particular distribution.

Git-repository for packaging stuff is available on GitHub. Each branch corresponds to one distribution e.g. trusty, jessie etc. The scripts for building all of this stuff is here. It uses pbuilder to build packages, so all packages are building in a clean environment.

If you do not need yadedaily-package any more, just remove the corresponding line in /etc/apt/sources.list and the package itself:

```
sudo apt-get remove yadedaily
```

To remove our key from keyring, execute the following command:

```
sudo apt-key remove AA915EEB
```

Since 2011 all Ubuntu versions (starting from 11.10, Oneiric) and Debian (starting from Wheezy) are having already Yade in their main repositories. There are only stable releases are placed. To install the program, run the following:

```
sudo apt-get install yade
```

To check, what version of Yade is in specific distribution, visit the links for Ubuntu and Debian. Debian-Backports repository is updating regularly to bring the newest Yade to a users of stable Debians.

Daily and stable Yade versions can coexist without any conflicts.

5.2 Source code

Installation from source code is reasonable, when you want to add or modify constitutive laws, engines or functions... Installing the latest trunk version allows one to use newly added features, which are not yet available in packaged versions.

5.2.1 Download

If you want to install from source, you can install either a release (numbered version, which is frozen) or the current development version (updated by the developers frequently). You should download the development version (called `trunk`) if you want to modify the source code, as you might encounter problems that will be fixed by the developers. Release version will not be modified (except for updates due to critical and easy-to-fix bugs), but they are in a more stabilized state that trunk generally.

1. Releases can be downloaded from the download page, as compressed archive. Uncompressing the archive gives you a directory with the sources.
2. developement version (trunk) can be obtained from the code repository at github.

We use GIT (the `git` command) for code management (install the `git` package in your distribution and create a GitHub account):

```
git clone git@github.com:yade/trunk.git
```

will download the whole code repository of `trunk`. Check out *Yade on GitHub* for more.

Alternatively, a read-only checkout is possible via https without a GitHub account (easier if you don't want to modify the main Yade branch):

```
git clone https://github.com/yade/trunk.git
```

For those behind firewall, you can download the sources from our GitHub repository as compressed archive.

Release and trunk sources are compiled in the same way. To be notified about new commits into the trunk, use watch option on GitHub.

5.2.2 Prerequisites

Yade relies on a number of external software to run; they are checked before the compilation starts. Some of them are only optional. The last ones are only relevant for using the fluid coupling module (*FlowEngine*).

- cmake build system
- gcc compiler (g++); other compilers will not work; you need g++>=4.2 for openMP support
- boost 1.35 or later
- Qt library
- freeglut3
- libQGLViewer
- python, numpy, ipython
- matplotlib
- eigen algebra library (minimal required version 3.2.1)

- gdb debugger
- sqlite3 database engine
- Loki library
- VTK library (optional but recommended)
- CGAL library (optional)
- SuiteSparse sparse algebra library (fluid coupling, optional, requires eigen>=3.1)
- OpenBLAS optimized and parallelized alternative to the standard blas+lapack (fluid coupling, optional)
- Metis matrix preconditioning (fluid coupling, optional)

Most of the list above is very likely already packaged for your distribution. In case you are confronted with some errors concerning not available packages (e.g. Package libmetis-dev is not available) it may be necessary to add yade external ppa from https://launchpad.net/~yade-users/+archive/external (see below) as well as http://www.yade-dem.org/packages (see the top of this page):

```
sudo add-apt-repository ppa:yade-users/external
sudo apt-get update
```

The following commands have to be executed in command line of corresponding distributions. Just copy&paste to the terminal. To perform commands you should have root privileges

- **Ubuntu**, **Debian** and their derivatives:

```
    sudo apt-get install cmake git freeglut3-dev libloki-dev \
    libboost-all-dev fakeroot dpkg-dev build-essential g++ \
    python-dev ipython python-matplotlib libsqlite3-dev python-numpy python-tk gnuplot \
    libgts-dev python-pygraphviz libvtk5-dev python-scientific libeigen3-dev \
    python-xlib python-qt4 pyqt4-dev-tools gtk2-engines-pixbuf python-argparse \
    libqglviewer-dev python-imaging libjs-jquery python-sphinx python-git python-bibtex \
    libxmu-dev libxi-dev libcgal-dev help2man libbz2-dev zlib1g-dev python-minieigen
```

Some of packages (for example, cmake, eigen3) are mandatory, some of them are optional. Watch for notes and warnings/errors, which are shown by cmake during configuration step. If the missing package is optional, some of Yade features will be disabled (see the messages at the end of configuration).

Additional packages, which can become mandatory later:

```
sudo apt-get install python-gts
```

For effective usage of direct solvers in the PFV-type fluid coupling, the following libraries are recommended, together with eigen>=3.1: blas, lapack, suitesparse, and metis. All four of them are available in many different versions. Different combinations are possible and not all of them will work. The following was found to be effective on recent deb-based systems. On ubuntu 12.04, better compile openblas with USE_OPENMP=1, else yade will run on a single core:

```
sudo apt-get install libopenblas-dev libsuitesparse-metis-dev
```

Some packages listed here are relatively new and they can be absent in your distribution (for example, libmetis-dev or python-gts). They can be installed from yade-dem.org/packages or from our external PPA. If not installed the related features will be disabled automatically.

If you are using other distribution, than Debian or its derivatives, you should install the softwares listed above. Their names can differ from the names of Debian-packages.

Warning: If you have Ubuntu 14.04 Trusty, you need to add -DCMAKE_CXX_FLAGS="-frounding-math" during the configuration step of compilation (see below) or to install libcgal-dev from our external PPA. Otherwise the following error occurs on AMD64 architectures:

```
terminate called after throwing an instance of 'CGAL::Assertion_exception'
what():  CGAL ERROR: assertion violation!
Expr: -CGAL_IA_MUL(-1.1, 10.1) != CGAL_IA_MUL(1.1, 10.1)
File: /usr/include/CGAL/Interval_nt.h
Line: 209
Explanation: Wrong rounding: did you forget the  -frounding-math  option if you use GCC (or  -fp-model strict
Aborted
```

5.2.3 Compilation

You should create a separate build-place-folder, where Yade will be configured and where the source code will be compiled. Here is an example for a folder structure:

```
myYade/                 ## base directory
        trunk/          ## folder for sourcecode in which you use github
        build/          ## folder in which sources will be compiled; build-directory; use cmake here
        install/        ## install folder
```

Then inside this build-directory you should start cmake to configure the compilation process:

```
cmake -DCMAKE_INSTALL_PREFIX=/path/to/installfolder /path/to/sources
```

For the folder structure given above call the following command in folder "build":

cmake -DCMAKE_INSTALL_PREFIX=../install ../trunk

Additional options can be configured in the same line with the following syntax:

```
cmake -DOPTION1=VALUE1 -DOPTION2=VALUE2
```

The following options are available:

- CMAKE_INSTALL_PREFIX: path where Yade should be installed (/usr/local by default)
- LIBRARY_OUTPUT_PATH: path to install libraries (lib by default)
- DEBUG: compile in debug-mode (OFF by default)
- CMAKE_VERBOSE_MAKEFILE: output additional information during compiling (OFF by default)
- SUFFIX: suffix, added after binary-names (version number by default)
- NOSUFFIX: do not add a suffix after binary-name (OFF by default)
- YADE_VERSION: explicitely set version number (is defined from git-directory by default)
- ENABLE_GUI: enable GUI option (ON by default)
- ENABLE_CGAL: enable CGAL option (ON by default)
- ENABLE_VTK: enable VTK-export option (ON by default)
- ENABLE_OPENMP: enable OpenMP-parallelizing option (ON by default)
- ENABLE_GTS: enable GTS-option (ON by default)
- ENABLE_GL2PS: enable GL2PS-option (ON by default)
- ENABLE_LINSOLV: enable LINSOLV-option (ON by default)
- ENABLE_PFVFLOW: enable PFVFLOW-option, FlowEngine (ON by default)
- ENABLE_LBMFLOW: enable LBMFLOW-option, LBM_ENGINE (ON by default)
- ENABLE_SPH: enable SPH-option, Smoothed Particle Hydrodynamics (OFF by default)

- ENABLE_LIQMIGRATION: enable LIQMIGRATION-option, see [Mani2013] for details (OFF by default)
- ENABLE_MASK_ARBITRARY: enable MASK_ARBITRARY option (OFF by default)
- ENABLE_PROFILING: enable profiling, e.g. shows some more metrics, which can define bottlenecks of the code (OFF by default)
- runtimePREFIX: used for packaging, when install directory is not the same is runtime directory (/usr/local by default)
- CHUNKSIZE: used, if you want several sources to be compiled at once. Increases compilation speed and RAM-consumption during it (1 by default)
- VECTORIZE: enables vectorization and alignment in Eigen3 library, experimental (OFF by default)
- USE_QT5: use QT5 for GUI, experimental (OFF by default)

For using an extended parameters of cmake, please, follow the corresponding documentation on cmake-webpage.

Warning: To provide Qt4->Qt5 migration one needed to provide an additional option USE_QT5. This option should be On or Off according to the Qt version, which was used to compile libQGLViewer. On Debian/Ubuntu operating systems libQGLViewer of version 2.6.3 and higher are compiled against Qt5 (for other operating systems refer to the package archive of your distribution), so if you are using such version, please switch on this option. Otherwise, if you mix Qt-versions `Segmentation fault` will appear just after Yade is started. To provide necessary build dependencies for Qt5, install `python-pyqt5 pyqt5-dev-tools `` instead of ``python-qt4 pyqt4-dev-tools`, which is needed for Qt4.

If the compilation is finished without errors, you will see all enabled and disabled options. Then start the standard the compilation process:

```
make
```

The compilation process can take a long time, be patient. An additional parameter on many cores systems `-j` can be added to decrease compilation time and split the compilation on many cores. For example, on 4-core machines it would be reasonable to set the parameter `-j4`. Note, the Yade requires approximately 2GB/core for compilation, otherwise the swap-file will be used and a compilation time dramatically increases.

Installing performs with the following command:

```
make install
```

The "install" command will in fact also recompile if source files have been modified. Hence there is no absolute need to type the two commands separately. You may receive make errors if you don't permission to write into the target folder. These errors are not critical but without writing permissions Yade won't be installed in /usr/local/bin/.

After compilation finished successfully the new built can be started by navigating to /path/to/installfolder/bin and calling yade via (based on version yade-2014-02-20.git-a7048f4):

```
cd /path/to/installfolder/bin
./yade-2014-02-20.git-a7048f4
```

For building the documentation you should at first execute the command "make install" and then "make doc" to build it. The generated files will be stored in your current build directory/doc/sphinx/_build. Once again writing permissions are necessary for installing into /usr/local/share/doc/.

"make manpage" command generates and moves manpages in a standard place. "make check" command executes standard test to check the functionality of compiled program.

Yade can be compiled not only by GCC-compiler, but also by CLANG front-end for the LLVM compiler. For that you set the environment variables CC and CXX upon detecting the C and C++ compiler to use:

```
export CC=/usr/bin/clang
export CXX=/usr/bin/clang++
cmake -DOPTION1=VALUE1 -DOPTION2=VALUE2
```

Clang does not support OpenMP-parallelizing for the moment, that is why the feature will be disabled.

5.3 Yubuntu

If you are not running Ubuntu nor Debian, there is a way to create a Yubuntu live-usb on any usb mass-storage device (minimum recommended size is 5GB). It is a way to make a bootable usb-key with a preinstalled minimalist operating system (Xubuntu), including Yadedaily and Paraview.

More informations about this alternative are available here (see the README file first).

Chapter 6

Yade on GitHub

6.1 Fast checkout without GitHub account (read-only)

Getting the source code without registering on GitHub can be done via a single command. It will not allow interactions with the remote repository, which you access the read-only way:

```
git clone https://github.com/yade/trunk.git
```

6.2 Using branches on GitHub (for frequent commits see git/trunk section below)

Most usefull commands are below. For more details, see for instance http://gitref.org/index.html and https://help.github.com/articles/set-up-git

6.2.1 Setup

1. Register on github.com
2. Add your SSH key to GitHub:

   On the GitHub site Click "Account Settings" (top right) > Click "SSH keys" > Click "Add SSH key"
3. Set your username and email through terminal:

```
git config --global user.name "Firstname Lastname"
git config --global user.email "your_email@youremail.com"
```

4. Fork a repo:

   Click the "Fork" button on the https://github.com/yade/trunk
5. Set Up Your Local Repo through terminal:

```
git clone git@github.com:username/trunk.git
```

   This creates a new folder, named trunk, that contains the whole code.
6. Configure remotes

```
cd to/newly/created/folder
git remote add upstream git@github.com:yade/trunk.git
git fetch upstream
```

Now, your "trunk" folder is linked with the code hosted on github.com. Through appropriate commands explained below, you will be able to update your code to include changes commited by others, or to commit yourself changes that others can get.

6.2.2 Retrieving older Commits

In case you want to work with, or compile, an older version of Yade which is not tagged, you can create your own (local) branch of the corresponding daily build. Look here for details.

6.2.3 Committing and updating

For those used to other version control systems, note that the commit mechanisms in Git significantly differs from that of Bazaar or SVN. Therefore, don't expect to find a one-to-one command replacement. In some cases, however, the equivalent bazaar command is indicated below to ease the transition.

Inspecting changes

You may start by inspecting your changes with a few commands. For the "diff" command, it is convenient to copy from the output of "status" instead of typing the path to modified files.

```
git status
git diff path/to/modified/file.cpp
```

Committing changes

Then you proceed to commit through terminal:

```
git add path/to/new/file.cpp  #Version a newly created file: equivalent of "bzr add"
git commit path/to/new_or_modified/file.cpp -m'Commit message'``  #Validate a change. It can be done several ti
git push  #Push your changes into GitHub. Equivalent of "bzr commit", except that your are commiting to your ow
```

Changes will be pushed to your personal "fork", If you have tested your changes and you are ready to push them into the main trunk, just do a "pull request" [5] or create a patch from your commit via:

```
git format-patch origin  #create patch file in current folder)
```

and send to the developers maililng list (yade-dev@lists.launchpad.net) as attachment. In either way, after reviewing your changes they will be added to the main trunk.

When the pull request has been reviewed and accepted, your changes are integrated in the main trunk. Everyone will get them via `git fetch`.

Updating

You may want to get changes done by others:

```
git fetch upstream  #Pull new updates from the upstream to your branch. Eq. of "bzr update", updating the remot
git merge upstream/master  #Merge upstream changes into your master-branch (eq. of "bzr update", updating your
```

Alternatively, this will do fetch+merge all at once (discouraged if you have uncommited changes):

```
git pull
```

6.3 Working directly on git/trunk (recommended for frequent commits)

This direct access to trunk will sound more familiar to bzr or svn users. It is only possible for members of the git team "developpers". Send an email at yade-dev@lists.launchpad.net to join this team (don't forget to tell your git account name).

- Get trunk:

```
git clone git@github.com:yade/trunk.git
```

  This creates a new folder, named trunk, that contains the whole code.

- Update

```
git pull
```

- Commit to local repository

```
git commit filename1 filename2 ...
```

- Push changes to remote trunk

```
git push
```

  Now, the changes you made are included in the on-line code, and can be get back by every user.

To avoid confusing logs after each commit/pull/push cycle, it is better to setup automatic rebase:

```
git config --global branch.autosetuprebase always
```

Now your file ~/.gitconfig should include:

[**branch**] autosetuprebase = always

Check also .git/config file in your local trunk folder (rebase = true):

[**branch "master"**] remote = origin

merge = refs/heads/master

rebase = true

Auto-rebase may have unpleasant side effects by blocking "pull" if you have uncommited changes. In this case you can use "git stash":

```
git pull
lib/SConscript: needs update
refusing to pull with rebase: your working tree is not up-to-date
git stash #hide the uncommited changes away
git pull  #now it's ok
git push  #push the commited changes
git stash pop #get uncommited changes back
```

6.4 General guidelines for pushing to yade/trunk

1. Set autorebase once on the computer! (see above)
2. Inspect the diff to make sure you will not commit junk code (typically some "cout<<" left here and there), using in terminal:

```
git diff file1
```

   Or, alternatively, any GUI for git: gitg, git-cola...

3. Commit selectively:

```
git commit file1 file2 file3 -m "message" # is good
git commit -a -m "message"                # is bad. It is the best way to commit things that should not be
```

4. Be sure to work with an up-to-date version launching:

```
git pull
```

5. Make sure it compiles and that regression tests pass: try "yade –test" and "yade –check".
6. You can finally let all Yade-users enjoy your work:

```
git push
```

Thanks a lot for your cooperation to Yade!

Chapter 7

DEM Background

In this chapter, we mathematically describe general features of explicit DEM simulations, with some reference to Yade implementation of these algorithms. They are given roughly in the order as they appear in simulation; first, two particles might establish a new interaction, which consists in

1. detecting collision between particles;
2. creating new interaction and determining its properties (such as stiffness); they are either precomputed or derived from properties of both particles;

Then, for already existing interactions, the following is performed:

1. strain evaluation;
2. stress computation based on strains;
3. force application to particles in interaction.

This simplified description serves only to give meaning to the ordering of sections within this chapter. A more detailed description of this *simulation loop* is given later.

In this chapter we refer to kinematic variables of the contacts as ‘‘strains‘‘, although at this scale it is also common to speak of ‘‘displacements‘‘. Which semantic is more appropriate depends on the conceptual model one is starting from, and therefore it cannot be decided independently of specific problems. The reader familiar with displacements can mentaly replace normal strain and shear strain by normal displacement and shear displacement, respectively, without altering the meaning of what follows.

7.1 Collision detection

7.1.1 Generalities

Exact computation of collision configuration between two particles can be relatively expensive (for instance between *Sphere* and *Facet*). Taking a general pair of bodies i and j and their ‘‘exact‘‘ (In the sense of precision admissible by numerical implementation.) spatial predicates (called *Shape* in Yade) represented by point sets P_i, P_j the detection generally proceeds in 2 passes:

1. fast collision detection using approximate predicate $\tilde{P}_i$ and $\tilde{P}_j$; they are pre-constructed in such a way as to abstract away individual features of P_i and P_j and satisfy the condition

$$\forall \mathbf{x} \in \mathbb{R}^3 : x \in P_i \Rightarrow x \in \tilde{P}_i \tag{7.1}$$

(likewise for P_j). The approximate predicate is called ‘‘bounding volume” (*Bound* in Yade) since it bounds any particle's volume from outside (by virtue of the implication). It follows that $(P_i \cap P_j) \neq \emptyset \Rightarrow (\tilde{P}_i \cap \tilde{P}_j) \neq \emptyset$ and, by applying *modus tollens*,

$$\left(\tilde{P}_i \cap \tilde{P}_j\right) = \emptyset \Rightarrow \left(P_i \cap P_j\right) = \emptyset \tag{7.2}$$

which is a candidate exclusion rule in the proper sense.

2. By filtering away impossible collisions in (7.2), a more expensive, exact collision detection algorithms can be run on possible interactions, filtering out remaining spurious couples $(\tilde{P}_i \cap \tilde{P}_j) \neq \emptyset \wedge (P_i \cap P_j) = \emptyset$. These algorithms operate on P_i and P_j and have to be able to handle all possible combinations of shape types.

It is only the first step we are concerned with here.

7.1.2 Algorithms

Collision evaluation algorithms have been the subject of extensive research in fields such as robotics, computer graphics and simulations. They can be roughly divided in two groups:

Hierarchical algorithms which recursively subdivide space and restrict the number of approximate checks in the first pass, knowing that lower-level bounding volumes can intersect only if they are part of the same higher-level bounding volume. Hierarchy elements are bounding volumes of different kinds: octrees *[Jung1997]*, bounding spheres *[Hubbard1996]*, k-DOP's *[Klosowski1998]*.

Flat algorithms work directly with bounding volumes without grouping them in hierarchies first; let us only mention two kinds commonly used in particle simulations:

> **Sweep and prune** algorithm operates on axis-aligned bounding boxes, which overlap if and only if they overlap along all axes. These algorithms have roughly $\mathcal{O}(n \log n)$ complexity, where n is number of particles as long as they exploit *temporal coherence* of the simulation.
>
> **Grid algorithms** represent continuous R^3 space by a finite set of regularly spaced points, leading to very fast neighbor search; they can reach the $\mathcal{O}(n)$ complexity *[Munjiza1998]* and recent research suggests ways to overcome one of the major drawbacks of this method, which is the necessity to adjust grid cell size to the largest particle in the simulation (*[Munjiza2006]*, the "multistep" extension).

Temporal coherence expresses the fact that motion of particles in simulation is not arbitrary but governed by physical laws. This knowledge can be exploited to optimize performance.

Numerical stability of integrating motion equations dictates an upper limit on Δt (sect. *Stability considerations*) and, by consequence, on displacement of particles during one step. This consideration is taken into account in *[Munjiza2006]*, implying that any particle may not move further than to a neighboring grid cell during one step allowing the $\mathcal{O}(n)$ complexity; it is also explored in the periodic variant of the sweep and prune algorithm described below.

On a finer level, it is common to enlarge $\tilde{P}_i$ predicates in such a way that they satisfy the (7.1) condition during *several* timesteps; the first collision detection pass might then be run with stride, speeding up the simulation considerably. The original publication of this optimization by Verlet *[Verlet1967]* used enlarged list of neighbors, giving this technique the name *Verlet list*. In general cases, however, where neighbor lists are not necessarily used, the term *Verlet distance* is employed.

7.1.3 Sweep and prune

Let us describe in detail the sweep and prune algorithm used for collision detection in Yade (class *InsertionSortCollider*). Axis-aligned bounding boxes (*Aabb*) are used as $\tilde{P}_i$; each *Aabb* is given by lower and upper corner $\in R^3$ (in the following, $\tilde{P}_i^{x0}$, $\tilde{P}_i^{x1}$ are minimum/maximum coordinates of $\tilde{P}_i$ along the x-axis and so on). Construction of *Aabb* from various particle *Shape*'s (such as *Sphere*, *Facet*, *Wall*) is straightforward, handled by appropriate classes deriving form *BoundFunctor* (*Bo1_Sphere_Aabb*, *Bo1_Facet_Aabb*, ...).

Presence of overlap of two *Aabb*'s can be determined from conjunction of separate overlaps of intervals along each axis (*fig-sweep-and-prune*):

$$\left(\tilde{P}_i \cap \tilde{P}_j\right) \neq \emptyset \Leftrightarrow \bigwedge_{w \in \{x,y,z\}} \left[\left(\left(\tilde{P}_i^{w0}, \tilde{P}_i^{w1}\right) \cap \left(\tilde{P}_j^{w0}, \tilde{P}_j^{w1}\right)\right) \neq \emptyset\right]$$

where (a, b) denotes interval in R.

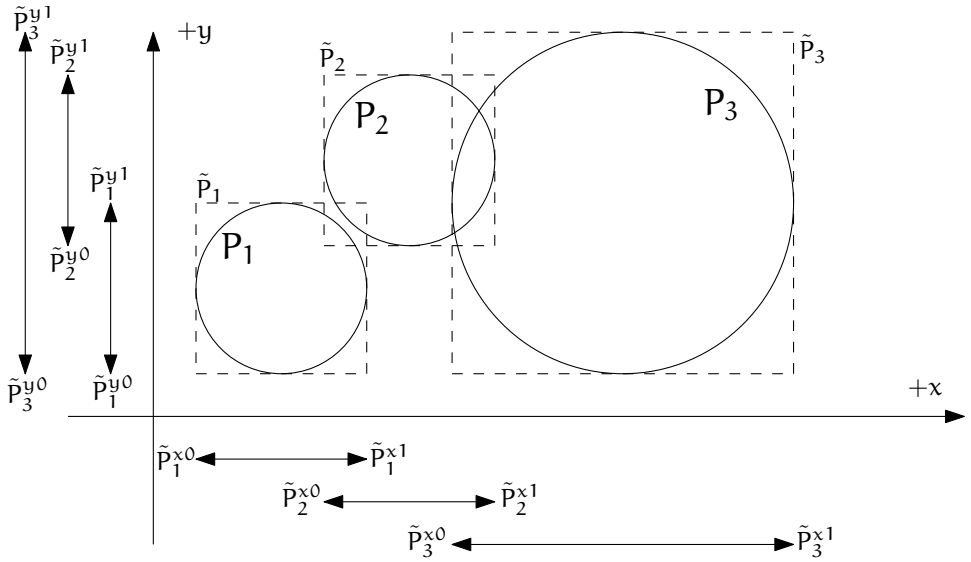


Fig. 7.1: Sweep and prune algorithm (shown in 2D), where *Aabb* of each sphere is represented by minimum and maximum value along each axis. Spatial overlap of *Aabb*'s is present if they overlap along all axes. In this case, $\tilde{P}_1 \cap \tilde{P}_2 \neq \emptyset$ (but note that $P_1 \cap P_2 = \emptyset$) and $\tilde{P}_2 \cap \tilde{P}_3 \neq \emptyset$.}

The collider keeps 3 separate lists (arrays) L_w for each axis $w \in \{x, y, z\}$

$$L_w = \bigcup_i \left\{ \tilde{P}_i^{w0}, \tilde{P}_i^{w1} \right\}$$

where i traverses all particles. L_w arrays (sorted sets) contain respective coordinates of minimum and maximum corners for each *Aabb* (we call these coordinates *bound* in the following); besides bound, each of list elements further carries `id` referring to particle it belongs to, and a flag whether it is lower or upper bound.

In the initial step, all lists are sorted (using quicksort, average $\mathcal{O}(n \log n)$) and one axis is used to create initial interactions: the range between lower and upper bound for each body is traversed, while bounds in-between indicate potential *Aabb* overlaps which must be checked on the remaining axes as well.

At each successive step, lists are already pre-sorted. Inversions occur where a particle's coordinate has just crossed another particle's coordinate; this number is limited by numerical stability of simulation and its physical meaning (giving spatio-temporal coherence to the algorithm). The insertion sort algorithm swaps neighboring elements if they are inverted, and has complexity between bigO{n} and bigO{n^2}, for pre-sorted and unsorted lists respectively. For our purposes, we need only to handle inversions, which by nature of the sort algorithm are detected inside the sort loop. An inversion might signify:

- overlap along the current axis, if an upper bound inverts (swaps) with a lower bound (i.e. that the upper bound with a higher coordinate was out of order in coming before the lower bound with a lower coordinate). Overlap along the other 2 axes is checked and if there is overlap along all axes, a new potential interaction is created.
- End of overlap along the current axis, if lower bound inverts (swaps) with an upper bound. If there is only potential interaction between the two particles in question, it is deleted.
- Nothing if both bounds are upper or both lower.

Aperiodic insertion sort

Let us show the sort algorithm on a sample sequence of numbers:

$$\| \quad 3 \qquad 7 \qquad 2 \qquad 4 \quad \|$$

Elements are traversed from left to right; each of them keeps inverting (swapping) with neighbors to the left, moving left itself, until any of the following conditions is satisfied:

($\leq$)	the sorting order with the left neighbor is correct, or
($\|\|$)	the element is at the beginning of the sequence.

We start at the leftmost element (the current element is marked $\boxed{\mathtt{i}}$)

$$\| \quad \boxed{3} \quad 7 \quad 2 \quad 4 \quad \|.$$

It obviously immediately satisfies ($\|\|$), and we move to the next element:

$$\| \quad 3 \overset{\leq}{\leftarrow} \boxed{7} \quad 2 \quad 4 \quad \|.$$

Condition ($\leq$) holds, therefore we move to the right. The $\boxed{2}$ is not in order (violating ($\leq$)) and two inversions take place; after that, ($\|\|$) holds:

$$\begin{aligned}
&\| \quad 3 \quad 7 \overset{\not\leq}{\leftarrow} \boxed{2} \quad 4 \quad \|,\\
&\| \quad 3 \overset{\not\leq}{\leftarrow} \boxed{2} \quad 7 \quad 4 \quad \|,\\
&\| \quad \boxed{2} \quad 3 \quad 7 \quad 4 \quad \|.
\end{aligned}$$

The last element $\boxed{4}$ first violates ($\leq$), but satisfies it after one inversion

$$\begin{aligned}
&\| \quad 2 \quad 3 \quad 7 \overset{\not\leq}{\leftarrow} \boxed{4} \quad \|,\\
&\| \quad 2 \quad 3 \overset{\leq}{\leftarrow} \boxed{4} \quad 7 \quad \|.
\end{aligned}$$

All elements having been traversed, the sequence is now sorted.

It is obvious that if the initial sequence were sorted, elements only would have to be traversed without any inversion to handle (that happens in $\mathcal{O}(\mathtt{n})$ time).

For each inversion during the sort in simulation, the function that investigates change in *Aabb* overlap is invoked, creating or deleting interactions.

The periodic variant of the sort algorithm is described in *Periodic insertion sort algorithm*, along with other periodic-boundary related topics.

Optimization with Verlet distances

As noted above, *[Verlet1967]* explored the possibility of running the collision detection only sparsely by enlarging predicates $\tilde{\mathtt{P}}_\mathtt{i}$.

In Yade, this is achieved by enlarging *Aabb* of particles by fixed relative length (or Verlet's distance) in all dimensions $\Delta\mathtt{L}$ (*InsertionSortCollider.sweepLength*). Suppose the collider run last time at step $\mathtt{m}$ and the current step is $\mathtt{n}$. *NewtonIntegrator* tracks the cummulated distance traversed by each particle between $\mathtt{m}$ and $\mathtt{n}$ by comparing the current position with the reference position from time $\mathtt{n}$ (*Bound::refPos*),

$$\mathtt{L}_{\mathtt{mn}} = |\mathtt{X}^{\mathtt{n}} - \mathtt{X}^{\mathtt{m}}| \tag{7.3}$$

triggering the collider re-run as soon as one particle gives:

$$\mathtt{L}_{\mathtt{mn}} > \Delta\mathtt{L}. \tag{7.4}$$

InsertionSortCollider.targetInterv is used to adjust $\Delta\mathtt{L}$ independently for each particle. Larger $\Delta\mathtt{L}$ will be assigned to the fastest ones, so that all particles would ideally reach the edge of their bounds after this "target" number of iterations. Results of using Verlet distance depend highly on the nature of simulation and choice of *InsertionSortCollider.targetInterv*. Adjusting the sizes independently for each particle is especially efficient if some parts of a problem have high-speed particles will others are not moving. If

it is not the case, no significant gain should be expected as compared to targetInterv=0 (assigning the same ΔL to all particles).

The number of particles and the number of available threads is also to be considered for choosing an appropriate Verlet's distance. A larger distance will result in less time spent in the collider (which runs single-threaded) and more time in computing interactions (multi-threaded). Typically, large ΔL will be used for large simulations with more than 10^5 particles on multi-core computers. On the other hand simulations with less than 10^4 particles on single processor will probably benefit from smaller ΔL. Users benchmarks may be found on Yade's wiki (see e.g. https://yade-dem.org/wiki/Colliders_performace).

7.2 Creating interaction between particles

Collision detection described above is only approximate. Exact collision detection depends on the geometry of individual particles and is handled separately. In Yade terminology, the *Collider* creates only *potential* interactions; potential interactions are evaluated exactly using specialized algorithms for collision of two spheres or other combinations. Exact collision detection must be run at every timestep since it is at every step that particles can change their mutual position (the collider is only run sometimes if the Verlet distance optimization is in use). Some exact collision detection algorithms are described in *Strain evaluation*; in Yade, they are implemented in classes deriving from *IGeomFunctor* (prefixed with `Ig2`).

Besides detection of geometrical overlap (which corresponds to *IGeom* in Yade), there are also non-geometrical properties of the interaction to be determined (*IPhys*). In Yade, they are computed for every new interaction by calling a functor deriving from *IPhysFunctor* (prefixed with `Ip2`) which accepts the given combination of *Material* types of both particles.

7.2.1 Stiffnesses

Basic DEM interaction defines two stiffnesses: normal stiffness K_N and shear (tangent) stiffness K_T. It is desirable that K_N be related to fictitious Young's modulus of the particles' material, while K_T is typically determined as a given fraction of computed K_N. The K_T/K_N ratio determines macroscopic Poisson's ratio of the arrangement, which can be shown by dimensional analysis: elastic continuum has two parameters (E and ν) and basic DEM model also has 2 parameters with the same dimensions K_N and K_T/K_N; macroscopic Poisson's ratio is therefore determined solely by K_T/K_N and macroscopic Young's modulus is then proportional to K_N and affected by K_T/K_N.

Naturally, such analysis is highly simplifying and does not account for particle radius distribution, packing configuration and other possible parameters such as the interaction radius introduced later.

Normal stiffness

The algorithm commonly used in Yade computes normal interaction stiffness as stiffness of two springs in serial configuration with lengths equal to the sphere radii (*fig-spheres-contact-stiffness*).

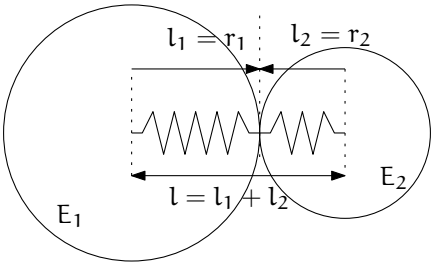


Fig. 7.2: Series of 2 springs representing normal stiffness of contact between 2 spheres.

Let us define distance $l = l_1 + l_2$, where l_i are distances between contact point and sphere centers, which are initially (roughly speaking) equal to sphere radii. Change of distance between the sphere centers Δl

is distributed onto deformations of both spheres $\Delta l = \Delta l_1 + \Delta l_2$ proportionally to their compliances. Displacement change Δl_i generates force $F_i = K_i \Delta l_i$, where K_i assures proportionality and has physical meaning and dimension of stiffness; K_i is related to the sphere material modulus E_i and some length $\tilde{l}_i$ proportional to r_i.

$$\begin{aligned}
\Delta l &= \Delta l_1 + \Delta l_2 \\
K_i &= E_i \tilde{l}_i \\
K_N \Delta l &= F = F_1 = F_2 \\
K_N \left(\Delta l_1 + \Delta l_2\right) &= F \\
K_N \left(\frac{F}{K_1} + \frac{F}{K_2}\right) &= F \\
K_1^{-1} + K_2^{-1} &= K_N^{-1} \\
K_N &= \frac{K_1 K_2}{K_1 + K_2} \\
K_N &= \frac{E_1 \tilde{l}_1 E_2 \tilde{l}_2}{E_1 \tilde{l}_1 + E_2 \tilde{l}_2}
\end{aligned}$$

The most used class computing interaction properties *Ip2_FrictMat_FrictMat_FrictPhys* uses $\tilde{l}_i = 2r_i$.

Some formulations define an equivalent cross-section A_{eq}, which in that case appears in the $\tilde{l}_i$ term as $K_i = E_i \tilde{l}_i = E_i \frac{A_{eq}}{l_i}$. Such is the case for the concrete model (*Ip2_CpmMat_CpmMat_CpmPhys*), where $A_{eq} = \min(r_1, r_2)$.

For reasons given above, no pretense about equality of particle-level E_i and macroscopic modulus E should be made. Some formulations, such as *[Hentz2003]*, introduce parameters to match them numerically. This is not appropriate, in our opinion, since it binds those values to particular features of the sphere arrangement that was used for calibration.

7.2.2 Other parameters

Non-elastic parameters differ for various material models. Usually, though, they are averaged from the particles' material properties, if it makes sense. For instance, *Ip2_CpmMat_CpmMat_CpmPhys* averages most quantities, while *Ip2_FrictMat_FrictMat_FrictPhys* computes internal friction angle as $\varphi = \min(\varphi_1, \varphi_2)$ to avoid friction with bodies that are frictionless.

7.3 Strain evaluation

In the general case, mutual configuration of two particles has 6 degrees of freedom (DoFs) just like a beam in 3D space: both particles have 6 DoFs each, but the interaction itself is free to move and rotate in space (with both spheres) having 6 DoFs itself; then $12 - 6 = 6$. They are shown at *fig-spheres-dofs*.

We will only describe normal and shear components of strain in the following, leaving torsion and bending aside. The reason is that most constitutive laws for contacts do not use the latter two.

7.3.1 Normal strain

Constants

Let us consider two spheres with *initial* centers $\bar{\mathbf{C}}_1$, $\bar{\mathbf{C}}_2$ and radii r_1, r_2 that enter into contact. The order of spheres within the contact is arbitrary and has no influence on the behavior. Then we define lengths

$$d_0 = |\bar{\mathbf{C}}_2 - \bar{\mathbf{C}}_1|$$

$$d_1 = r_1 + \frac{d_0 - r_1 - r_2}{2}, \qquad d_2 = d_0 - d_1.$$

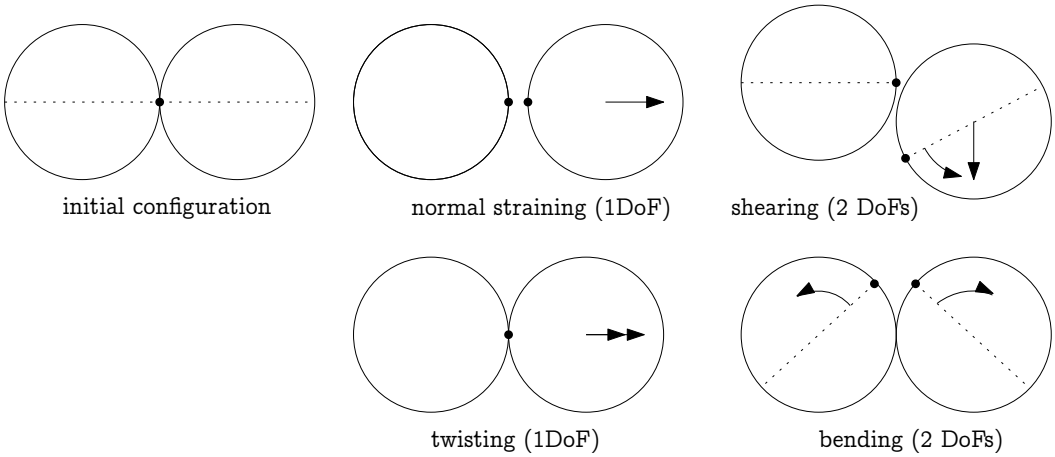


Fig. 7.3: Degrees of freedom of configuration of two spheres. Normal strain appears if there is a difference of linear velocity along the interaction axis ($\mathbf{n}$); shearing originates from the difference of linear velocities perpendicular to $\mathbf{n}$ *and* from the part of $\boldsymbol{\omega}_1 + \boldsymbol{\omega}_2$ perpendicular to $\mathbf{n}$; twisting is caused by the part of $\boldsymbol{\omega}_1 - \boldsymbol{\omega}_2$ parallel with $\mathbf{n}$; bending comes from the part of $\boldsymbol{\omega}_1 - \boldsymbol{\omega}_2$ perpendicular to $\mathbf{n}$.

These quantities are *constant* throughout the life of the interaction and are computed only once when the interaction is established. The distance d_0 is the *reference distance* and is used for the conversion of absolute displacements to dimensionless strain, for instance. It is also the distance where (for usual contact laws) there is neither repulsive nor attractive force between the spheres, whence the name *equilibrium distance*.

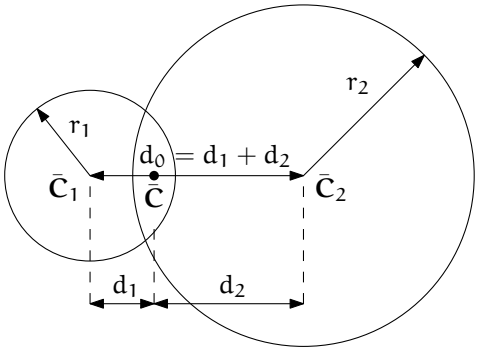


Fig. 7.4: Geometry of the initial contact of 2 spheres; this case pictures spheres which already overlap when the contact is created (which can be the case at the beginning of a simulation) for the sake of generality. The initial contact point $\bar{\mathbf{C}}$ is in the middle of the overlap zone.

Distances d_1 and d_2 define reduced (or expanded) radii of spheres; geometrical radii r_1 and r_2 are used only for collision detection and may not be the same as d_1 and d_2, as shown in fig. *fig-sphere-sphere*. This difference is exploited in cases where the average number of contacts between spheres should be increased, e.g. to influence the response in compression or to stabilize the packing. In such case, interactions will be created also for spheres that do not geometrically overlap based on the *interaction radius* R_I, a dimensionless parameter determining „non-locality“ of contact detection. For $R_I = 1$, only spheres that touch are considered in contact; the general condition reads

$$d_0 \leq R_I(r_1 + r_2). \tag{7.5}$$

The value of R_I directly influences the average number of interactions per sphere (percolation), which for some models is necessary in order to achieve realistic results. In such cases, *Aabb* (or $\tilde{P}_i$ predicates in general) must be enlarged accordingly (*Bo1_Sphere_Aabb.aabbEnlargeFactor*).

Contact cross-section

Some constitutive laws are formulated with strains and stresses (*Law2_ScGeom_CpmPhys_Cpm*, the concrete model described later, for instance); in that case, equivalent cross-section of the contact must be introduced for the sake of dimensionality. The exact definition is rather arbitrary; the CPM model (*Ip2_CpmMat_CpmMat_CpmPhys*) uses the relation

$$A_{eq} = \pi \min(r_1, r_2)^2 \tag{7.6}$$

which will be used to convert stresses to forces, if the constitutive law used is formulated in terms of stresses and strains. Note that other values than π can be used; it will merely scale macroscopic packing stiffness; it is only for the intuitive notion of a truss-like element between the particle centers that we choose A_{eq} representing the circle area. Besides that, another function than $\min(r_1, r_2)$ can be used, although the result should depend linearly on r_1 and r_2 so that the equation gives consistent results if the particle dimensions are scaled.

Variables

The following state variables are updated as spheres undergo motion during the simulation (as $\mathbf{C}_1^\circ$ and $\mathbf{C}_2^\circ$ change):

$$\mathbf{n}^\circ = \frac{\mathbf{C}_2^\circ - \mathbf{C}_1^\circ}{|\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|} \equiv \widehat{\mathbf{C}_2^\circ - \mathbf{C}_1^\circ} \tag{7.7}$$

and

$$\mathbf{C}^\circ = \mathbf{C}_1^\circ + \left(d_1 - \frac{d_0 - |\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|}{2} \right) \mathbf{n}. \tag{7.8}$$

The contact point $\mathbf{C}^\circ$ is always in the middle of the spheres' overlap zone (even if the overlap is negative, when it is in the middle of the empty space between the spheres). The *contact plane* is always perpendicular to the contact plane normal $\mathbf{n}^\circ$ and passes through $\mathbf{C}^\circ$.

Normal displacement and strain can be defined as

$$u_N = |\mathbf{C}_2^\circ - \mathbf{C}_1^\circ| - d_0,$$
$$\varepsilon_N = \frac{u_N}{d_0} = \frac{|\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|}{d_0} - 1.$$

Since u_N is always aligned with $\mathbf{n}$, it can be stored as a scalar value multiplied by $\mathbf{n}$ if necessary.

For massively compressive simulations, it might be beneficial to use the logarithmic strain, such that the strain tends to $-\infty$ (rather than -1) as centers of both spheres approach. Otherwise, repulsive force would remain finite and the spheres could penetrate through each other. Therefore, we can adjust the definition of normal strain as follows:

$$\varepsilon_N = \begin{cases} \log\left(\frac{|\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|}{d_0}\right) & \text{if } |\mathbf{C}_2^\circ - \mathbf{C}_1^\circ| < d_0 \\ \frac{|\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|}{d_0} - 1 & \text{otherwise.} \end{cases}$$

Such definition, however, has the disadvantage of effectively increasing rigidity (up to infinity) of contacts, requiring Δt to be adjusted, lest the simulation becomes unstable. Such dynamic adjustment is possible using a stiffness-based time-stepper (*GlobalStiffnessTimeStepper* in Yade).

7.3.2 Shear strain

In order to keep $\mathbf{u}_T$ consistent (e.g. that $\mathbf{u}_T$ must be constant if two spheres retain mutually constant configuration but move arbitrarily in space), then either $\mathbf{u}_T$ must track spheres' spatial motion or must (somehow) rely on sphere-local data exclusively.

Geometrical meaning of shear strain is shown in *fig-shear-2d*.

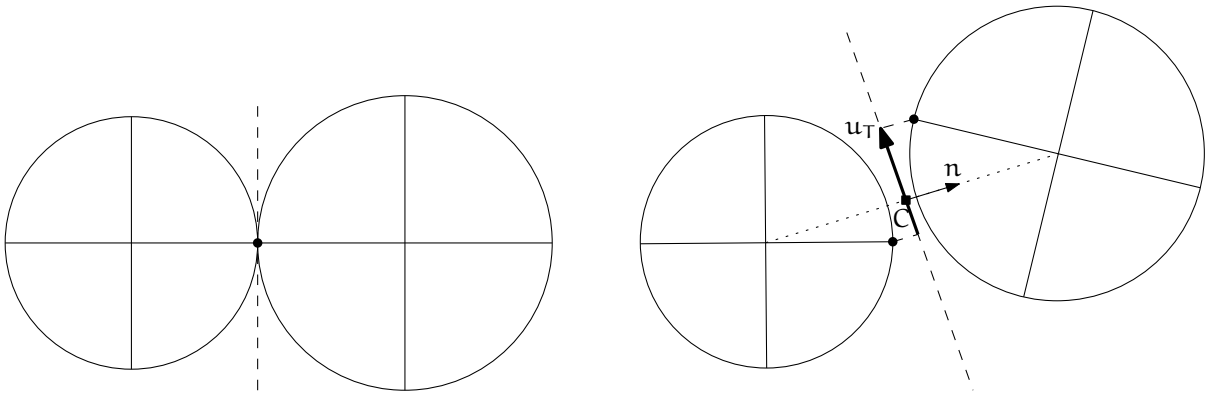


Fig. 7.5: Evolution of shear displacement $\mathbf{u}_T$ due to mutual motion of spheres, both linear and rotational. Left configuration is the initial contact, right configuration is after displacement and rotation of one particle.

The classical incremental algorithm is widely used in DEM codes and is described frequently (*[Luding2008]*, *[Alonso2004]*). Yade implements this algorithm in the *ScGeom* class. At each step, shear displacement $\mathbf{u}_T$ is updated; the update increment can be decomposed in 2 parts: motion of the interaction (i.e. $\mathbf{C}$ and $\mathbf{n}$) in global space and mutual motion of spheres.

1. Contact moves dues to changes of the spheres' positions $\mathbf{C}_1$ and $\mathbf{C}_2$, which updates current $\mathbf{C}^\circ$ and $\mathbf{n}^\circ$ as per (7.8) and (7.7). $\mathbf{u}_T^-$ is perpendicular to the contact plane at the previous step $\mathbf{n}^-$ and must be updated so that $\mathbf{u}_T^- + (\Delta\mathbf{u}_T) = \mathbf{u}_T^\circ \perp \mathbf{n}^\circ$; this is done by perpendicular projection to the plane first (which might decrease $|\mathbf{u}_T|$) and adding what corresponds to spatial rotation of the interaction instead:

$$(\Delta\mathbf{u}_T)_1 = -\mathbf{u}_T^- \times (\mathbf{n}^- \times \mathbf{n}^\circ)$$
$$(\Delta\mathbf{u}_T)_2 = -\mathbf{u}_T^- \times \left(\frac{\Delta t}{2}\mathbf{n}^\circ \cdot (\boldsymbol{\omega}_1^{\ominus} + \boldsymbol{\omega}_2^{\ominus})\right)\mathbf{n}^\circ$$

2. Mutual movement of spheres, using only its part perpendicular to $\mathbf{n}^\circ$; $\boldsymbol{v}_{12}$ denotes mutual velocity of spheres at the contact point:

$$\boldsymbol{v}_{12} = \left(\boldsymbol{v}_2^{\ominus} + \boldsymbol{\omega}_2^- \times (-d_2\mathbf{n}^\circ)\right) - \left(\boldsymbol{v}_1^{\ominus} + \boldsymbol{\omega}_1^{\ominus} \times (d_1\mathbf{n}^\circ)\right)$$
$$\boldsymbol{v}_{12}^\perp = \boldsymbol{v}_{12} - (\mathbf{n}^\circ \cdot \boldsymbol{v}_{12})\mathbf{n}^\circ$$
$$(\Delta\mathbf{u}_T)_3 = -\Delta t \boldsymbol{v}_{12}^\perp$$

Finally, we compute

$$\mathbf{u}_T^\circ = \mathbf{u}_T^- + (\Delta\mathbf{u}_T)_1 + (\Delta\mathbf{u}_T)_2 + (\Delta\mathbf{u}_T)_3.$$

7.4 Stress evaluation (example)

Once strain on a contact is computed, it can be used to compute stresses/forces acting on both spheres.

The constitutive law presented here is the most usual DEM formulation, originally proposed by Cundall. While the strain evaluation will be similar to algorithms described in the previous section regardless of stress evaluation, stress evaluation itself depends on the nature of the material being modeled. The constitutive law presented here is the most simple non-cohesive elastic case with dry friction, which Yade implements in *Law2_ScGeom_FrictPhys_CundallStrack* (all constitutive laws derive from base class *LawFunctor*).

In DEM generally, some constitutive laws are expressed using strains and stresses while others prefer displacement/force formulation. The law described here falls in the latter category.

When new contact is established (discussed in *Engines*) it has its properties (*IPhys*) computed from *Materials* associated with both particles. In the simple case of frictional material *FrictMat*, *Ip2_FrictMat_FrictMat_FrictPhys* creates a new *FrictPhys* instance, which defines normal stiffness K_N, shear stiffness K_T and friction angle φ.

At each step, given normal and shear displacements u_N, $\mathbf{u}_T$, normal and shear forces are computed (if $u_N > 0$, the contact is deleted without generating any forces):

$$\begin{aligned}\mathbf{F}_N &= K_N u_N \mathbf{n},\\ \mathbf{F}_T^t &= K_T \mathbf{u}_T\end{aligned}$$

where $\mathbf{F}_N$ is normal force and $\mathbf{F}_T$ is trial shear force. A simple non-associated stress return algorithm is applied to compute final shear force

$$\mathbf{F}_T = \begin{cases}\mathbf{F}_T^t \frac{|\mathbf{F}_N| \tan\varphi}{\mathbf{F}_T^t} & \text{if } |\mathbf{F}_T| > |\mathbf{F}_N| \tan\varphi,\\ \mathbf{F}_T^t & \text{otherwise.}\end{cases}$$

Summary force $\mathbf{F} = \mathbf{F}_N + \mathbf{F}_T$ is then applied to both particles – each particle accumulates forces and torques acting on it in the course of each step. Because the force computed acts at contact point $\mathbf{C}$, which is difference from spheres' centers, torque generated by $\mathbf{F}$ must also be considered.

$$\begin{aligned}\mathbf{F}_1 &+= \mathbf{F} & \mathbf{F}_2 &+= -\mathbf{F}\\ \mathbf{T}_1 &+= d_1(-\mathbf{n}) \times \mathbf{F} & \mathbf{T}_2 &+= d_2 \mathbf{n} \times \mathbf{F}.\end{aligned}$$

7.5 Motion integration

Each particle accumulates generalized forces (forces and torques) from the contacts in which it participates. These generalized forces are then used to integrate motion equations for each particle separately; therefore, we omit i indices denoting the i-th particle in this section.

The customary leapfrog scheme (also known as the Verlet scheme) is used, with some adjustments for rotation of non-spherical particles, as explained below. The "leapfrog" name comes from the fact that even derivatives of position/orientation are known at on-step points, whereas odd derivatives are known at mid-step points. Let us recall that we use a^-, a°, a^+ for on-step values of a at $t - \Delta t$, t and $t + \Delta t$ respectively; and $a^\ominus$, $a^\oplus$ for mid-step values of a at $t - \Delta t/2$, $t + \Delta t/2$.

Described integration algorithms are implemented in the *NewtonIntegrator* class in Yade.

7.5.1 Position

Integrating motion consists in using current acceleration $\ddot{\mathbf{u}}^\circ$ on a particle to update its position from the current value $\mathbf{u}^\circ$ to its value at the next timestep $\mathbf{u}^+$. Computation of acceleration, knowing current forces $\mathbf{F}$ acting on the particle in question and its mass m, is simply

$$\ddot{\mathbf{u}}^\circ = \mathbf{F}/m.$$

Using the 2nd order finite difference with step Δt, we obtain

$$\ddot{\mathbf{u}}^\circ \cong \frac{\mathbf{u}^- - 2\mathbf{u}^\circ + \mathbf{u}^+}{\Delta t^2}$$

from which we express

$$\begin{aligned}\mathbf{u}^+ &= 2\mathbf{u}^\circ - \mathbf{u}^- + \ddot{\mathbf{u}}^\circ \Delta t^2 =\\ &= \mathbf{u}^\circ + \Delta t \underbrace{\left(\frac{\mathbf{u}^\circ - \mathbf{u}^-}{\Delta t} + \ddot{\mathbf{u}}^\circ \Delta t\right)}_{(\dagger)}.\end{aligned}$$

Typically, $\mathbf{u}^{-}$ is already not known (only $\mathbf{u}^{\circ}$ is); we notice, however, that

$$\dot{\mathbf{u}}^{\ominus} \simeq \frac{\mathbf{u}^{\circ} - \mathbf{u}^{-}}{\Delta t},$$

i.e. the mean velocity during the previous step, which is known. Plugging this approximate into the (†) term, we also notice that mean velocity during the current step can be approximated as

$$\dot{\mathbf{u}}^{\oplus} \simeq \dot{\mathbf{u}}^{\ominus} + \ddot{\mathbf{u}}^{\circ}\Delta t,$$

which is (†); we arrive finally at

$$\mathbf{u}^{+} = \mathbf{u}^{\circ} + \Delta t \left(\dot{\mathbf{u}}^{\ominus} + \ddot{\mathbf{u}}^{\circ}\Delta t\right).$$

The algorithm can then be written down by first computing current mean velocity $\dot{\mathbf{u}}^{\oplus}$ which we need to store for the next step (just as we use its old value $\dot{\mathbf{u}}^{\ominus}$ now), then computing the position for the next time step $\mathbf{u}^{+}$:

$$\dot{\mathbf{u}}^{\oplus} = \dot{\mathbf{u}}^{\ominus} + \ddot{\mathbf{u}}^{\circ}\Delta t$$
$$\mathbf{u}^{+} = \mathbf{u}^{\circ} + \dot{\mathbf{u}}^{\oplus}\Delta t.$$

Positions are known at times $i\Delta t$ (if Δt is constant) while velocities are known at $i\Delta t + \frac{\Delta t}{2}$. The facet that they interleave (jump over each other) in such way gave rise to the colloquial name "leapfrog" scheme.

7.5.2 Orientation (spherical)

Updating particle orientation q° proceeds in an analogous way to position update. First, we compute current angular acceleration $\dot{\boldsymbol{\omega}}^{\circ}$ from known current torque $\mathbf{T}$. For spherical particles where the inertia tensor is diagonal in any orientation (therefore also in current global orientation), satisfying $\mathbf{I}_{11} = \mathbf{I}_{22} = \mathbf{I}_{33}$, we can write

$$\dot{\boldsymbol{\omega}}^{\circ}_{i} = \mathbf{T}_{i}/\mathbf{I}_{11},$$

We use the same approximation scheme, obtaining an equation analogous to (**??**)

$$\boldsymbol{\omega}^{\oplus} = \boldsymbol{\omega}^{\ominus} + \Delta t\dot{\boldsymbol{\omega}}^{\circ}.$$

The quaternion Δq representing rotation vector $\boldsymbol{\omega}^{\oplus}\Delta t$ is constructed, i.e. such that

$$(\Delta q)_{\vartheta} = |\boldsymbol{\omega}^{\oplus}|,$$
$$(\Delta q)_{\mathbf{u}} = \widehat{\boldsymbol{\omega}^{\oplus}}$$

Finally, we compute the next orientation q^{+} by rotation composition

$$q^{+} = \Delta q q^{\circ}.$$

7.5.3 Orientation (aspherical)

Integrating rotation of aspherical particles is considerably more complicated than their position, as their local reference frame is not inertial. Rotation of rigid body in the local frame, where inertia matrix $\mathbf{I}$ is diagonal, is described in the continuous form by Euler's equations ($i \in \{1,2,3\}$ and i, j, k are subsequent indices):

$$\mathbf{T}_{i} = \mathbf{I}_{ii}\dot{\boldsymbol{\omega}}_{i} + (\mathbf{I}_{kk} - \mathbf{I}_{jj})\boldsymbol{\omega}_{j}\boldsymbol{\omega}_{k}.$$

Due to the presence of the current values of both $\boldsymbol{\omega}$ and $\dot{\boldsymbol{\omega}}$, they cannot be solved using the standard leapfrog algorithm (that was the case for translational motion and also for the spherical bodies' rotation where this equation reduced to $\mathbf{T} = \mathbf{I}\dot{\boldsymbol{\omega}}$).

The algorithm presented here is described by *[Allen1989]* (pg. 84–89) and was designed by Fincham for molecular dynamics problems; it is based on extending the leapfrog algorithm by mid-step/on-step

estimators of quantities known at on-step/mid-step points in the basic formulation. Although it has received criticism and more precise algorithms are known (*[Omelyan1999]*, *[Neto2006]*, *[Johnson2008]*), this one is currently implemented in Yade for its relative simplicity.

Each body has its local coordinate system based on the principal axes of inertia for that body. We use $\widetilde{\bullet}$ to denote vectors in local coordinates. The orientation of the local system is given by the current particle's orientation q° as a quaternion; this quaternion can be expressed as the (current) rotation matrix $\boldsymbol{A}$. Therefore, every vector $\boldsymbol{a}$ is transformed as $\widetilde{\boldsymbol{a}} = q\boldsymbol{a}q^* = \boldsymbol{A}\boldsymbol{a}$. Since $\boldsymbol{A}$ is a rotation (orthogonal) matrix, the inverse rotation $\boldsymbol{A}^{-1} = \boldsymbol{A}^T$.

For given particle in question, we know

- $\widetilde{\boldsymbol{I}}^\circ$ (constant) inertia matrix; diagonal, since in local, principal coordinates,
- $\boldsymbol{T}^\circ$ external torque,
- q° current orientation (and its equivalent rotation matrix $\boldsymbol{A}$),
- $\boldsymbol{\omega}^\ominus$ mid-step angular velocity,
- $\boldsymbol{L}^\ominus$ mid-step angular momentum; this is an auxiliary variable that must be tracked in addition for use in this algorithm. It will be zero in the initial step.

Our goal is to compute new values of the latter three, that is $\boldsymbol{L}^\oplus$, q^+, $\boldsymbol{\omega}^\oplus$. We first estimate current angular momentum and compute current local angular velocity:

$$\begin{aligned}
\boldsymbol{L}^\circ &= \boldsymbol{L}^\ominus + \boldsymbol{T}^\circ\frac{\Delta t}{2}, & \widetilde{\boldsymbol{L}}^\circ &= \boldsymbol{A}\boldsymbol{L}^\circ,\\
\boldsymbol{L}^\oplus &= \boldsymbol{L}^\ominus + \boldsymbol{T}^\circ\Delta t, & \widetilde{\boldsymbol{L}}^\oplus &= \boldsymbol{A}\boldsymbol{L}^\oplus,\\
\widetilde{\boldsymbol{\omega}}^\circ &= \widetilde{\boldsymbol{I}}^{\circ\,-1}\widetilde{\boldsymbol{L}}^\circ,\\
\widetilde{\boldsymbol{\omega}}^\oplus &= \widetilde{\boldsymbol{I}}^{\circ\,-1}\widetilde{\boldsymbol{L}}^\oplus.
\end{aligned}$$

Then we compute $\dot{q}^\circ$, using q° and $\widetilde{\boldsymbol{\omega}}^\circ$:

$$\begin{pmatrix}\dot{q}^\circ_w\\ \dot{q}^\circ_x\\ \dot{q}^\circ_y\\ \dot{q}^\circ_z\end{pmatrix} = \frac{1}{2}\begin{pmatrix}q^\circ_w & -q^\circ_x & -q^\circ_y & -q^\circ_z\\ q^\circ_x & q^\circ_w & -q^\circ_z & q^\circ_y\\ q^\circ_y & q^\circ_z & q^\circ_w & -q^\circ_x\\ q^\circ_z & -q^\circ_y & q^\circ_x & q^\circ_w\end{pmatrix}\begin{pmatrix}0\\ \widetilde{\omega}^\circ_x\\ \widetilde{\omega}^\circ_y\\ \widetilde{\omega}^\circ_z\end{pmatrix},$$
$$q^\oplus = q^\circ + \dot{q}^\circ\frac{\Delta t}{2}.$$

We evaluate $\dot{q}^\oplus$ from $q^\oplus$ and $\widetilde{\boldsymbol{\omega}}^\oplus$ in the same way as in (**??**) but shifted by $\Delta t/2$ ahead. Then we can finally compute the desired values

$$\begin{aligned}
q^+ &= q^\circ + \dot{q}^\oplus\Delta t,\\
\boldsymbol{\omega}^\oplus &= \boldsymbol{A}^{-1}\widetilde{\boldsymbol{\omega}}^\oplus
\end{aligned}$$

7.5.4 Clumps (rigid aggregates)

DEM simulations frequently make use of rigid aggregates of particles to model complex shapes *[Price2007]* called *clumps*, typically composed of many spheres. Dynamic properties of clumps are computed from the properties of its members:

- For non-overlapping clump members the clump's mass m_c is summed over members, the inertia tensor $\boldsymbol{I}_c$ is computed using the parallel axes theorem: $\boldsymbol{I}_c = \sum_i (m_i * d_i^2 + I_i)$, where m_i is the mass of clump member i, d_i is the distance from center of clump member i to clump's centroid and I_i is the inertia tensor of the clump member i.

- For overlapping clump members the clump's mass m_c is summed over cells using a regular grid spacing inside axis-aligned bounding box (*Aabb*) of the clump, the inertia tensor is computed using the parallel axes theorem: $\mathbf{I}_c = \sum_j (m_j * d_j^2 + I_j)$, where m_j is the mass of cell j, d_j is the distance from cell center to clump's centroid and I_j is the inertia tensor of the cell j.

Local axes are oriented such that they are principal and inertia tensor is diagonal and clump's orientation is changed to compensate rotation of the local system, as to not change the clump members' positions in global space. Initial positions and orientations of all clump members in local coordinate system are stored.

In Yade (class *Clump*), clump members behave as stand-alone particles during simulation for purposes of collision detection and contact resolution, except that they have no contacts created among themselves within one clump. It is at the stage of motion integration that they are treated specially. Instead of integrating each of them separately, forces/torques on those particles $\mathbf{F}_i$, $\mathbf{T}_i$ are converted to forces/torques on the clump itself. Let us denote r_i relative position of each particle with regards to clump's centroid, in global orientation. Then summary force and torque on the clump are

$$\mathbf{F}_c = \sum F_i,$$
$$\mathbf{T}_c = \sum r_i \times F_i + T_i.$$

Motion of the clump is then integrated, using aspherical rotation integration. Afterwards, clump members are displaced in global space, to keep their initial positions and orientations in the clump's local coordinate system. In such a way, relative positions of clump members are always the same, resulting in the behavior of a rigid aggregate.

7.5.5 Numerical damping

In simulations of quasi-static phenomena, it it desirable to dissipate kinetic energy of particles. Since most constitutive laws (including *Law_ScGeom_FrictPhys_Basic* shown above, *Stress evaluation (example)*) do not include velocity-based damping (such as one in *[Addetta2001]*), it is possible to use artificial numerical damping. The formulation is described in *[Pfc3dManual30]*, although our version is slightly adapted. The basic idea is to decrease forces which increase the particle velocities and vice versa by $(\Delta F)_d$, comparing the current acceleration sense and particle velocity sense. This is done by component, which makes the damping scheme clearly non-physical, as it is not invariant with respect to coordinate system rotation; on the other hand, it is very easy to compute. Cundall proposed the form (we omit particle indices i since it applies to all of them separately):

$$\frac{(\Delta \mathbf{F})_{dw}}{\mathbf{F}_w} = -\lambda_d \operatorname{sgn}(\mathbf{F}_w \dot{\mathbf{u}}_w^{\ominus}), \quad w \in \{x, y, z\}$$

where λ_d is the damping coefficient. This formulation has several advantages *[Hentz2003]*:

- it acts on forces (accelerations), not constraining uniform motion;
- it is independent of eigenfrequencies of particles, they will be all damped equally;
- it needs only the dimensionless parameter λ_d which does not have to be scaled.

In Yade, we use the adapted form

$$\frac{(\Delta \mathbf{F})_{dw}}{\mathbf{F}_w} = -\lambda_d \operatorname{sgn} \mathbf{F}_w \underbrace{\left(\dot{u}_w^{\ominus} + \frac{\ddot{\mathbf{u}}_w^{\circ} \Delta t}{2} \right)}_{\simeq \dot{u}_w^{\circ}}, \tag{7.9}$$

where we replaced the previous mid-step velocity $\dot{u}^{\ominus}$ by its on-step estimate in parentheses. This is to avoid locked-in forces that appear if the velocity changes its sign due to force application at each step, i.e. when the particle in question oscillates around the position of equilibrium with $2\Delta t$ period.

In Yade, damping (7.9) is implemented in the *NewtonIntegrator* engine; the damping coefficient λ_d is *NewtonIntegrator.damping*.

7.5.6 Stability considerations

Critical timestep

In order to ensure stability for the explicit integration sceheme, an upper limit is imposed on Δt:

$$\Delta t_{cr} = \frac{2}{\omega_{max}} \tag{7.10}$$

where ω_{max} is the highest eigenfrequency within the system.

Single mass-spring system

Single 1D mass-spring system with mass m and stiffness K is governed by the equation

$$m\ddot{x} = -Kx$$

where x is displacement from the mean (equilibrium) position. The solution of harmonic oscillation is $x(t) = A\cos(\omega t + \varphi)$ where phase φ and amplitude A are determined by initial conditions. The angular frequency

$$\omega^{(1)} = \sqrt{\frac{K}{m}} \tag{7.11}$$

does not depend on initial conditions. Since there is one single mass, $\omega^{(1)}_{max} = \omega^{(1)}$. Plugging (7.11) into (7.10), we obtain

$$\Delta t^{(1)}_{cr} = 2/\omega^{(1)}_{max} = 2\sqrt{m/K}$$

for a single oscillator.

General mass-spring system

In a general mass-spring system, the highest frequency occurs if two connected masses m_i, m_j are in opposite motion; let us suppose they have equal velocities (which is conservative) and they are connected by a spring with stiffness K_i: displacement Δx_i of m_i will be accompained by $\Delta x_j = -\Delta x_i$ of m_j, giving $\Delta F_i = -K_i(\Delta x_i - (-\Delta x_i)) = -2K_i\Delta x_i$. That results in apparent stiffness $K^{(2)}_i = 2K_i$, giving maximum eigenfrequency of the whole system

$$\omega_{max} = \max_i \sqrt{K^{(2)}_i/m_i}.$$

The overall critical timestep is then

$$\Delta t_{cr} = \frac{2}{\omega_{max}} = \min_i 2\sqrt{\frac{m_i}{K^{(2)}_i}} = \min_i 2\sqrt{\frac{m_i}{2K_i}} = \min_i \sqrt{2}\sqrt{\frac{m_i}{K_i}}. \tag{7.12}$$

This equation can be used for all 6 degrees of freedom (DOF) in translation and rotation, by considering generalized mass and stiffness matrices M and K, and replacing fractions $\frac{m_i}{K_i}$ by eigen values of $M.K^{-1}$. The critical timestep is then associated to the eigen mode with highest frequency :

$$\Delta t_{cr} = \min \Delta t_{crk}, \quad k \in \{1, ..., 6\}. \tag{7.13}$$

DEM simulations

In DEM simulations, per-particle stiffness $\mathbf{K}_{ij}$ is determined from the stiffnesses of contacts in which it participates *[Chareyre2005]*. Suppose each contact has normal stiffness K_{Nk}, shear stiffness K_{Tk} =

ξK_{Nk} and is oriented by normal $\mathbf{n}_k$. A translational stiffness matrix $\mathbf{K}_{ij}$ can be defined as the sum of contributions of all contacts in which it participates (indices k), as

$$\mathbf{K}_{ij} = \sum_k (K_{Nk} - K_{Tk})\mathbf{n}_i\mathbf{n}_j + K_{Tk} = \sum_j K_{Nk}\left((1-\xi)\mathbf{n}_i\mathbf{n}_j + \xi\right) \tag{7.14}$$

with i and $j \in \{x, y, z\}$. Equations (7.13) and (7.14) determine Δt_{cr} in a simulation. A similar approach generalized to all 6 DOFs is implemented by the *GlobalStiffnessTimeStepper* engine in Yade. The derivation of generalized stiffness including rotational terms is very similar but not developped here, for simplicity. For full reference, see "PFC3D - Theoretical Background".

Note that for computation efficiency reasons, eigenvalues of the stiffness matrices are not computed. They are only approximated assuming than DOF's are uncoupled, and using diagonal terms of $K.M^{-1}$. They give good approximates in typical mechanical systems.

There is one important condition that $\omega_{max} > 0$: if there are no contacts between particles and $\omega_{max} = 0$, we would obtain value $\Delta t_{cr} = \infty$. While formally correct, this value is numerically erroneous: we were silently supposing that stiffness remains constant during each timestep, which is not true if contacts are created as particles collide. In case of no contact, therefore, stiffness must be pre-estimated based on future interactions, as shown in the next section.

Estimation of Δt_{cr} by wave propagation speed

Estimating timestep in absence of interactions is based on the connection between interaction stiffnesses and the particle's properties. Note that in this section, symbols E and ρ refer exceptionally to Young's modulus and density of *particles*, not of macroscopic arrangement.

In Yade, particles have associated *Material* which defines density ρ (*Material.density*), and also may define (in *ElastMat* and derived classes) particle's "Young's modulus" E (*ElastMat.young*). ρ is used when particle's mass m is initially computed from its ρ, while E is taken in account when creating new interaction between particles, affecting stiffness K_N. Knowing m and K_N, we can estimate (7.14) for each particle; we obviously neglect

- number of interactions per particle N_i; for a "reasonable" radius distribution, however, there is a geometrically imposed upper limit (6 for a 2D-packing of spheres with equal radii, for instance);
- the exact relationship the between particles' rigidities E_i, E_j, supposing only that K_N is somehow proportional to them.

By defining E and ρ, particles have continuum-like quantities. Explicit integration schemes for continuum equations impose a critical timestep based on sonic speed $\sqrt{E/\rho}$; the elastic wave must not propagate farther than the minimum distance of integration points l_{min} during one step. Since E, ρ are parameters of the elastic continuum and l_{min} is fixed beforehand, we obtain

$$\Delta t_{cr}^{(c)} = l_{min}\sqrt{\frac{\rho}{E}}.$$

For our purposes, we define E and ρ for each particle separately; l_{min} can be replaced by the sphere's radius R_i; technically, $l_{min} = 2R_i$ could be used, but because of possible interactions of spheres and facets (which have zero thickness), we consider $l_{min} = R_i$ instead. Then

$$\Delta t_{cr}^{(p)} = \min_i R_i\sqrt{\frac{\rho_i}{E_i}}.$$

This algorithm is implemented in the *utils.PWaveTimeStep* function.

Let us compare this result to (7.12); this necessitates making several simplifying hypotheses:

- all particles are spherical and have the same radius R;
- the sphere's material has the same E and ρ;
- the average number of contacts per sphere is N;
- the contacts have sufficiently uniform spatial distribution around each particle;

- the $\xi = K_N/K_T$ ratio is constant for all interactions;
- contact stiffness K_N is computed from E using a formula of the form

$$K_N = E\pi' R', \tag{7.15}$$

where π' is some constant depending on the algorithm in usefootnote{For example, $\pi' = \pi/2$ in the concrete particle model (*Ip2_CpmMat_CpmMat_CpmPhys*), while $\pi' = 2$ in the classical DEM model (*Ip2_FrictMat_FrictMat_FrictPhys*) as implemented in Yade.} and R' is half-distance between spheres in contact, equal to R for the case of interaction radius $R_I = 1$. If $R_I = 1$ (and $R' \equiv R$ by consequence), all interactions will have the same stiffness K_N. In other cases, we will consider K_N as the average stiffness computed from average R' (see below).

As all particles have the same parameters, we drop the i index in the following formulas.

We try to express the average per-particle stiffness from (7.14). It is a sum over all interactions where K_N and ξ are scalars that will not rotate with interaction, while $\mathbf{n}_w$ is w-th component of unit interaction normal $\mathbf{n}$. Since we supposed uniform spatial distribution, we can replace $\mathbf{n}_w^2$ by its average value $\overline{\mathbf{n}}_w^2$. Recognizing components of $\mathbf{n}$ as direction cosines, the average values of $\mathbf{n}_w^2$ is $1/3$. %we find the average value by integrating over all possible orientations, which are uniformly distributed in space:

Moreover, since all directions are equal, we can write the per-body stiffness as $K = \mathbf{K}_w$ for all $w \in \{x, y, z\}$. We obtain

$$K = \sum K_N \left((1-\xi)\frac{1}{3} + \xi \right) = \sum K_N \frac{1-2\xi}{3}$$

and can put constant terms (everything) in front of the summation. $\sum 1$ equals the number of contacts per sphere, i.e. N. Arriving at

$$K = N K_N \frac{1-2\xi}{3},$$

we substitute K into (7.12) using (7.15):

$$\Delta t_{cr} = \sqrt{2}\sqrt{\frac{m}{K}} = \sqrt{2}\sqrt{\frac{\frac{4}{3}\pi R^3 \rho}{N E \pi' R \frac{1-2\xi}{3}}} = \underbrace{R\sqrt{\frac{\rho}{E}}}_{\Delta t_{cr}^{(p)}} 2\sqrt{\frac{\pi/\pi'}{N(1-2\xi)}}.$$

The ratio of timestep $\Delta t_{cr}^{(p)}$ predicted by the p-wave velocity and numerically stable timestep Δt_{cr} is the inverse value of the last (dimensionless) term:

$$\frac{\Delta t_{cr}^{(p)}}{\Delta t_{cr}} = 2\sqrt{\frac{N(1+\xi)}{\pi/\pi'}}.$$

Actual values of this ratio depend on characteristics of packing N, $K_N/K_T = \xi$ ratio and the way of computing contact stiffness from particle rigidity. Let us show it for two models in Yade:

Concrete particle model computes contact stiffness from the equivalent area A_{eq} first (7.6),

$$A_{eq} = \pi R^2 K_N \qquad\qquad = \frac{A_{eq} E}{d_0}.$$

d_0 is the initial contact length, which will be, for interaction radius (7.5) $R_I > 1$, in average larger than $2R$. For $R_I = 1.5$ (sect. sect-calibration-elastic-properties), we can roughly estimate $\overline{d}_0 = 1.25 \cdot 2R = \frac{5}{2}R$, getting

$$K_N = E\left(\frac{2}{5}\pi\right) R$$

where $\frac{2}{5}\pi = \pi'$ by comparison with (7.15).

Interaction radius $R_I = 1.5$ leads to average $N \approx 12$ interactions per sphere for dense packing of spheres with the same radius R. $\xi = 0.2$ is calibrated (sect. sect-calibration-elastic-properties) to match the desired macroscopic Poisson's ratio $\nu = 0.2$.

Finally, we obtain the ratio

$$\frac{\Delta t_{cr}^{(p)}}{\Delta t_{cr}} = 2\sqrt{\frac{12(1-2\cdot 0.2)}{\frac{\pi}{(2/5)\pi}}} = 3.39,$$

showing significant overestimation by the p-wave algorithm.

Non-cohesive dry friction model is the basic model proposed by Cundall explained in *Stress evaluation (example)*. Supposing almost-constant sphere radius R and rather dense packing, each sphere will have $N = 6$ interactions on average (that corresponds to maximally dense packing of spheres with a constant radius). If we use the *Ip2_FrictMat_FrictMat_FrictPhys* class, we have $\pi' = 2$, as $K_N = E2R$; we again use $\xi = 0.2$ (for lack of a more significant value). In this case, we obtain the result

$$\frac{\Delta t_{cr}^{(p)}}{\Delta t_{cr}} = 2\sqrt{\frac{6(1-2\cdot 0.2)}{\pi/2}} = 3.02$$

which again overestimates the numerical critical timestep.

To conclude, p-wave timestep gives estimate proportional to the real Δt_{cr}, but in the cases shown, the value of about $\Delta t = 0.3\Delta t_{cr}^{(p)}$ should be used to guarantee stable simulation.

Non-elastic Δt constraints

Let us note at this place that not only Δt_{cr} assuring numerical stability of motion integration is a constraint. In systems where particles move at relatively high velocities, position change during one timestep can lead to non-elastic irreversible effects such as damage. The Δt needed for reasonable result can be lower Δt_{cr}. We have no rigorously derived rules for such cases.

7.6 Periodic boundary conditions

While most DEM simulations happen in R^3 space, it is frequently useful to avoid boundary effects by using periodic space instead. In order to satisfy periodicity conditions, periodic space is created by repetition of parallelepiped-shaped cell. In Yade, periodic space is implemented in the *Cell* class. The geometry of the cell in the reference coordinates system is defined by three edges of the parallepiped. The corresponding base vectors are stored in the columns of matrix $\mathbf{H}$ (*Cell.hSize*).

The initial $\mathbf{H}$ can be explicitly defined as a 3x3 matrix at the beginning of the simulation. There are no restricitions on the possible shapes: any parallelepiped is accepted as the initial cell. If the base vectors are axis-aligned, defining only their sizes can be more convenient than defining the full $\mathbf{H}$ matrix; in that case it is enough to define the norms of columns in $\mathbf{H}$ (see *Cell.size*).

After the definition of the initial cell's geometry, $\mathbf{H}$ should generally not be modified by direct assignment. Instead, its deformation rate will be defined via the velocity gradient *Cell.velGrad* described below. It is the only variable that let the period deformation be correctly accounted for in constitutive laws and Newton integrator (*NewtonIntegrator*).

7.6.1 Deformations handling

The deformation of the cell over time is defined via a matrix representing the gradient of an homogeneous velocity field $\nabla \boldsymbol{v}$ (*Cell.velGrad*). This gradient represents arbitrary combinations of rotations and stretches. It can be imposed externaly or updated by *boundary controllers* (see *PeriTriaxController* or *Peri3dController*) in order to reach target strain values or to maintain some prescribed stress.

The velocity gradient is integrated automatically over time, and the cumulated transformation is reflected in the transformation matrix $\mathbf{F}$ (*Cell.trsf*) and the current shape of the cell $\mathbf{H}$. The per-step transformation update reads (it is similar for $\mathbf{H}$), with I the identity matrix:

$$\mathbf{F}^{+} = (I + \nabla \boldsymbol{v}\Delta t)\mathbf{F}^{\circ}.$$

$\mathbf{F}$ can be set back to identity at any point in simulations, in order to define the current state as reference for strains definition in boundary controllers. It will have no effect on $\mathbf{H}$.

Along with the automatic integration of cell transformation, there is an option to homothetically displace all particles so that $\nabla\boldsymbol{v}$ is applied over the whole simulation (enabled via *Cell.homoDeform*). This avoids all boundary effects coming from change of the velocity gradient.

7.6.2 Collision detection in periodic cell

In usual implementations, particle positions are forced to be inside the cell by wrapping their positions if they get over the boundary (so that they appear on the other side). As we wanted to avoid abrupt changes of position (it would make particle's velocity inconsistent with step displacement change), a different method was chosen.

Approximate collision detection

Pass 1 collision detection (based on sweep and prune algorithm, sect. *Sweep and prune*) operates on axis-aligned bounding boxes (*Aabb*) of particles. During the collision detection phase, bounds of all *Aabb's* are wrapped inside the cell in the first step. At subsequent runs, every bound remembers by how many cells it was initially shifted from coordinate given by the *Aabb* and uses this offset repeatedly as it is being updated from *Aabb* during particle's motion. Bounds are sorted using the periodic insertion sort algorithm (sect. *Periodic insertion sort algorithm*), which tracks periodic cell boundary ‖.

Upon inversion of two *Aabb*'s, their collision along all three axes is checked, wrapping real coordinates inside the cell for that purpose.

This algorithm detects collisions as if all particles were inside the cell but without the need of constructing "ghost particles" (to represent periodic image of a particle which enters the cell from the other side) or changing the particle's positions.

It is required by the implementation (and partly by the algorithm itself) that particles do not span more than half of the current cell size along any axis; the reason is that otherwise two (or more) contacts between both particles could appear, on each side. Since Yade identifies contacts by *Body.id* of both bodies, they would not be distinguishable.

In presence of shear, the sweep-and-prune collider could not sort bounds independently along three axes: collision along x axis depends on the mutual position of particles on the y axis. Therefore, bounding boxes *are expressed in transformed coordinates* which are perpendicular in the sense of collision detection. This requires some extra computation: *Aabb* of sphere in transformed coordinates will no longer be cube, but cuboid, as the sphere itself will appear as ellipsoid after transformation. Inversely, the sphere in simulation space will have a parallelepiped bounding "box", which is cuboid around the ellipsoid in transformed axes (the *Aabb* has axes aligned with transformed cell basis). This is shown in fig. *fig-cell-shear-aabb*.

The restriction of a single particle not spanning more than half of the transformed axis becomes stringent as *Aabb* is enlarged due to shear. Considering *Aabb* of a sphere with radius r in the cell where $x' \equiv x$, $z' \equiv z$, but $\angle(y, y') = \varphi$, the x-span of the *Aabb* will be multiplied by $1/\cos\varphi$. For the infinite shear $\varphi \to \pi/2$, which can be desirable to simulate, we have $1/\cos\varphi \to \infty$. Fortunately, this limitation can be easily circumvented by realizing the quasi-identity of all periodic cells which, if repeated in space, create the same grid with their corners: the periodic cell can be flipped, keeping all particle interactions intact, as shown in fig. *fig-cell-flip*. It only necessitates adjusting the *Interaction.cellDist* of interactions and re-initialization of the collider (`Collider::invalidatePersistentData`). Cell flipping is implemented in the *utils.flipCell* function.

This algorithm is implemented in *InsertionSortCollider* and is used whenever simulation is periodic (*Omega.isPeriodic*); individual *BoundFunctor's* are responsible for computing sheared *Aabb's*; currently it is implemented for spheres and facets (in *Bo1_Sphere_Aabb* and *Bo1_Facet_Aabb* respectively).

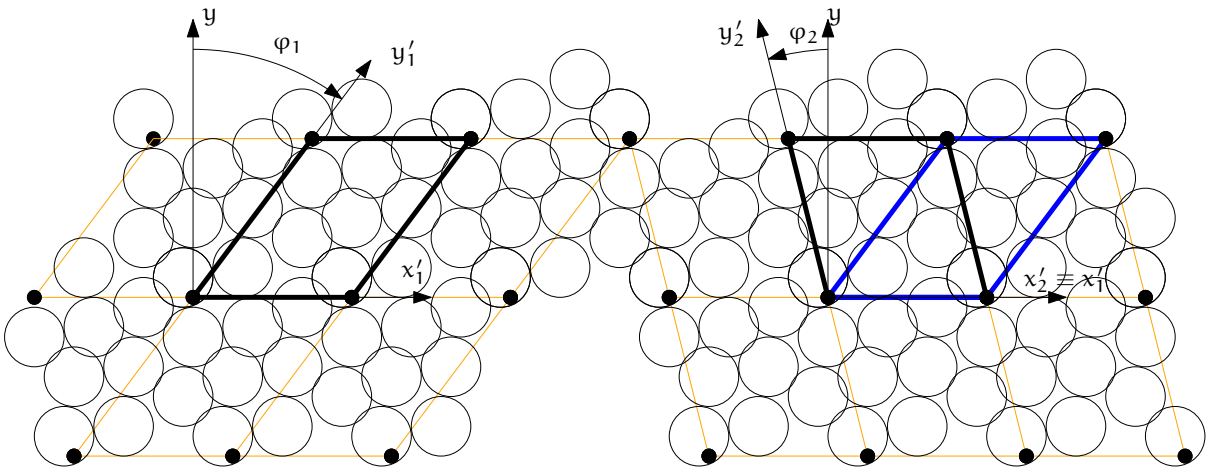


Fig. 7.6: Flipping cell (*utils.flipCell*) to avoid infinite stretch of the bounding boxes' spans with growing φ. Cell flip does not affect interactions from the point of view of the simulation. The periodic arrangement on the left is the same as the one on the right, only the cell is situated differently between identical grid points of repetition; at the same time $|\varphi_2| < |\varphi_1|$ and sphere bounding box's x-span stretched by $1/\cos\varphi$ becomes smaller. Flipping can be repeated, making effective infinite shear possible.

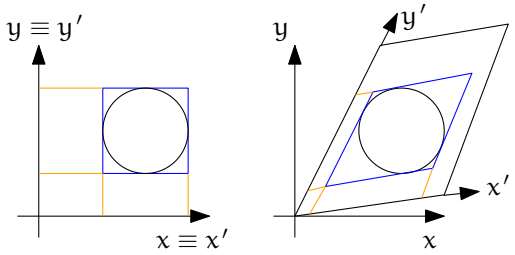


Fig. 7.7: Constructing axis-aligned bounding box (*Aabb*) of a sphere in simulation space coordinates (without periodic cell – left) and transformed cell coordinates (right), where collision detection axes x', y' are not identical with simulation space axes x, y. Bounds' projection to axes is shown by orange lines.

Exact collision detection

When the collider detects approximate contact (on the *Aabb* level) and the contact does not yet exist, it creates *potential* contact, which is subsequently checked by exact collision algorithms (depending on the combination of *Shapes*). Since particles can interact over many periodic cells (recall we never change their positions in simulation space), the collider embeds the relative cell coordinate of particles in the interaction itself (*Interaction.cellDist*) as an *integer* vector $\mathbf{c}$. Multiplying current cell size $\mathbf{Ts}$ by $\mathbf{c}$ component-wise, we obtain particle offset $\Delta\mathbf{x}$ in aperiodic R^3; this value is passed (from *InteractionLoop*) to the functor computing exact collision (*IGeomFunctor*), which adds it to the position of the particle *Interaction.id2*.

By storing the integral offset $\mathbf{c}$, $\Delta\mathbf{x}$ automatically updates as cell parameters change.

Periodic insertion sort algorithm

The extension of sweep and prune algorithm (described in *Sweep and prune*) to periodic boundary conditions is non-trivial. Its cornerstone is a periodic variant of the insertion sort algorithm, which involves keeping track of the "period" of each boundary; e.g. taking period $\langle 0, 10\rangle$, then $8_1 \equiv -2_2 < 2_2$ (subscript indicating period). Doing so efficiently (without shuffling data in memory around as bound wraps from one period to another) requires moving period boundary rather than bounds themselves and making the comparison work transparently at the edge of the container.

This algorithm was also extended to handle non-orthogonal periodic *Cell* boundaries by working in transformed rather than Cartesian coordinates; this modifies computation of *Aabb* from Cartesian coordinates in which bodies are positioned (treated in detail in *Approximate collision detection*).

The sort algorithm is tracking *Aabb* extrema along all axes. At the collider's initialization, each value is assigned an integral period, i.e. its distance from the cell's interior expressed in the cell's dimension along its respective axis, and is wrapped to a value inside the cell. We put the period number in subscript.

Let us give an example of coordinate sequence along x axis (in a real case, the number of elements would be even, as there is maximum and minimum value couple for each particle; this demonstration only shows the sorting algorithm, however.)

$$4_1 \qquad 12_2 \quad \| \quad -1_2 \qquad -2_4 \qquad 5_0$$

with cell x-size $s_x = 10$. The 4_1 value then means that the real coordinate x_i of this extremum is $x_i + 1 \cdot 10 = 4$, i.e. $x_i = -4$. The $\|$ symbol denotes the periodic cell boundary.

Sorting starts from the first element in the cell, i.e. right of $\|$, and inverts elements as in the aperiodic variant. The rules are, however, more complicated due to the presence of the boundary $\|$:

$(\leq)$	stop inverting if neighbors are ordered;
$(\|\bullet)$	current element left of $\|$ is below 0 (lower period boundary); in this case, decrement element's period, decrease its coordinate by s_x and move $\|$ right;
$(\bullet\|)$	current element right of $\|$ is above s_x (upper period boundary); increment element's period, increase its coordinate by s_x and move $\|$ left;
$(\nleftrightarrow)$	inversion across $\|$ must subtract s_x from the left coordinate during comparison. If the elements are not in order, they are swapped, but they must have their periods changed as they traverse $\|$. Apply $(\|\circ)$ if necessary;
$(\|\circ)$	if after $(\nleftrightarrow)$ the element that is now right of $\|$ has $x_i < s_x$, decrease its coordinate by s_x and decrement its period. Do not move $\|$.

In the first step, $(\|\bullet)$ is applied, and inversion with 12_2 happens; then we stop because of $(\leq)$:

$$4_1 \qquad 12_2 \quad \| \quad \boxed{-1_2} \qquad -2_4 \qquad 5_0,$$

$$4_1 \qquad 12_2 \xleftarrow{\not\leq} \boxed{9_1} \quad \| \quad -2_4 \qquad 5_0,$$

$$4_1 \xleftarrow{\leq} \boxed{9_1} \qquad 12_2 \quad \| \quad -2_4 \qquad 5_0.$$

We move to next element $\boxed{-2_4}$; first, we apply ($\|\bullet$), then invert until ($\leq$):

$$\begin{aligned}
&4_1 \quad 9_1 \quad 12_2 \quad \| \quad \boxed{-2_4} \quad 5_0,\\
&4_1 \quad 9_1 \quad 12_2 \quad \boxed{8_3} \quad \| \quad 5_0, \qquad (\not\leq)\\
&4_1 \quad 9_1 \quad \boxed{8_3} \quad 12_2 \quad \| \quad 5_0, \qquad (\not\leq)\\
&4_1 \quad \boxed{8_3} \quad 9_1 \quad 12_2 \quad \| \quad 5_0. \qquad (\leq)
\end{aligned}$$

The next element is $\boxed{5_0}$; we satisfy ($\nparallel$), therefore instead of comparing $12_2 > 5_0$, we must do $(12_2 - s_x) = 2_3 \leq 5$; we adjust periods when swapping over $\|$ and apply ($\|\circ$), turning 12_2 into 2_3; then we keep inverting, until ($\leq$):

$$\begin{aligned}
&4_1 \quad 8_3 \quad 9_1 \quad 12_2 \quad \| \quad \boxed{5_0}, \qquad (\not\leq)\\
&4_1 \quad 8_3 \quad 9_1 \quad \boxed{5_{-1}} \quad \| \quad 2_3, \qquad (\not\leq)\\
&4_1 \quad 8_3 \quad \boxed{5_{-1}} \quad 9_1 \quad \| \quad 2_3, \qquad (\not\leq)\\
&4_1 \quad \boxed{5_{-1}} \quad 8_3 \quad 9_1 \quad \| \quad 2_3. \qquad (\leq)
\end{aligned}$$

We move (wrapping around) to $\boxed{4_1}$, which is ordered:

$$\boxed{4_1} \quad 5_{-1} \quad 8_3 \quad 9_1 \quad \| \quad 2_3 \qquad (\geq)$$

and so is the last element

$$4_1 \quad \boxed{5_{-1}} \quad 8_3 \quad 9_1 \quad \| \quad 2_3. \qquad (\leq)$$

7.7 Computational aspects

7.7.1 Cost

The DEM computation using an explicit integration scheme demands a relatively high number of steps during simulation, compared to implicit scehemes. The total computation time Z of simulation spanning T seconds (of simulated time), containing N particles in volume V depends on:

- linearly, the number of steps $i = T/(s_t \Delta t_{cr})$, where s_t is timestep safety factor; Δt_{cr} can be estimated by p-wave velocity using E and ρ (sect. *Estimation of by wave propagation speed*) as $\Delta t_{cr}^{(p)} = r\sqrt{\frac{\rho}{E}}$. Therefore

$$i = \frac{T}{s_t r}\sqrt{\frac{E}{\rho}}.$$

- the number of particles N; for fixed value of simulated domain volume V and particle radius r

$$N = p\frac{V}{\frac{4}{3}\pi r^3},$$

where p is packing porosity, roughly $\frac{1}{2}$ for dense irregular packings of spheres of similar radius.

The dependency is not strictly linear (which would be the best case), as some algorithms do not scale linearly; a case in point is the sweep and prune collision detection algorithm introduced in sect. *Sweep and prune*, with scaling roughly $\mathcal{O}(N \log N)$.

The number of interactions scales with N, as long as packing characteristics are the same.

- the number of computational cores n_{cpu}; in the ideal case, the dependency would be inverse-linear were all algorithms parallelized (in Yade, collision detection is not).

Let us suppose linear scaling. Additionally, let us suppose that the material to be simulated (E, ρ) and the simulation setup (V, T) are given in advance. Finally, dimensionless constants s_t, p and n_{cpu} will have a fixed value. This leaves us with one last degree of freedom, r. We may write

$$Z \propto iN\frac{1}{n_{cpu}} = \frac{T}{s_t r}\sqrt{\frac{E}{\rho}}p\frac{V}{\frac{4}{3}\pi r^3}\frac{1}{n_{cpu}} \propto \frac{1}{r}\frac{1}{r^3} = \frac{1}{r^4}.$$

This (rather trivial) result is essential to realize DEM scaling; if we want to have finer results, refining the "mesh" by halving r, the computation time will grow $2^4 = 16$ times.

For very crude estimates, one can use a known simulation to obtain a machine "constant"

$$\mu = \frac{Z}{Ni}$$

with the meaning of time per particle and per timestep (in the order of 10^{-6} s for current machines). μ will be only useful if simulation characteristics are similar and non-linearities in scaling do not have major influence, i.e. N should be in the same order of magnitude as in the reference case.

7.7.2 Result indeterminism

It is naturally expected that running the same simulation several times will give exactly the same results: although the computation is done with finite precision, round-off errors would be deterministically the same at every run. While this is true for *single-threaded* computation where exact order of all operations is given by the simulation itself, it is not true anymore in *multi-threaded* computation which is described in detail in later sections.

The straight-forward manner of parallel processing in explicit DEM is given by the possibility of treating interactions in arbitrary order. Strain and stress is evaluated for each interaction independently, but forces from interactions have to be summed up. If summation order is also arbitrary (in Yade, forces are accumulated for each thread in the order interactions are processed, then summed together), then the results can be slightly different. For instance

```
(1/10.)+(1/13.)+(1/17.)=0.23574660633484162
(1/17.)+(1/13.)+(1/10.)=0.23574660633484165
```

As forces generated by interactions are assigned to bodies in quasi-random order, summary force F_i on the body can be different between single-threaded and multi-threaded computations, but also between different runs of multi-threaded computation with exactly the same parameters. Exact thread scheduling by the kernel is not predictable since it depends on asynchronous events (hardware interrupts) and other unrelated tasks running on the system; and it is thread scheduling that ultimately determines summation order of force contributions from interactions.

Numerical damping influence

The effect of summation order can be significantly amplified by the usage of a *discontinuous* damping function in *NewtonIntegrator* given in (7.9) as

$$\frac{(\Delta \mathbf{F})_{dw}}{\mathbf{F}_w} = -\lambda_d \operatorname{sgn}\mathbf{F}_w\left(\dot{u}^{\ominus}_w + \frac{\ddot{u}^{\circ}_w \Delta t}{2}\right).$$

If the sgn argument is close to zero then the least significant finite precision artifact can determine whether the equation (relative increment of $\mathbf{F}_w$) is $+\lambda_d$ or $-\lambda_d$. Given commonly used values of $\lambda_d = 0.4$, it means that such artifact propagates from least significant place to the most significant one at once.

Chapter 8

Class reference (yade.wrapper module)

8.1 Bodies

8.1.1 Body

class `yade.wrapper.Body`(*inherits Serializable*)

A particle, basic element of simulation; interacts with other bodies.

`aspherical`(=*false*)
: Whether this body has different inertia along principal axes; *NewtonIntegrator* makes use of this flag to call rotation integration routine for aspherical bodies, which is more expensive.

`bound`(=*uninitalized*)
: *Bound*, approximating volume for the purposes of collision detection.

`bounded`(=*true*)
: Whether this body should have *Body.bound* created. Note that bodies without a *bound* do not participate in collision detection. (In c++, use `Body::isBounded`/`Body::setBounded`)

`chain`
: Returns Id of chain to which the body belongs.

`clumpId`
: Id of clump this body makes part of; invalid number if not part of clump; see *Body::isStandalone*, *Body::isClump*, *Body::isClumpMember* properties.

  Not meant to be modified directly from Python, use *O.bodies.appendClumped* instead.

`dict`() → dict
: Return dictionary of attributes.

`dynamic`(=*true*)
: Whether this body will be moved by forces. (In c++, use `Body::isDynamic`/`Body::setDynamic`)

`flags`(=*FLAG_BOUNDED*)
: Bits of various body-related flags. *Do not access directly.* In c++, use isDynamic/setDynamic, isBounded/setBounded, isAspherical/setAspherical. In python, use *Body.dynamic*, *Body.bounded*, *Body.aspherical*.

`groupMask`(=*1*)
: Bitmask for determining interactions.

id(=*Body::ID_NONE*)
Unique id of this body.

intrs() → list
Return all interactions in which this body participates.

isClump
True if this body is clump itself, false otherwise.

isClumpMember
True if this body is clump member, false otherwise.

isStandalone
True if this body is neither clump, nor clump member; false otherwise.

iterBorn
Returns step number at which the body was added to simulation.

mask
Shorthand for *Body::groupMask*

mat
Shorthand for *Body::material*

material(=*uninitalized*)
Material instance associated with this body.

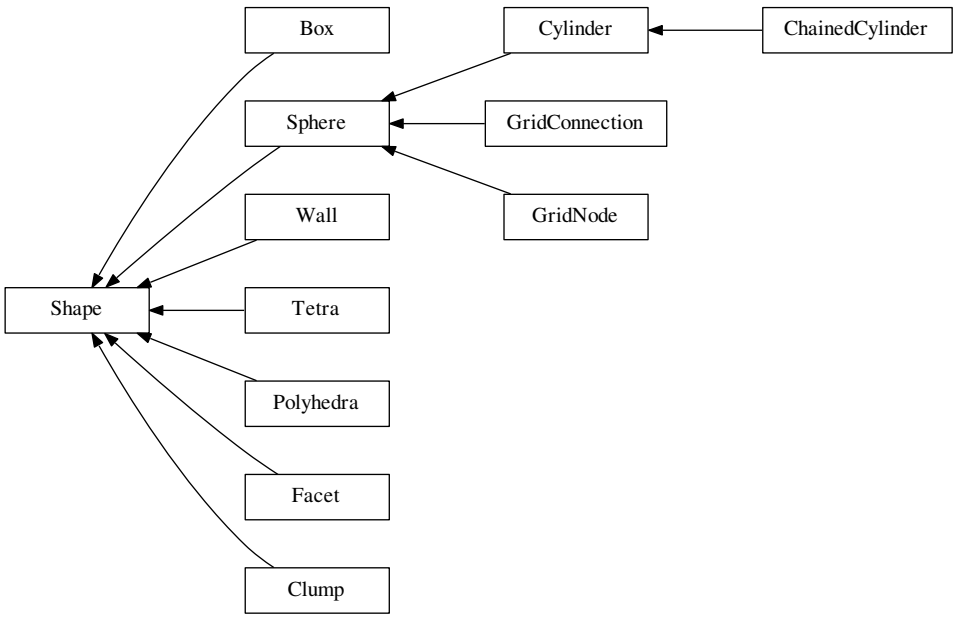
shape(=*uninitalized*)
Geometrical *Shape*.

state(=*new State*)
Physical *state*.

timeBorn
Returns time at which the body was added to simulation.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

8.1.2 Shape



class `yade.wrapper.Shape`(*inherits Serializable*)
Geometry of a body

`color`(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

`dict`() → dict
Return dictionary of attributes.

`dispHierarchy`([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

`dispIndex`
Return class index of this instance.

`highlight`(=*false*)
Whether this Shape will be highlighted when rendered.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wire`(=*false*)
Whether this Shape is rendered using color surfaces, or only wireframe (can still be overridden by global config of the renderer).

class `yade.wrapper.Box`(*inherits Shape → Serializable*)
Box (cuboid) particle geometry. (Avoid using in new code, prefer *Facet* instead.

`color`(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

`dict`() → dict
Return dictionary of attributes.

`dispHierarchy`([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

`dispIndex`
Return class index of this instance.

`extents`(=*uninitalized*)
Half-size of the cuboid

`highlight`(=*false*)
Whether this Shape will be highlighted when rendered.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wire`(=*false*)
Whether this Shape is rendered using color surfaces, or only wireframe (can still be overridden by global config of the renderer).

class `yade.wrapper.ChainedCylinder`(*inherits Cylinder → Sphere → Shape → Serializable*)
Geometry of a deformable chained cylinder, using geometry *Cylinder*.

`chainedOrientation`(=*Quaternionr::Identity()*)
Deviation of node1 orientation from node-to-node vector

`color`(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

`dict`() → dict
Return dictionary of attributes.

dispHierarchy(*[(bool)names=True]*) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

highlight(=*false*)
Whether this Shape will be highlighted when rendered.

initLength(=*0*)
tensile-free length, used as reference for tensile strain

length(=*NaN*)
Length [m]

radius(=*NaN*)
Radius [m]

segment(=*Vector3r::Zero()*)
Length vector

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

wire(=*false*)
Whether this Shape is rendered using color surfaces, or only wireframe (can still be overridden by global config of the renderer).

class yade.wrapper.**Clump**(*inherits Shape → Serializable*)
Rigid aggregate of bodies

color(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

dict() → dict
Return dictionary of attributes.

dispHierarchy(*[(bool)names=True]*) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

highlight(=*false*)
Whether this Shape will be highlighted when rendered.

members
Return clump members as {'id1':(relPos,relOri),...}

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

wire(=*false*)
Whether this Shape is rendered using color surfaces, or only wireframe (can still be overridden by global config of the renderer).

class yade.wrapper.**Cylinder**(*inherits Sphere → Shape → Serializable*)
Geometry of a cylinder, as Minkowski sum of line and sphere.

color(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

dict() → dict
Return dictionary of attributes.

`dispHierarchy(`[*(bool)names=True*]`)` → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

`dispIndex`
Return class index of this instance.

`highlight(`=*false*`)`
Whether this Shape will be highlighted when rendered.

`length(`=*NaN*`)`
Length [m]

`radius(`=*NaN*`)`
Radius [m]

`segment(`=*Vector3r::Zero()*`)`
Length vector

`updateAttrs(`*(dict)arg2*`)` → None
Update object attributes from given dictionary

`wire(`=*false*`)`
Whether this Shape is rendered using color surfaces, or only wireframe (can still be overridden by global config of the renderer).

class `yade.wrapper.Facet(`*inherits Shape → Serializable*`)`
Facet (triangular particle) geometry.

`area(`=*NaN*`)`
Facet's area

`color(`=*Vector3r(1, 1, 1)*`)`
Color for rendering (normalized RGB).

`dict()` → dict
Return dictionary of attributes.

`dispHierarchy(`[*(bool)names=True*]`)` → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

`dispIndex`
Return class index of this instance.

`highlight(`=*false*`)`
Whether this Shape will be highlighted when rendered.

`normal(`=*Vector3r(NaN, NaN, NaN)*`)`
Facet's normal (in local coordinate system)

`setVertices(`*(Vector3)arg2, (Vector3)arg3, (Vector3)arg4*`)` → None
TODO

`updateAttrs(`*(dict)arg2*`)` → None
Update object attributes from given dictionary

`vertices(`=*vector<Vector3r>(3, Vector3r(NaN, NaN, NaN))*`)`
Vertex positions in local coordinates.

`wire(`=*false*`)`
Whether this Shape is rendered using color surfaces, or only wireframe (can still be overridden by global config of the renderer).

class `yade.wrapper.GridConnection(`*inherits Sphere → Shape → Serializable*`)`
GridConnection shape. Component of a grid designed to link two *GridNodes*. It's highly recommended to use utils.gridConnection(...) to generate correct *GridConnections*.

cellDist(=*Vector3i(0, 0, 0)*)
missing doc :(

color(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

highlight(=*false*)
Whether this Shape will be highlighted when rendered.

node1(=*uninitalized*)
First *Body* the GridConnection is connected to.

node2(=*uninitalized*)
Second *Body* the GridConnection is connected to.

periodic(=*false*)
true if two nodes from different periods are connected.

radius(=*NaN*)
Radius [m]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

wire(=*false*)
Whether this Shape is rendered using color surfaces, or only wireframe (can still be overridden by global config of the renderer).

class yade.wrapper.**GridNode**(*inherits Sphere → Shape → Serializable*)
GridNode shape, component of a grid. To create a Grid, place the nodes first, they will define the spacial discretisation of it. It's highly recommended to use utils.gridNode(...) to generate correct *GridNodes*. Note that the GridNodes should only be in an Interaction with other GridNodes. The Sphere-Grid contact is only handled by the *GridConnections*.

ConnList(=*uninitalized*)
List of *GridConnections* the GridNode is connected to.

addConnection(*(Body)Body*) → None
Add a GridConnection to the GridNode.

color(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

highlight(=*false*)
Whether this Shape will be highlighted when rendered.

radius(=*NaN*)
Radius [m]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

wire(=*false*)
Whether this Shape is rendered using color surfaces, or only wireframe (can still be overridden by global config of the renderer).

class yade.wrapper.**Polyhedra**(*inherits Shape → Serializable*)
Polyhedral (convex) geometry.

GetCentroid() → Vector3
return polyhedra's centroid

GetInertia() → Vector3
return polyhedra's inertia tensor

GetOri() → Quaternion
return polyhedra's orientation

GetSurfaceTriangulation() → object
triangulation of facets (for plotting)

GetSurfaces() → object
get indices of surfaces' vertices (for postprocessing)

GetVolume() → float
return polyhedra's volume

Initialize() → None
Initialization

color(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

highlight(=*false*)
Whether this Shape will be highlighted when rendered.

seed(=*time(__null)*)
Seed for random generator.

setVertices(*(object)arg2*) → None
set vertices and update receiver

size(=*Vector3r(1., 1., 1.)*)
Size of the grain in meters - x,y,z - before random rotation

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

v(=*uninitalized*)
Tetrahedron vertices in global coordinate system.

wire(=*false*)
Whether this Shape is rendered using color surfaces, or only wireframe (can still be overridden by global config of the renderer).

class `yade.wrapper.Sphere`(*inherits Shape → Serializable*)
Geometry of spherical particle.

`color`(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

`dict`() → dict
Return dictionary of attributes.

`dispHierarchy`([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

`dispIndex`
Return class index of this instance.

`highlight`(=*false*)
Whether this Shape will be highlighted when rendered.

`radius`(=*NaN*)
Radius [m]

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wire`(=*false*)
Whether this Shape is rendered using color surfaces, or only wireframe (can still be overridden by global config of the renderer).

class `yade.wrapper.Tetra`(*inherits Shape → Serializable*)
Tetrahedron geometry.

`color`(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

`dict`() → dict
Return dictionary of attributes.

`dispHierarchy`([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

`dispIndex`
Return class index of this instance.

`highlight`(=*false*)
Whether this Shape will be highlighted when rendered.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`v`(=*std::vector<Vector3r>(4)*)
Tetrahedron vertices (in local coordinate system).

`wire`(=*false*)
Whether this Shape is rendered using color surfaces, or only wireframe (can still be overridden by global config of the renderer).

class `yade.wrapper.Wall`(*inherits Shape → Serializable*)
Object representing infinite plane aligned with the coordinate system (axis-aligned wall).

`axis`(=*0*)
Axis of the normal; can be 0,1,2 for +x, +y, +z respectively (Body's orientation is disregarded for walls)

`color`(=*Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

`dict()` → dict
Return dictionary of attributes.

`dispHierarchy(`[*(bool)names=True*]`)` → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

`dispIndex`
Return class index of this instance.

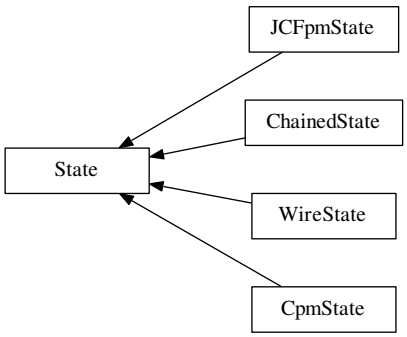
`highlight(`*=false*`)`
Whether this Shape will be highlighted when rendered.

`sense(`*=0*`)`
Which side of the wall interacts: -1 for negative only, 0 for both, +1 for positive only

`updateAttrs(`*(dict)arg2*`)` → None
Update object attributes from given dictionary

`wire(`*=false*`)`
Whether this Shape is rendered using color surfaces, or only wireframe (can still be overridden by global config of the renderer).

8.1.3 State



class `yade.wrapper.State(`*inherits Serializable*`)`
State of a body (spatial configuration, internal variables).

`angMom(`*=Vector3r::Zero()*`)`
Current angular momentum

`angVel(`*=Vector3r::Zero()*`)`
Current angular velocity

`blockedDOFs`
Degress of freedom where linear/angular velocity will be always constant (equal to zero, or to an user-defined value), regardless of applied force/torque. String that may contain 'xyzXYZ' (translations and rotations).

`densityScaling(`*=1*`)`
(auto-updated) see *GlobalStiffnessTimeStepper::targetDt*.

`dict()` → dict
Return dictionary of attributes.

`dispHierarchy(`[*(bool)names=True*]`)` → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

displ() → Vector3
Displacement from *reference position* (*pos* - *refPos*)

inertia(=*Vector3r::Zero()*)
Inertia of associated body, in local coordinate system.

isDamped(=*true*)
Damping in *Newtonintegrator* can be deactivated for individual particles by setting this variable to FALSE. E.g. damping is inappropriate for particles in free flight under gravity but it might still be applicable to other particles in the same simulation.

mass(=*0*)
Mass of this body

ori
Current orientation.

pos
Current position.

press
Returns the pressure (only for SPH-model).

refOri(=*Quaternionr::Identity()*)
Reference orientation

refPos(=*Vector3r::Zero()*)
Reference position

rho
Returns the current density (only for SPH-model).

rho0
Returns the rest density (only for SPH-model).

rot() → Vector3
Rotation from *reference orientation* (as rotation vector)

se3(=*Se3r(Vector3r::Zero(), Quaternionr::Identity())*)
Position and orientation as one object.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

vel(=*Vector3r::Zero()*)
Current linear velocity.

class yade.wrapper.**ChainedState**(*inherits State → Serializable*)
State of a chained bodies, containing information on connectivity in order to track contacts jumping over contiguous elements. Chains are 1D lists from which id of chained bodies are retrieved via *rank* and *chainNumber*.

addToChain(*(int)bodyId*) → None
Add body to current active chain

angMom(=*Vector3r::Zero()*)
Current angular momentum

angVel(=*Vector3r::Zero()*)
Current angular velocity

bId(=*-1*)
id of the body containing - for postLoad operations only.

blockedDOFs
Degress of freedom where linear/angular velocity will be always constant (equal to zero, or to

an user-defined value), regardless of applied force/torque. String that may contain 'xyzXYZ' (translations and rotations).

chainNumber(=*0*)
chain id.

currentChain = 0

densityScaling(=*1*)
(auto-updated) see *GlobalStiffnessTimeStepper::targetDt*.

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

displ() → Vector3
Displacement from *reference position* (*pos* - *refPos*)

inertia(=*Vector3r::Zero()*)
Inertia of associated body, in local coordinate system.

isDamped(=*true*)
Damping in *Newtonintegrator* can be deactivated for individual particles by setting this variable to FALSE. E.g. damping is inappropriate for particles in free flight under gravity but it might still be applicable to other particles in the same simulation.

mass(=*0*)
Mass of this body

ori
Current orientation.

pos
Current position.

press
Returns the pressure (only for SPH-model).

rank(=*0*)
rank in the chain.

refOri(=*Quaternionr::Identity()*)
Reference orientation

refPos(=*Vector3r::Zero()*)
Reference position

rho
Returns the current density (only for SPH-model).

rho0
Returns the rest density (only for SPH-model).

rot() → Vector3
Rotation from *reference orientation* (as rotation vector)

se3(=*Se3r(Vector3r::Zero(), Quaternionr::Identity())*)
Position and orientation as one object.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

vel(=*Vector3r::Zero()*)
Current linear velocity.

class `yade.wrapper.CpmState`(*inherits State → Serializable*)

State information about body use by *cpm-model*.

None of that is used for computation (at least not now), only for post-processing.

`angMom`(=*Vector3r::Zero()*)
: Current angular momentum

`angVel`(=*Vector3r::Zero()*)
: Current angular velocity

`blockedDOFs`
: Degress of freedom where linear/angular velocity will be always constant (equal to zero, or to an user-defined value), regardless of applied force/torque. String that may contain 'xyzXYZ' (translations and rotations).

`damageTensor`(=*Matrix3r::Zero()*)
: Damage tensor computed with microplane theory averaging. state.damageTensor.trace() = state.normDmg

`densityScaling`(=*1*)
: *(auto-updated)* see *GlobalStiffnessTimeStepper::targetDt*.

`dict`() → dict
: Return dictionary of attributes.

`dispHierarchy`(*(bool)names=True*) → list
: Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

`dispIndex`
: Return class index of this instance.

`displ`() → Vector3
: Displacement from *reference position* (*pos* - *refPos*)

`epsVolumetric`(=*0*)
: Volumetric strain around this body (unused for now)

`inertia`(=*Vector3r::Zero()*)
: Inertia of associated body, in local coordinate system.

`isDamped`(=*true*)
: Damping in *Newtonintegrator* can be deactivated for individual particles by setting this variable to FALSE. E.g. damping is inappropriate for particles in free flight under gravity but it might still be applicable to other particles in the same simulation.

`mass`(=*0*)
: Mass of this body

`normDmg`(=*0*)
: Average damage including already deleted contacts (it is really not damage, but 1-relResidualStrength now)

`numBrokenCohesive`(=*0*)
: Number of (cohesive) contacts that damaged completely

`numContacts`(=*0*)
: Number of contacts with this body

`ori`
: Current orientation.

`pos`
: Current position.

`press`
: Returns the pressure (only for SPH-model).

refOri(=*Quaternionr::Identity()*)
Reference orientation

refPos(=*Vector3r::Zero()*)
Reference position

rho
Returns the current density (only for SPH-model).

rho0
Returns the rest density (only for SPH-model).

rot() → Vector3
Rotation from *reference orientation* (as rotation vector)

se3(=*Se3r(Vector3r::Zero(), Quaternionr::Identity())*)
Position and orientation as one object.

stress(=*Matrix3r::Zero()*)
Stress tensor of the spherical particle (under assumption that particle volume = pi*r*r*r*4/3.) for packing fraction 0.62

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

vel(=*Vector3r::Zero()*)
Current linear velocity.

class yade.wrapper.**JCFpmState**(*inherits State → Serializable*)
JCFpm state information about each body.

angMom(=*Vector3r::Zero()*)
Current angular momentum

angVel(=*Vector3r::Zero()*)
Current angular velocity

blockedDOFs
Degress of freedom where linear/angular velocity will be always constant (equal to zero, or to an user-defined value), regardless of applied force/torque. String that may contain 'xyzXYZ' (translations and rotations).

densityScaling(=*1*)
(auto-updated) see *GlobalStiffnessTimeStepper::targetDt*.

dict() → dict
Return dictionary of attributes.

dispHierarchy(*[(bool)names=True]*) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

displ() → Vector3
Displacement from *reference position* (*pos* - *refPos*)

inertia(=*Vector3r::Zero()*)
Inertia of associated body, in local coordinate system.

isDamped(=*true*)
Damping in *Newtonintegrator* can be deactivated for individual particles by setting this variable to FALSE. E.g. damping is inappropriate for particles in free flight under gravity but it might still be applicable to other particles in the same simulation.

joint(=*0*)
Indicates the number of joint surfaces to which the particle belongs (0-> no joint, 1->1 joint, etc..). [-]

jointNormal1(=*Vector3r::Zero()*)
Specifies the normal direction to the joint plane 1. Rk: the ideal here would be to create a vector of vector wich size is defined by the joint integer (as much joint normals as joints). However, it needs to make the pushback function works with python since joint detection is done through a python script. lines 272 to 312 of cpp file should therefore be adapted. [-]

jointNormal2(=*Vector3r::Zero()*)
Specifies the normal direction to the joint plane 2. [-]

jointNormal3(=*Vector3r::Zero()*)
Specifies the normal direction to the joint plane 3. [-]

mass(=*0*)
Mass of this body

noIniLinks(=*0*)
Number of initial cohesive interactions. [-]

onJoint(=*false*)
Identifies if the particle is on a joint surface.

ori
Current orientation.

pos
Current position.

press
Returns the pressure (only for SPH-model).

refOri(=*Quaternionr::Identity()*)
Reference orientation

refPos(=*Vector3r::Zero()*)
Reference position

rho
Returns the current density (only for SPH-model).

rho0
Returns the rest density (only for SPH-model).

rot() → Vector3
Rotation from *reference orientation* (as rotation vector)

se3(=*Se3r(Vector3r::Zero(), Quaternionr::Identity())*)
Position and orientation as one object.

shearBreak(=*0*)
Number of shear breakages. [-]

shearBreakRel(=*0*)
Relative number (in [0;1], compared with *noIniLinks*) of shear breakages. [-]

tensBreak(=*0*)
Number of tensile breakages. [-]

tensBreakRel(=*0*)
Relative number (in [0;1], compared with *noIniLinks*) of tensile breakages. [-]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

vel(=*Vector3r::Zero()*)
Current linear velocity.

class yade.wrapper.**WireState**(*inherits State → Serializable*)
Wire state information of each body.

None of that is used for computation (at least not now), only for post-processing.

angMom(=*Vector3r::Zero()*)
Current angular momentum

angVel(=*Vector3r::Zero()*)
Current angular velocity

blockedDOFs
Degress of freedom where linear/angular velocity will be always constant (equal to zero, or to an user-defined value), regardless of applied force/torque. String that may contain 'xyzXYZ' (translations and rotations).

densityScaling(=*1*)
(auto-updated) see *GlobalStiffnessTimeStepper::targetDt*.

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

displ() → Vector3
Displacement from *reference position* (*pos* - *refPos*)

inertia(=*Vector3r::Zero()*)
Inertia of associated body, in local coordinate system.

isDamped(=*true*)
Damping in *Newtonintegrator* can be deactivated for individual particles by setting this variable to FALSE. E.g. damping is inappropriate for particles in free flight under gravity but it might still be applicable to other particles in the same simulation.

mass(=*0*)
Mass of this body

numBrokenLinks(=*0*)
Number of broken links (e.g. number of wires connected to the body which are broken). [-]

ori
Current orientation.

pos
Current position.

press
Returns the pressure (only for SPH-model).

refOri(=*Quaternionr::Identity()*)
Reference orientation

refPos(=*Vector3r::Zero()*)
Reference position

rho
Returns the current density (only for SPH-model).

rho0
Returns the rest density (only for SPH-model).

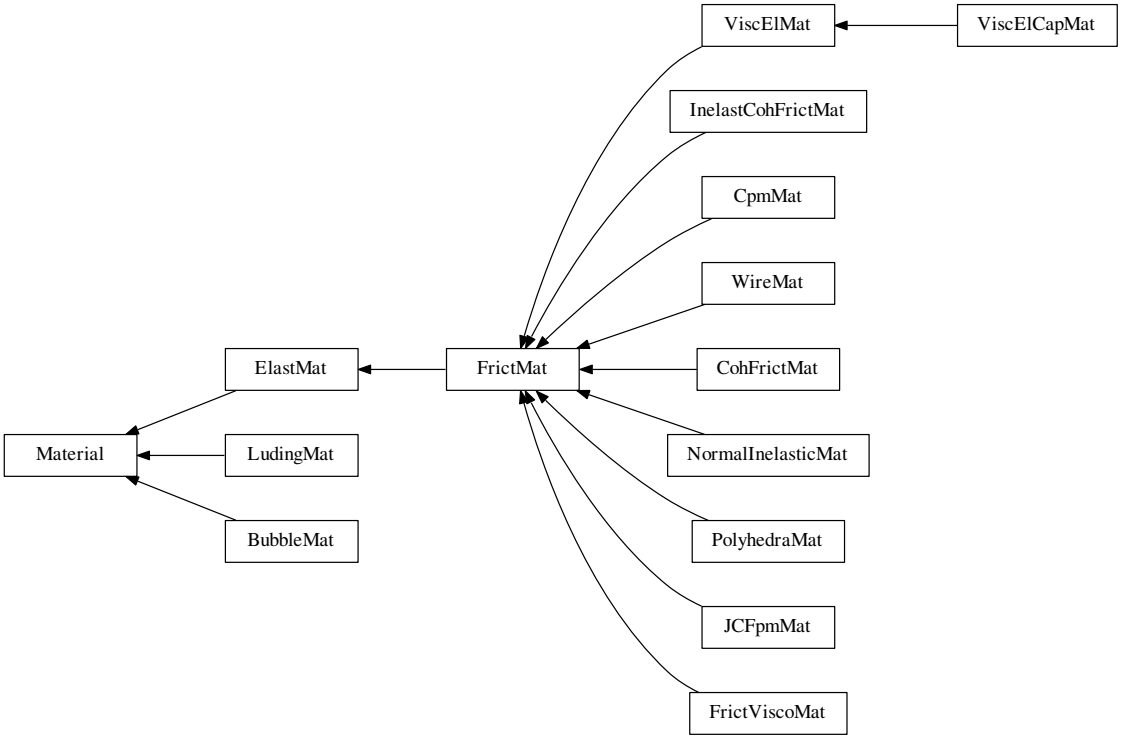
rot() → Vector3
Rotation from *reference orientation* (as rotation vector)

se3(=*Se3r(Vector3r::Zero(), Quaternionr::Identity())*)
Position and orientation as one object.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

vel(=*Vector3r::Zero()*)
Current linear velocity.

8.1.4 Material



class yade.wrapper.**Material**(*inherits Serializable*)
Material properties of a *body*.

density(=*1000*)
Density of the material [kg/m^3]

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

id(=*-1*, *not shared*)
Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via *O.materials.append*. (This id was necessary since before boost::serialization was used, shared pointers were not tracked properly; it might disappear in the future)

label(=*uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in *MaterialContainer*.

newAssocState() → State
Return new *State* instance, which is associated with this *Material*. Some materials have special requirement on *Body::state* type and calling this function when the body is created will ensure that they match. (This is done automatically if you use utils.sphere, … functions from python).

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**BubbleMat**(*inherits Material → Serializable*)
material for bubble interactions, for use with other Bubble classes

density(*=1000*)
Density of the material [kg/m^3]

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

id(*=-1, not shared*)
Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via *O.materials.append*. (This id was necessary since before boost::serialization was used, shared pointers were not tracked properly; it might disappear in the future)

label(*=uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in *MaterialContainer*.

newAssocState() → State
Return new *State* instance, which is associated with this *Material*. Some materials have special requirement on *Body::state* type and calling this function when the body is created will ensure that they match. (This is done automatically if you use utils.sphere, … functions from python).

surfaceTension(*=0.07197*)
The surface tension in the fluid surrounding the bubbles. The default value is that of water at 25 degrees Celcius.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**CohFrictMat**(*inherits FrictMat → ElastMat → Material → Serializable*)

alphaKr(*=2.0*)
Dimensionless rolling stiffness.

alphaKtw(*=2.0*)
Dimensionless twist stiffness.

density(*=1000*)
Density of the material [kg/m^3]

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

etaRoll(*=-1.*)
Dimensionless rolling (aka 'bending') strength. If negative, rolling moment will be elastic.

etaTwist(*=-1.*)
Dimensionless twisting strength. If negative, twist moment will be elastic.

fragile(*=true*)
do cohesion disappear when contact strength is exceeded

frictionAngle(*=.5*)
Contact friction angle (in radians). Hint : use 'radians(degreesValue)' in python scripts.

id(*=-1*, *not shared*)
Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via *O.materials.append*. (This id was necessary since before boost::serialization was used, shared pointers were not tracked properly; it might disappear in the future)

isCohesive(*=true*)

label(*=uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in *MaterialContainer*.

momentRotationLaw(*=false*)
Use bending/twisting moment at contact. The contact will have moments only if both bodies have this flag true. See *CohFrictPhys::cohesionDisablesFriction* for details.

newAssocState() → State
Return new *State* instance, which is associated with this *Material*. Some materials have special requirement on *Body::state* type and calling this function when the body is created will ensure that they match. (This is done automatically if you use utils.sphere, … functions from python).

normalCohesion(*=-1*)
Tensile strength, homogeneous to a pressure. If negative the normal force is purely elastic.

poisson(*=.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

shearCohesion(*=-1*)
Shear strength, homogeneous to a pressure. If negative the shear force is purely elastic.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(*=1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**CpmMat**(*inherits* *FrictMat* → *ElastMat* → *Material* → *Serializable*)
Concrete material, for use with other Cpm classes.

Note: *Density* is initialized to 4800 kgm ^{3}automatically, which gives approximate 2800 kgm ^{3} on 0.5 density packing.

Concrete Particle Model (CPM)

CpmMat is particle material, *Ip2_CpmMat_CpmMat_CpmPhys* averages two particles' materials, creating *CpmPhys*, which is then used in interaction resultion by *Law2_ScGeom_CpmPhys_Cpm*. *CpmState* is associated to *CpmMat* and keeps state defined on particles rather than interactions (such as number of completely damaged interactions).

The model is contained in externally defined macro CPM_MATERIAL_MODEL, which features damage in tension, plasticity in shear and compression and rate-dependence. For commercial reasons, rate-dependence and compression-plasticity is not present in reduced version of the model, used when CPM_MATERIAL_MODEL is not defined. The full model will be described in detail in my (Václav Šmilauer) thesis along with calibration procedures (rigidity, poisson's ratio, compressive/tensile strength ratio, fracture energy, behavior under confinement, rate-dependent behavior).

Even the public model is useful enough to run simulation on concrete samples, such as uniaxial tension-compression test.

damLaw(*=1*)
Law for damage evolution in uniaxial tension. 0 for linear stress-strain softening branch, 1 (default) for exponential damage evolution law

density(*=1000*)
Density of the material [kg/m^3]

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

dmgRateExp(*=0*)
Exponent for normal viscosity function. [-]

dmgTau(*=-1*, *deactivated if negative*)
Characteristic time for normal viscosity. [s]

epsCrackOnset(*=NaN*)
Limit elastic strain [-]

frictionAngle(*=.5*)
Contact friction angle (in radians). Hint : use 'radians(degreesValue)' in python scripts.

id(*=-1*, *not shared*)
Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via *O.materials.append*. (This id was necessary since before boost::serialization was used, shared pointers were not tracked properly; it might disappear in the future)

isoPrestress(*=0*)
Isotropic prestress of the whole specimen. [Pa]

label(*=uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in *MaterialContainer*.

neverDamage(*=false*)
If true, no damage will occur (for testing only).

newAssocState() → State
Return new *State* instance, which is associated with this *Material*. Some materials have special requirement on *Body::state* type and calling this function when the body is created will ensure that they match. (This is done automatically if you use utils.sphere, … functions from python).

plRateExp(*=0*)
Exponent for visco-plasticity function. [-]

plTau(*=-1*, *deactivated if negative*)
Characteristic time for visco-plasticity. [s]

poisson(=*.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

relDuctility(=*NaN*)
relative ductility of bonds in normal direction

sigmaT(=*NaN*)
Initial cohesion [Pa]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(=*1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**ElastMat**(*inherits Material → Serializable*)
Purely elastic material. The material parameters may have different meanings depending on the *IPhysFunctor* used : true Young and Poisson in *Ip2_FrictMat_FrictMat_MindlinPhys*, or contact stiffnesses in *Ip2_FrictMat_FrictMat_FrictPhys*.

density(=*1000*)
Density of the material [kg/m³]

dict() → dict
Return dictionary of attributes.

dispHierarchy(*[(bool)names=True]*) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

id(=*-1, not shared*)
Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via *O.materials.append*. (This id was necessary since before boost::serialization was used, shared pointers were not tracked properly; it might disappear in the future)

label(=*uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in *MaterialContainer*.

newAssocState() → State
Return new *State* instance, which is associated with this *Material*. Some materials have special requirement on *Body::state* type and calling this function when the body is created will ensure that they match. (This is done automatically if you use utils.sphere, … functions from python).

poisson(=*.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(=*1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**FrictMat**(*inherits ElastMat → Material → Serializable*)
Elastic material with contact friction. See also *ElastMat*.

density(=*1000*)
Density of the material [kg/m³]

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

frictionAngle(*=.5*)
Contact friction angle (in radians). Hint : use 'radians(degreesValue)' in python scripts.

id(*=-1, not shared*)
Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via *O.materials.append*. (This id was necessary since before boost::serialization was used, shared pointers were not tracked properly; it might disappear in the future)

label(*=uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in *MaterialContainer*.

newAssocState() → State
Return new *State* instance, which is associated with this *Material*. Some materials have special requirement on *Body::state* type and calling this function when the body is created will ensure that they match. (This is done automatically if you use utils.sphere, … functions from python).

poisson(*=.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(*=1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**FrictViscoMat**(*inherits FrictMat → ElastMat → Material → Serializable*)
Material for use with the FrictViscoPM classes

betan(*=0.*)
Fraction of the viscous damping coefficient in normal direction equal to $\frac{c_n}{C_{n,crit}}$.

density(*=1000*)
Density of the material [kg/m^3]

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

frictionAngle(*=.5*)
Contact friction angle (in radians). Hint : use 'radians(degreesValue)' in python scripts.

id(*=-1, not shared*)
Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via *O.materials.append*. (This id was necessary since before boost::serialization was used, shared pointers were not tracked properly; it might disappear in the future)

label(=*uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in *MaterialContainer*.

newAssocState() → State
Return new *State* instance, which is associated with this *Material*. Some materials have special requirement on *Body::state* type and calling this function when the body is created will ensure that they match. (This is done automatically if you use utils.sphere, … functions from python).

poisson(=*.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(=*1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**InelastCohFrictMat**(*inherits FrictMat → ElastMat → Material → Serializable*)

alphaKr(=*2.0*)
Dimensionless coefficient used for the rolling stiffness.

alphaKtw(=*2.0*)
Dimensionless coefficient used for the twist stiffness.

compressionModulus(=*0.0*)
Compresion elasticity modulus

creepBending(=*0.0*)
Bending creeping coefficient. Usual values between 0 and 1.

creepTension(=*0.0*)
Tension/compression creeping coefficient. Usual values between 0 and 1.

creepTwist(=*0.0*)
Twist creeping coefficient. Usual values between 0 and 1.

density(=*1000*)
Density of the material [kg/m^3]

dict() → dict
Return dictionary of attributes.

dispHierarchy(*[(bool)names=True]*) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

epsilonMaxCompression(=*0.0*)
Maximal plastic strain compression

epsilonMaxTension(=*0.0*)
Maximal plastic strain tension

etaMaxBending(=*0.0*)
Maximal plastic bending strain

etaMaxTwist(=*0.0*)
Maximal plastic twist strain

frictionAngle(=*.5*)
Contact friction angle (in radians). Hint : use 'radians(degreesValue)' in python scripts.

id(=*-1*, *not shared*)
Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via *O.materials.append*. (This id was necessary since before boost::serialization was used, shared pointers were not tracked properly; it might disappear in the future)

label(=*uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in *MaterialContainer*.

newAssocState() → State
Return new *State* instance, which is associated with this *Material*. Some materials have special requirement on *Body::state* type and calling this function when the body is created will ensure that they match. (This is done automatically if you use utils.sphere, … functions from python).

nuBending(=*0.0*)
Bending elastic stress limit

nuTwist(=*0.0*)
Twist elastic stress limit

poisson(=*.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

shearCohesion(=*0.0*)
Shear elastic stress limit

shearModulus(=*0.0*)
shear elasticity modulus

sigmaCompression(=*0.0*)
Compression elastic stress limit

sigmaTension(=*0.0*)
Tension elastic stress limit

tensionModulus(=*0.0*)
Tension elasticity modulus

unloadBending(=*0.0*)
Bending plastic unload coefficient. Usual values between 0 and +infinity.

unloadTension(=*0.0*)
Tension/compression plastic unload coefficient. Usual values between 0 and +infinity.

unloadTwist(=*0.0*)
Twist plastic unload coefficient. Usual values between 0 and +infinity.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(=*1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**JCFpmMat**(*inherits* *FrictMat* → *ElastMat* → *Material* → *Serializable*)
Possibly jointed, cohesive frictional material, for use with other JCFpm classes

cohesion(=*0.*)
Defines the maximum admissible tangential force in shear, for Fn=0, in the matrix (*FsMax* = cohesion * *crossSection*). [Pa]

density(=*1000*)
Density of the material [kg/m^3]

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

frictionAngle(*=.5*)
Contact friction angle (in radians). Hint : use 'radians(degreesValue)' in python scripts.

id(*=-1, not shared*)
Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via *O.materials.append*. (This id was necessary since before boost::serialization was used, shared pointers were not tracked properly; it might disappear in the future)

jointCohesion(*=0.*)
Defines the *maximum admissible tangential force in shear*, for Fn=0, on the joint surface. [Pa]

jointDilationAngle(*=0*)
Defines the dilatancy of the joint surface (only valid for *smooth contact logic*). [rad]

jointFrictionAngle(*=-1*)
Defines Coulomb friction on the joint surface. [rad]

jointNormalStiffness(*=0.*)
Defines the normal stiffness on the joint surface. [Pa/m]

jointShearStiffness(*=0.*)
Defines the shear stiffness on the joint surface. [Pa/m]

jointTensileStrength(*=0.*)
Defines the *maximum admissible normal force in traction* on the joint surface. [Pa]

label(*=uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in *MaterialContainer*.

newAssocState() → State
Return new *State* instance, which is associated with this *Material*. Some materials have special requirement on *Body::state* type and calling this function when the body is created will ensure that they match. (This is done automatically if you use utils.sphere, … functions from python).

poisson(*=.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

tensileStrength(*=0.*)
Defines the maximum admissible normal force in traction in the matrix (*FnMax* = tensileStrength * *crossSection*). [Pa]

type(*=0*)
If particles of two different types interact, it will be with friction only (no cohesion).[-]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(*=1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**LudingMat**(*inherits Material → Serializable*)
Material for simple Ludning's model of contact.

G0(*=NaN*)
Viscous damping

PhiF(=*NaN*)
Dimensionless plasticity depth

density(=*1000*)
Density of the material [kg/m^3]

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

frictionAngle(=*NaN*)
Friction angle [rad]

id(=*-1*, *not shared*)
Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via *O.materials.append*. (This id was necessary since before boost::serialization was used, shared pointers were not tracked properly; it might disappear in the future)

k1(=*NaN*)
Slope of loading plastic branch

kc(=*NaN*)
Slope of irreversible, tensile adhesive branch

kp(=*NaN*)
Slope of unloading and reloading limit elastic branch

label(=*uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in *MaterialContainer*.

newAssocState() → State
Return new *State* instance, which is associated with this *Material*. Some materials have special requirement on *Body::state* type and calling this function when the body is created will ensure that they match. (This is done automatically if you use utils.sphere, … functions from python).

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**NormalInelasticMat**(*inherits FrictMat → ElastMat → Material → Serializable*)
Material class for particles whose contact obey to a normal inelasticity (governed by this *coeff_dech*).

coeff_dech(=*1.0*)
=kn(unload) / kn(load)

density(=*1000*)
Density of the material [kg/m^3]

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

frictionAngle(=*.5*)
Contact friction angle (in radians). Hint : use 'radians(degreesValue)' in python scripts.

id(=*-1*, *not shared*)
Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via *O.materials.append*. (This id was necessary since before boost::serialization was used, shared pointers were not tracked properly; it might disappear in the future)

label(=*uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in *MaterialContainer*.

newAssocState() → State
Return new *State* instance, which is associated with this *Material*. Some materials have special requirement on *Body::state* type and calling this function when the body is created will ensure that they match. (This is done automatically if you use utils.sphere, … functions from python).

poisson(=*.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(=*1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**PolyhedraMat**(*inherits FrictMat → ElastMat → Material → Serializable*)
Elastic material with Coulomb friction.

IsSplitable(=*0*)
To be splitted ... or not

density(=*1000*)
Density of the material [kg/m^3]

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

frictionAngle(=*.5*)
Contact friction angle (in radians). Hint : use 'radians(degreesValue)' in python scripts.

id(=*-1*, *not shared*)
Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via *O.materials.append*. (This id was necessary since before boost::serialization was used, shared pointers were not tracked properly; it might disappear in the future)

label(=*uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in *MaterialContainer*.

newAssocState() → State
Return new *State* instance, which is associated with this *Material*. Some materials have special requirement on *Body::state* type and calling this function when the body is created

will ensure that they match. (This is done automatically if you use utils.sphere, … functions from python).

poisson(=*.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

strength(=*100*)
Stress at which polyhedra of volume 4/3*pi [mm] breaks.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(=*1e8*)
TODO

class yade.wrapper.**ViscElCapMat**(*inherits ViscElMat → FrictMat → ElastMat → Material → Serializable*)
Material for extended viscoelastic model of contact with capillary parameters.

Capillar(=*false*)
True, if capillar forces need to be added.

CapillarType(=*"“*)
Different types of capillar interaction: Willett_numeric, Willett_analytic *[Willett2000]* , Weigert *[Weigert1999]* , Rabinovich *[Rabinov2005]* , Lambert (simplified, corrected Rabinovich model) *[Lambert2008]*

KernFunctionPressure(=*Lucy*)
Kernel function for pressure calculation (by default - Lucy). The following kernel functions are available: Lucy=1.

KernFunctionVisco(=*Lucy*)
Kernel function for viscosity calculation (by default - Lucy). The following kernel functions are available: Lucy=1.

SPHmode(=*false*)
True, if SPH-mode is enabled.

Vb(=*0.0*)
Liquid bridge volume [m^3]

cn(=*NaN*)
Normal viscous constant. Attention, this parameter cannot be set if tc, en or es is defined!

cs(=*NaN*)
Shear viscous constant. Attention, this parameter cannot be set if tc, en or es is defined!

dcap(=*0.0*)
Damping coefficient for the capillary phase [-]

density(=*1000*)
Density of the material [kg/m³]

dict() → dict
Return dictionary of attributes.

dispHierarchy(*[(bool)names=True]*) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

en(=*NaN*)
Restitution coefficient in normal direction

et(=*NaN*)
Restitution coefficient in tangential direction

frictionAngle(=*.5*)
Contact friction angle (in radians). Hint : use 'radians(degreesValue)' in python scripts.

gamma(=*0.0*)
Surface tension [N/m]

h(=*-1*)
Core radius. See Mueller *[Mueller2003]* .

id(=*-1*, *not shared*)
Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via *O.materials.append*. (This id was necessary since before boost::serialization was used, shared pointers were not tracked properly; it might disappear in the future)

kn(=*NaN*)
Normal elastic stiffness. Attention, this parameter cannot be set if tc, en or es is defined!

ks(=*NaN*)
Shear elastic stiffness. Attention, this parameter cannot be set if tc, en or es is defined!

label(=*uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in *MaterialContainer*.

mR(=*0.0*)
Rolling resistance, see *[Zhou1999536]*.

mRtype(=*1*)
Rolling resistance type, see *[Zhou1999536]*. mRtype=1 - equation (3) in *[Zhou1999536]*; mRtype=2 - equation (4) in *[Zhou1999536]*.

mu(=*-1*)
Viscosity. See Mueller *[Morris1997]* .

newAssocState() → State
Return new *State* instance, which is associated with this *Material*. Some materials have special requirement on *Body::state* type and calling this function when the body is created will ensure that they match. (This is done automatically if you use utils.sphere, … functions from python).

poisson(=*.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

tc(=*NaN*)
Contact time

theta(=*0.0*)
Contact angle [°]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(=*1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**ViscElMat**(*inherits FrictMat → ElastMat → Material → Serializable*)
Material for simple viscoelastic model of contact from analytical solution of a pair spheres interaction problem *[Pournin2001]* .

KernFunctionPressure(=*Lucy*)
Kernel function for pressure calculation (by default - Lucy). The following kernel functions are available: Lucy=1.

KernFunctionVisco(=*Lucy*)
Kernel function for viscosity calculation (by default - Lucy). The following kernel functions are available: Lucy=1.

SPHmode(=*false*)
True, if SPH-mode is enabled.

cn(=*NaN*)
Normal viscous constant. Attention, this parameter cannot be set if tc, en or es is defined!

cs(=*NaN*)
Shear viscous constant. Attention, this parameter cannot be set if tc, en or es is defined!

density(=*1000*)
Density of the material [kg/m^3]

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

en(=*NaN*)
Restitution coefficient in normal direction

et(=*NaN*)
Restitution coefficient in tangential direction

frictionAngle(=*.5*)
Contact friction angle (in radians). Hint : use 'radians(degreesValue)' in python scripts.

h(=*-1*)
Core radius. See Mueller *[Mueller2003]* .

id(=*-1*, *not shared*)
Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via *O.materials.append*. (This id was necessary since before boost::serialization was used, shared pointers were not tracked properly; it might disappear in the future)

kn(=*NaN*)
Normal elastic stiffness. Attention, this parameter cannot be set if tc, en or es is defined!

ks(=*NaN*)
Shear elastic stiffness. Attention, this parameter cannot be set if tc, en or es is defined!

label(=*uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in *MaterialContainer*.

mR(=*0.0*)
Rolling resistance, see *[Zhou1999536]*.

mRtype(=*1*)
Rolling resistance type, see *[Zhou1999536]*. mRtype=1 - equation (3) in *[Zhou1999536]*; mRtype=2 - equation (4) in *[Zhou1999536]*.

mu(=*-1*)
Viscosity. See Mueller *[Morris1997]* .

newAssocState() → State
Return new *State* instance, which is associated with this *Material*. Some materials have special requirement on *Body::state* type and calling this function when the body is created will ensure that they match. (This is done automatically if you use utils.sphere, … functions from python).

poisson(=*.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

tc(=*NaN*)
Contact time

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(=*1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**WireMat**(*inherits FrictMat → ElastMat → Material → Serializable*)
Material for use with the Wire classes

as(=*0.*)
Cross-section area of a single wire used to transform stress into force. [m^2]

density(=*1000*)
Density of the material [kg/m^3]

diameter(=*0.0027*)
Diameter of the single wire in [m] (the diameter is used to compute the cross-section area of the wire).

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

frictionAngle(=*.5*)
Contact friction angle (in radians). Hint : use 'radians(degreesValue)' in python scripts.

id(=*-1*, *not shared*)
Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via *O.materials.append*. (This id was necessary since before boost::serialization was used, shared pointers were not tracked properly; it might disappear in the future)

isDoubleTwist(=*false*)
Type of the mesh. If true two particles of the same material which body ids differ by one will be considered as double-twisted interaction.

label(=*uninitalized*)
Textual identifier for this material; can be used for shared materials lookup in *MaterialContainer*.

lambdaEps(=*0.47*)
Parameter between 0 and 1 to reduce strain at failure of a double-twisted wire (as used by *[Bertrand2008]*). [-]

lambdaF(=*1.0*)
Parameter between 0 and 1 introduced by *[Thoeni2013]* which defines where the shifted force-displacement curve intersects with the new initial stiffness: $F^* = \lambda_F F_{elastic}$. [-]

lambdak(=*0.73*)
Parameter between 0 and 1 to compute the elastic stiffness of a double-twisted wire (as used by *[Bertrand2008]*): $k^D = 2(\lambda_k k_h + (1-\lambda_k)k^S)$. [-]

lambdau(=*0.2*)
Parameter between 0 and 1 introduced by *[Thoeni2013]* which defines the maximum shift

of the force-displacement curve in order to take an additional initial elongation (e.g. wire distortion/imperfections, slipping, system flexibility) into account: $\Delta l^* = \lambda_u l_0 \mathrm{rnd}(\mathrm{seed})$. [-]

newAssocState() → State
Return new *State* instance, which is associated with this *Material*. Some materials have special requirement on *Body::state* type and calling this function when the body is created will ensure that they match. (This is done automatically if you use utils.sphere, … functions from python).

poisson(=*.25*)
Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

seed(=*12345*)
Integer used to initialize the random number generator for the calculation of the distortion. If the integer is equal to 0 a internal seed number based on the time is computed. [-]

strainStressValues(=*uninitalized*)
Piecewise linear definition of the stress-strain curve by set of points (strain[-]>0,stress[Pa]>0) for one single wire. Tension only is considered and the point (0,0) is not needed! NOTE: Vector needs to be initialized!

strainStressValuesDT(=*uninitalized*)
Piecewise linear definition of the stress-strain curve by set of points (strain[-]>0,stress[Pa]>0) for the double twist. Tension only is considered and the point (0,0) is not needed! If this value is given the calculation will be based on two different stress-strain curves without considering the parameter introduced by *[Bertrand2008]* (see *[Thoeni2013]*).

type
Three different types are considered:

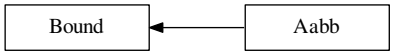
0	Corresponds to Bertrand's approach (see *[Bertrand2008]*): only one stress-strain curve is used
1	New approach: two separate stress-strain curves can be used (see *[Thoeni2013]*)
2	New approach with stochastically distorted contact model: two separate stress-strain curves with changed initial stiffness and horizontal shift (shift is random if seed ≥ 0, for more details see *[Thoeni2013]*)

By default the type is 0.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

young(=*1e9*)
elastic modulus [Pa]. It has different meanings depending on the Ip functor.

8.1.5 Bound



class yade.wrapper.**Bound**(*inherits Serializable*)
Object bounding part of space taken by associated body; might be larger, used to optimalize collision detection

color(=*Vector3r(1, 1, 1)*)
Color for rendering this object

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

lastUpdateIter(*=0*)
record iteration of last reference position update *(auto-updated)*

max(*=Vector3r(NaN, NaN, NaN)*)
Upper corner of box containing this bound (and the *Body* as well)

min(*=Vector3r(NaN, NaN, NaN)*)
Lower corner of box containing this bound (and the *Body* as well)

refPos(*=Vector3r(NaN, NaN, NaN)*)
Reference position, updated at current body position each time the bound dispatcher update bounds *(auto-updated)*

sweepLength(*=0*)
The length used to increase the bounding boxe size, can be adjusted on the basis of previous displacement if *BoundDispatcher::targetInterv*>0. *(auto-updated)*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Aabb**(*inherits Bound → Serializable*)
Axis-aligned bounding box, for use with *InsertionSortCollider*. (This class is quasi-redundant since min,max are already contained in *Bound* itself. That might change at some point, though.)

color(*=Vector3r(1, 1, 1)*)
Color for rendering this object

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

lastUpdateIter(*=0*)
record iteration of last reference position update *(auto-updated)*

max(*=Vector3r(NaN, NaN, NaN)*)
Upper corner of box containing this bound (and the *Body* as well)

min(*=Vector3r(NaN, NaN, NaN)*)
Lower corner of box containing this bound (and the *Body* as well)

refPos(*=Vector3r(NaN, NaN, NaN)*)
Reference position, updated at current body position each time the bound dispatcher update bounds *(auto-updated)*

sweepLength(*=0*)
The length used to increase the bounding boxe size, can be adjusted on the basis of previous displacement if *BoundDispatcher::targetInterv*>0. *(auto-updated)*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

8.2 Interactions

8.2.1 Interaction

class `yade.wrapper.Interaction`(*inherits Serializable*)

Interaction between pair of bodies.

`cellDist`

Distance of bodies in cell size units, if using periodic boundary conditions; id2 is shifted by this number of cells from its *State::pos* coordinates for this interaction to exist. Assigned by the collider.

> **Warning:** (internal) cellDist must survive Interaction::reset(), it is only initialized in ctor. Interaction that was cancelled by the constitutive law, was reset() and became only potential must have thepriod information if the geometric functor again makes it real. Good to know after few days of debugging that :-)

`dict()` → dict

Return dictionary of attributes.

`geom`(=*uninitalized*)

Geometry part of the interaction.

`id1`(=*0*)

Id of the first body in this interaction.

`id2`(=*0*)

Id of the second body in this interaction.

`isActive`

True if this interaction is active. Otherwise the forces from this interaction will not be taken into account. True by default.

`isReal`

True if this interaction has both geom and phys; False otherwise.

`iterBorn`(=*-1*)

Step number at which the interaction was added to simulation.

`iterMadeReal`(=*-1*)

Step number at which the interaction was fully (in the sense of geom and phys) created. (Should be touched only by *IPhysDispatcher* and *InteractionLoop*, therefore they are made friends of Interaction

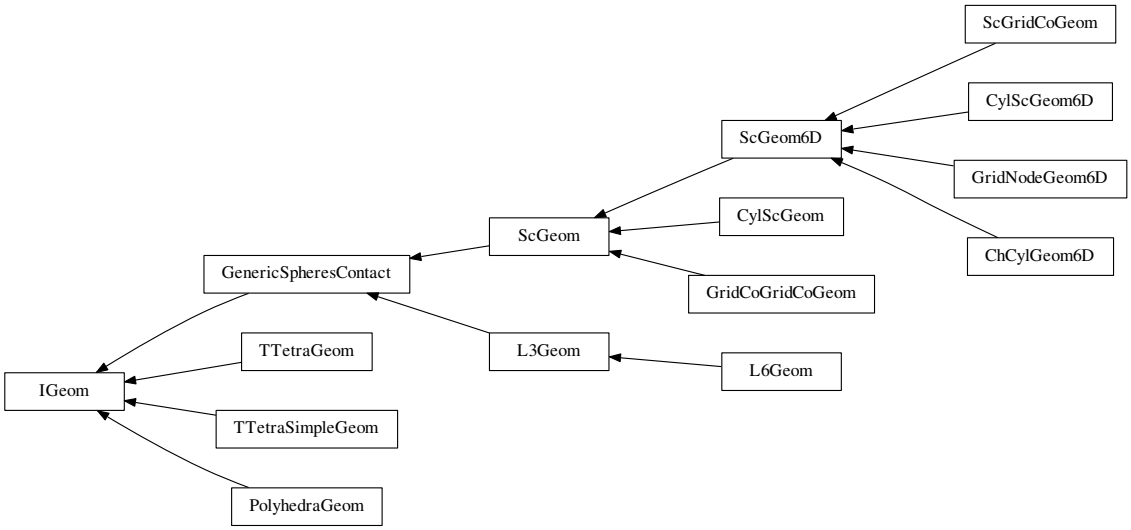
`phys`(=*uninitalized*)

Physical (material) part of the interaction.

`updateAttrs`(*(dict)arg2*) → None

Update object attributes from given dictionary

8.2.2 IGeom



class `yade.wrapper.IGeom`(*inherits Serializable*)

Geometrical configuration of interaction

`dict()` → dict

Return dictionary of attributes.

`dispHierarchy`([*(bool)names=True*]) → list

Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

`dispIndex`

Return class index of this instance.

`updateAttrs`(*(dict)arg2*) → None

Update object attributes from given dictionary

class `yade.wrapper.ChCylGeom6D`(*inherits ScGeom6D → ScGeom → GenericSpheresContact → IGeom → Serializable*)

Test

`bending`(=*Vector3r::Zero()*)

Bending at contact as a vector defining axis of rotation and angle (angle=norm).

`contactPoint`(=*uninitalized*)

some reference point for the interaction (usually in the middle). *(auto-computed)*

`dict()` → dict

Return dictionary of attributes.

`dispHierarchy`([*(bool)names=True*]) → list

Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

`dispIndex`

Return class index of this instance.

`incidentVel`(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3

Return incident velocity of the interaction (see also *Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting* for explanation of the ratcheting argument).

`initialOrientation1`(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)

Orientation of body 1 one at initialisation time *(auto-updated)*

initialOrientation2(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Orientation of body 2 one at initialisation time *(auto-updated)*

normal(=*uninitalized*)
Unit vector oriented along the interaction, from particle #1, towards particle #2. *(auto-updated)*

penetrationDepth(=*NaN*)
Penetration distance of spheres (positive if overlapping)

refR1(=*uninitalized*)
Reference radius of particle #1. *(auto-computed)*

refR2(=*uninitalized*)
Reference radius of particle #2. *(auto-computed)*

relAngVel(*(Interaction)i*) → Vector3
Return relative angular velocity of the interaction.

shearInc(=*Vector3r::Zero()*)
Shear displacement increment in the last step

twist(=*0*)
Elastic twist angle (around *normal axis*) of the contact.

twistCreep(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Stored creep, substracted from total relative rotation for computation of elastic moment *(auto-updated)*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**CylScGeom**(*inherits ScGeom → GenericSpheresContact → IGeom → Serializable*)
Geometry of a cylinder-sphere contact.

contactPoint(=*uninitalized*)
some reference point for the interaction (usually in the middle). *(auto-computed)*

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

end(=*Vector3r::Zero()*)
position of 2nd node *(auto-updated)*

id3(=*0*)
id of next chained cylinder *(auto-updated)*

incidentVel(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3
Return incident velocity of the interaction (see also *Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting* for explanation of the ratcheting argument).

isDuplicate(=*0*)
this flag is turned true (1) automatically if the contact is shared between two chained cylinders. A duplicated interaction will be skipped once by the constitutive law, so that only one contact at a time is effective. If isDuplicate=2, it means one of the two duplicates has no longer geometric interaction, and should be erased by the constitutive laws.

normal(=*uninitalized*)
Unit vector oriented along the interaction, from particle #1, towards particle #2. *(auto-updated)*

onNode(=*false*)
contact on node?

penetrationDepth(=*NaN*)
Penetration distance of spheres (positive if overlapping)

refR1(=*uninitalized*)
Reference radius of particle #1. *(auto-computed)*

refR2(=*uninitalized*)
Reference radius of particle #2. *(auto-computed)*

relAngVel(*(Interaction)i*) → Vector3
Return relative angular velocity of the interaction.

relPos(=*0*)
position of the contact on the cylinder (0: node-, 1:node+) *(auto-updated)*

shearInc(=*Vector3r::Zero()*)
Shear displacement increment in the last step

start(=*Vector3r::Zero()*)
position of 1st node *(auto-updated)*

trueInt(=*-1*)
Defines the body id of the cylinder where the contact is real, when *CylScGeom::isDuplicate*>0.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**CylScGeom6D**(*inherits ScGeom6D → ScGeom → GenericSpheresContact → IGeom → Serializable*)
Class representing *geometry* of two *bodies* in contact. The contact has 6 DOFs (normal, 2×shear, twist, 2xbending) and uses *ScGeom* incremental algorithm for updating shear.

bending(=*Vector3r::Zero()*)
Bending at contact as a vector defining axis of rotation and angle (angle=norm).

contactPoint(=*uninitalized*)
some reference point for the interaction (usually in the middle). *(auto-computed)*

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

end(=*Vector3r::Zero()*)
position of 2nd node *(auto-updated)*

id3(=*0*)
id of next chained cylinder *(auto-updated)*

incidentVel(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3
Return incident velocity of the interaction (see also *Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting* for explanation of the ratcheting argument).

initialOrientation1(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Orientation of body 1 one at initialisation time *(auto-updated)*

initialOrientation2(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Orientation of body 2 one at initialisation time *(auto-updated)*

isDuplicate(=*0*)
this flag is turned true (1) automatically if the contact is shared between two chained cylinders.

A duplicated interaction will be skipped once by the constitutive law, so that only one contact at a time is effective. If isDuplicate=2, it means one of the two duplicates has no longer geometric interaction, and should be erased by the constitutive laws.

normal(=*uninitalized*)
Unit vector oriented along the interaction, from particle #1, towards particle #2. *(auto-updated)*

onNode(=*false*)
contact on node?

penetrationDepth(=*NaN*)
Penetration distance of spheres (positive if overlapping)

refR1(=*uninitalized*)
Reference radius of particle #1. *(auto-computed)*

refR2(=*uninitalized*)
Reference radius of particle #2. *(auto-computed)*

relAngVel(*(Interaction)i*) → Vector3
Return relative angular velocity of the interaction.

relPos(=*0*)
position of the contact on the cylinder (0: node-, 1:node+) *(auto-updated)*

shearInc(=*Vector3r::Zero()*)
Shear displacement increment in the last step

start(=*Vector3r::Zero()*)
position of 1st node *(auto-updated)*

trueInt(=*-1*)
Defines the body id of the cylinder where the contact is real, when *CylScGeom::isDuplicate*>0.

twist(=*0*)
Elastic twist angle (around *normal axis*) of the contact.

twistCreep(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Stored creep, substracted from total relative rotation for computation of elastic moment *(auto-updated)*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**GenericSpheresContact**(*inherits IGeom → Serializable*)
Class uniting *ScGeom* and *L3Geom*, for the purposes of *GlobalStiffnessTimeStepper*. (It might be removed in the future). Do not use this class directly.

contactPoint(=*uninitalized*)
some reference point for the interaction (usually in the middle). *(auto-computed)*

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

normal(=*uninitalized*)
Unit vector oriented along the interaction, from particle #1, towards particle #2. *(auto-updated)*

refR1(=*uninitalized*)
Reference radius of particle #1. *(auto-computed)*

`refR2`(=*uninitalized*)
Reference radius of particle #2. *(auto-computed)*

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.GridCoGridCoGeom`(*inherits* *ScGeom* → *GenericSpheresContact* → *IGeom* → *Serializable*)
Geometry of a *GridConnection*-*GridConnection* contact.

`contactPoint`(=*uninitalized*)
some reference point for the interaction (usually in the middle). *(auto-computed)*

`dict`() → dict
Return dictionary of attributes.

`dispHierarchy`([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

`dispIndex`
Return class index of this instance.

`incidentVel`(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3
Return incident velocity of the interaction (see also *Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting* for explanation of the ratcheting argument).

`normal`(=*uninitalized*)
Unit vector oriented along the interaction, from particle #1, towards particle #2. *(auto-updated)*

`penetrationDepth`(=*NaN*)
Penetration distance of spheres (positive if overlapping)

`refR1`(=*uninitalized*)
Reference radius of particle #1. *(auto-computed)*

`refR2`(=*uninitalized*)
Reference radius of particle #2. *(auto-computed)*

`relAngVel`(*(Interaction)i*) → Vector3
Return relative angular velocity of the interaction.

`relPos1`(=*0*)
position of the contact on the first connection (0: node-, 1:node+) *(auto-updated)*

`relPos2`(=*0*)
position of the contact on the first connection (0: node-, 1:node+) *(auto-updated)*

`shearInc`(=*Vector3r::Zero()*)
Shear displacement increment in the last step

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.GridNodeGeom6D`(*inherits* *ScGeom6D* → *ScGeom* → *GenericSpheresContact* → *IGeom* → *Serializable*)
Geometry of a *GridNode*-*GridNode* contact. Inherits almost everything from *ScGeom6D*.

`bending`(=*Vector3r::Zero()*)
Bending at contact as a vector defining axis of rotation and angle (angle=norm).

`connectionBody`(=*uninitalized*)
Reference to the *GridNode Body* who is linking the two *GridNodes*.

`contactPoint`(=*uninitalized*)
some reference point for the interaction (usually in the middle). *(auto-computed)*

`dict`() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

incidentVel(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3
Return incident velocity of the interaction (see also *Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting* for explanation of the ratcheting argument).

initialOrientation1(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Orientation of body 1 one at initialisation time *(auto-updated)*

initialOrientation2(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Orientation of body 2 one at initialisation time *(auto-updated)*

normal(=*uninitalized*)
Unit vector oriented along the interaction, from particle #1, towards particle #2. *(auto-updated)*

penetrationDepth(=*NaN*)
Penetration distance of spheres (positive if overlapping)

refR1(=*uninitalized*)
Reference radius of particle #1. *(auto-computed)*

refR2(=*uninitalized*)
Reference radius of particle #2. *(auto-computed)*

relAngVel(*(Interaction)i*) → Vector3
Return relative angular velocity of the interaction.

shearInc(=*Vector3r::Zero()*)
Shear displacement increment in the last step

twist(=*0*)
Elastic twist angle (around *normal axis*) of the contact.

twistCreep(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Stored creep, substracted from total relative rotation for computation of elastic moment *(auto-updated)*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**L3Geom**(*inherits GenericSpheresContact → IGeom → Serializable*)
Geometry of contact given in local coordinates with 3 degress of freedom: normal and two in shear plane. [experimental]

F(=*Vector3r::Zero()*)
Applied force in local coordinates [debugging only, will be removed]

contactPoint(=*uninitalized*)
some reference point for the interaction (usually in the middle). *(auto-computed)*

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

`normal`(=*uninitalized*)
Unit vector oriented along the interaction, from particle #1, towards particle #2. *(auto-updated)*

`refR1`(=*uninitalized*)
Reference radius of particle #1. *(auto-computed)*

`refR2`(=*uninitalized*)
Reference radius of particle #2. *(auto-computed)*

`trsf`(=*Matrix3r::Identity()*)
Transformation (rotation) from global to local coordinates. (the translation part is in *GenericSpheresContact.contactPoint*)

`u`(=*Vector3r::Zero()*)
Displacement components, in local coordinates. *(auto-updated)*

`u0`
Zero displacement value; u0 should be always subtracted from the *geometrical* displacement *u* computed by appropriate *IGeomFunctor*, resulting in *u*. This value can be changed for instance

1. by *IGeomFunctor*, e.g. to take in account large shear displacement value unrepresentable by underlying geomeric algorithm based on quaternions)
2. by *LawFunctor*, to account for normal equilibrium position different from zero geometric overlap (set once, just after the interaction is created)
3. by *LawFunctor* to account for plastic slip.

Note: Never set an absolute value of *u0*, only increment, since both *IGeomFunctor* and *LawFunctor* use it. If you need to keep track of plastic deformation, store it in *IPhys* isntead (this might be changed: have *u0* for *LawFunctor* exclusively, and a separate value stored (when that is needed) inside classes deriving from *L3Geom*.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.L6Geom`(*inherits L3Geom → GenericSpheresContact → IGeom → Serializable*)
Geometric of contact in local coordinates with 6 degrees of freedom. [experimental]

`F`(=*Vector3r::Zero()*)
Applied force in local coordinates [debugging only, will be removed]

`contactPoint`(=*uninitalized*)
some reference point for the interaction (usually in the middle). *(auto-computed)*

`dict`() → dict
Return dictionary of attributes.

`dispHierarchy`([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

`dispIndex`
Return class index of this instance.

`normal`(=*uninitalized*)
Unit vector oriented along the interaction, from particle #1, towards particle #2. *(auto-updated)*

`phi`(=*Vector3r::Zero()*)
Rotation components, in local coordinates. *(auto-updated)*

`phi0`(=*Vector3r::Zero()*)
Zero rotation, should be always subtracted from *phi* to get the value. See *L3Geom.u0*.

refR1(=*uninitalized*)
Reference radius of particle #1. *(auto-computed)*

refR2(=*uninitalized*)
Reference radius of particle #2. *(auto-computed)*

trsf(=*Matrix3r::Identity()*)
Transformation (rotation) from global to local coordinates. (the translation part is in *GenericSpheresContact.contactPoint*)

u(=*Vector3r::Zero()*)
Displacement components, in local coordinates. *(auto-updated)*

u0
Zero displacement value; u0 should be always subtracted from the *geometrical* displacement u computed by appropriate *IGeomFunctor*, resulting in u. This value can be changed for instance

1. by *IGeomFunctor*, e.g. to take in account large shear displacement value unrepresentable by underlying geomeric algorithm based on quaternions)
2. by *LawFunctor*, to account for normal equilibrium position different from zero geometric overlap (set once, just after the interaction is created)
3. by *LawFunctor* to account for plastic slip.

Note: Never set an absolute value of *u0*, only increment, since both *IGeomFunctor* and *LawFunctor* use it. If you need to keep track of plastic deformation, store it in *IPhys* isntead (this might be changed: have *u0* for *LawFunctor* exclusively, and a separate value stored (when that is needed) inside classes deriving from *L3Geom*.

---

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**PolyhedraGeom**(*inherits IGeom → Serializable*)
Geometry of interaction between 2 *vector*, including volumetric characteristics

contactPoint(=*Vector3r::Zero()*)
Contact point (global coords), centriod of the overlapping polyhedron

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

equivalentCrossSection(=*NaN*)
Cross-section area of the overlap (perpendicular to the normal) - not used

equivalentPenetrationDepth(=*NaN*)
volume / equivalentCrossSection - not used

normal(=*Vector3r::Zero()*)
Normal direction of the interaction

orthonormal_axis(=*Vector3r::Zero()*)

penetrationVolume(=*NaN*)
Volume of overlap [m^3]

shearInc(=*Vector3r::Zero()*)
Shear displacement increment in the last step

twist_axis(=*Vector3r::Zero()*)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.ScGeom(*inherits GenericSpheresContact → IGeom → Serializable*)
Class representing *geometry* of a contact point between two *bodies*. It is more general than sphere-sphere contact even though it is primarily focused on spheres interactions (reason for the 'Sc' namming); it is also used for representing contacts of a *Sphere* with non-spherical bodies (*Facet*, *Plane*, *Box*, *ChainedCylinder*), or between two non-spherical bodies (*ChainedCylinder*). The contact has 3 DOFs (normal and 2×shear) and uses incremental algorithm for updating shear.

We use symbols $\mathbf{x}$, $\boldsymbol{\nu}$, $\boldsymbol{\omega}$ respectively for position, linear and angular velocities (all in global coordinates) and r for particles radii; subscripted with 1 or 2 to distinguish 2 spheres in contact. Then we define branch length and unit contact normal

$$l = \|\mathbf{x}_2 - \mathbf{x}_1\|, \mathbf{n} = \frac{\mathbf{x}_2 - \mathbf{x}_1}{\|\mathbf{x}_2 - \mathbf{x}_1\|}$$

The relative velocity of the spheres is then

$$\boldsymbol{\nu}_{12} = \frac{r_1 + r_2}{l}(\boldsymbol{\nu}_2 - \boldsymbol{\nu}_1) - (r_2\boldsymbol{\omega}_2 + r_1\boldsymbol{\omega}_1) \times \mathbf{n}$$

where the fraction multiplying translational velocities is to make the definition objective and avoid ratcheting effects (see *Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting*). The shear component is

$$\boldsymbol{\nu}^s_{12} = \boldsymbol{\nu}_{12} - (\mathbf{n} \cdot \boldsymbol{\nu}_{12})\mathbf{n}.$$

Tangential displacement increment over last step then reads

$$\Delta\mathbf{x}^s_{12} = \Delta t \boldsymbol{\nu}^s_{12}.$$

contactPoint(=*uninitalized*)
some reference point for the interaction (usually in the middle). *(auto-computed)*

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

incidentVel(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3
Return incident velocity of the interaction (see also *Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting* for explanation of the ratcheting argument).

normal(=*uninitalized*)
Unit vector oriented along the interaction, from particle #1, towards particle #2. *(auto-updated)*

penetrationDepth(=*NaN*)
Penetration distance of spheres (positive if overlapping)

refR1(=*uninitalized*)
Reference radius of particle #1. *(auto-computed)*

refR2(=*uninitalized*)
Reference radius of particle #2. *(auto-computed)*

relAngVel(*(Interaction)i*) → Vector3
Return relative angular velocity of the interaction.

shearInc(=*Vector3r::Zero()*)
Shear displacement increment in the last step

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**ScGeom6D**(*inherits ScGeom → GenericSpheresContact → IGeom → Serializable*)
Class representing *geometry* of two *bodies* in contact. The contact has 6 DOFs (normal, 2×shear, twist, 2xbending) and uses *ScGeom* incremental algorithm for updating shear.

bending(=*Vector3r::Zero()*)
Bending at contact as a vector defining axis of rotation and angle (angle=norm).

contactPoint(=*uninitalized*)
some reference point for the interaction (usually in the middle). *(auto-computed)*

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

incidentVel(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3
Return incident velocity of the interaction (see also *Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting* for explanation of the ratcheting argument).

initialOrientation1(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Orientation of body 1 one at initialisation time *(auto-updated)*

initialOrientation2(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Orientation of body 2 one at initialisation time *(auto-updated)*

normal(=*uninitalized*)
Unit vector oriented along the interaction, from particle #1, towards particle #2. *(auto-updated)*

penetrationDepth(=*NaN*)
Penetration distance of spheres (positive if overlapping)

refR1(=*uninitalized*)
Reference radius of particle #1. *(auto-computed)*

refR2(=*uninitalized*)
Reference radius of particle #2. *(auto-computed)*

relAngVel(*(Interaction)i*) → Vector3
Return relative angular velocity of the interaction.

shearInc(=*Vector3r::Zero()*)
Shear displacement increment in the last step

twist(=*0*)
Elastic twist angle (around *normal axis*) of the contact.

twistCreep(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Stored creep, substracted from total relative rotation for computation of elastic moment *(auto-updated)*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**ScGridCoGeom**(*inherits ScGeom6D → ScGeom → GenericSpheresContact → IGeom → Serializable*)
Geometry of a *GridConnection*-*Sphere* contact.

bending(=*Vector3r::Zero()*)
Bending at contact as a vector defining axis of rotation and angle (angle=norm).

contactPoint(=*uninitalized*)
some reference point for the interaction (usually in the middle). *(auto-computed)*

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

id3(=*0*)
id of the first *GridNode*. *(auto-updated)*

id4(=*0*)
id of the second *GridNode*. *(auto-updated)*

incidentVel(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3
Return incident velocity of the interaction (see also *Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting* for explanation of the ratcheting argument).

initialOrientation1(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Orientation of body 1 one at initialisation time *(auto-updated)*

initialOrientation2(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Orientation of body 2 one at initialisation time *(auto-updated)*

isDuplicate(=*0*)
this flag is turned true (1) automatically if the contact is shared between two Connections. A duplicated interaction will be skipped once by the constitutive law, so that only one contact at a time is effective. If isDuplicate=2, it means one of the two duplicates has no longer geometric interaction, and should be erased by the constitutive laws.

normal(=*uninitalized*)
Unit vector oriented along the interaction, from particle #1, towards particle #2. *(auto-updated)*

penetrationDepth(=*NaN*)
Penetration distance of spheres (positive if overlapping)

refR1(=*uninitalized*)
Reference radius of particle #1. *(auto-computed)*

refR2(=*uninitalized*)
Reference radius of particle #2. *(auto-computed)*

relAngVel(*(Interaction)i*) → Vector3
Return relative angular velocity of the interaction.

relPos(=*0*)
position of the contact on the connection (0: node-, 1:node+) *(auto-updated)*

shearInc(=*Vector3r::Zero()*)
Shear displacement increment in the last step

trueInt(=*-1*)
Defines the body id of the *GridConnection* where the contact is real, when *ScGridCoGeom::isDuplicate*>0.

twist(=*0*)
Elastic twist angle (around *normal axis*) of the contact.

twistCreep(=*Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Stored creep, substracted from total relative rotation for computation of elastic moment *(auto-updated)*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**TTetraGeom**(*inherits IGeom → Serializable*)
Geometry of interaction between 2 *tetrahedra*, including volumetric characteristics

contactPoint(=*uninitalized*)
Contact point (global coords)

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

equivalentCrossSection(=*NaN*)
Cross-section of the overlap (perpendicular to the axis of least inertia

equivalentPenetrationDepth(=*NaN*)
??

maxPenetrationDepthA(=*NaN*)
??

maxPenetrationDepthB(=*NaN*)
??

normal(=*uninitalized*)
Normal of the interaction, directed in the sense of least inertia of the overlap volume

penetrationVolume(=*NaN*)
Volume of overlap [m^3]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**TTetraSimpleGeom**(*inherits IGeom → Serializable*)
EXPERIMENTAL. Geometry of interaction between 2 *tetrahedra*

contactPoint(=*uninitalized*)
Contact point (global coords)

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

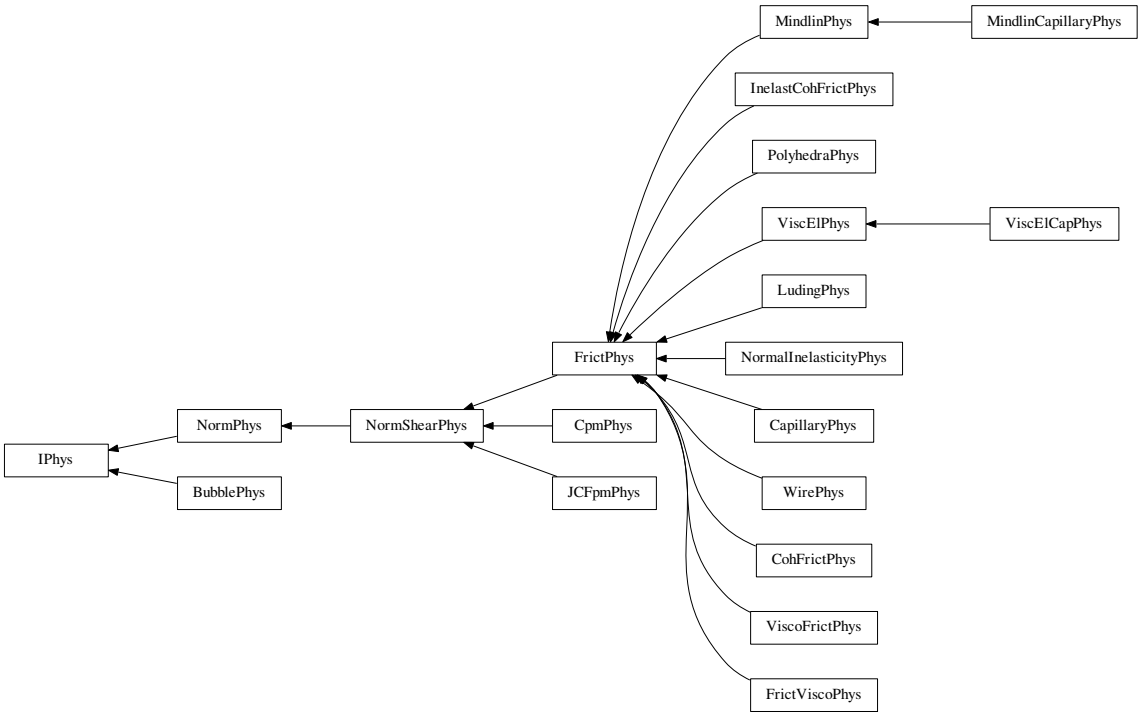
flag(=*0*)
TODO

normal(=*uninitalized*)
Normal of the interaction TODO

penetrationVolume(=*NaN*)
Volume of overlap [m^3]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

8.2.3 IPhys



class yade.wrapper.**IPhys**(*inherits Serializable*)
Physical (material) properties of *interaction*.

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**BubblePhys**(*inherits IPhys → Serializable*)
Physics of bubble-bubble interactions, for use with BubbleMat

Dmax(*=NaN*)
Maximum penetrationDepth of the bubbles before the force displacement curve changes to an artificial exponential curve. Setting this value will have no effect. See Law2_ScGeom_-BubblePhys_Bubble::pctMaxForce for more information

static computeForce(*(float)arg1*, *(float)arg2*, *(float)arg3*, *(int)arg4*, *(float)arg5*, *(float)arg6*, *(float)arg7*, *(BubblePhys)arg8*) → float :
Computes the normal force acting between the two interacting bubbles using the Newton-Rhapson method

dict() → dict
Return dictionary of attributes.

dispHierarchy(*[(bool)names=True]*) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

fN(*=NaN*)
Contact normal force

newtonIter(*=50*)
Maximum number of force iterations allowed

newtonTol(*=1e-6*)
Convergence criteria for force iterations

normalForce(*=Vector3r::Zero()*)
Normal force

rAvg(*=NaN*)
Average radius of the two interacting bubbles

surfaceTension(*=NaN*)
Surface tension of the surrounding liquid

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**CapillaryPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
Physics (of interaction) for *Law2_ScGeom_CapillaryPhys_Capillarity*.

Delta1(*=0.*)
Defines the surface area wetted by the meniscus on the smallest grains of radius R1 (R1<R2)

Delta2(*=0.*)
Defines the surface area wetted by the meniscus on the biggest grains of radius R2 (R1<R2)

capillaryPressure(*=0.*)
Value of the capillary pressure Uc. Defined as Ugas-Uliquid, obtained from *corresponding Law2 parameter*

dict() → dict
Return dictionary of attributes.

dispHierarchy(*[(bool)names=True]*) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

fCap(*=Vector3r::Zero()*)
Capillary force produced by the presence of the meniscus. This is the force acting on particle #2

fusionNumber(*=0.*)
Indicates the number of meniscii that overlap with this one

isBroken(*=false*)
Might be set to true by the user to make liquid bridge inactive (capillary force is zero)

kn(*=0*)
Normal stiffness

ks(*=0*)
Shear stiffness

meniscus(=*false*)
True when a meniscus with a non-zero liquid volume (*vMeniscus*) has been computed for this interaction

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

tangensOfFrictionAngle(=*NaN*)
tan of angle of friction

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

vMeniscus(=*0.*)
Volume of the meniscus

class yade.wrapper.**CohFrictPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)

cohesionBroken(=*true*)
is cohesion active? Set to false at the creation of a cohesive contact, and set to true when a fragile contact is broken

cohesionDisablesFriction(=*false*)
is shear strength the sum of friction and adhesion or only adhesion?

creep_viscosity(=*-1*)
creep viscosity [Pa.s/m].

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

fragile(=*true*)
do cohesion disappear when contact strength is exceeded?

initCohesion(=*false*)
Initialize the cohesive behaviour with current state as equilibrium state (same as *Ip2_CohFrictMat_CohFrictMat_CohFrictPhys::setCohesionNow* but acting on only one interaction)

kn(=*0*)
Normal stiffness

kr(=*0*)
rotational stiffness [N.m/rad]

ks(=*0*)
Shear stiffness

ktw(=*0*)
twist stiffness [N.m/rad]

maxRollPl(=*0.0*)
Coefficient of rolling friction (negative means elastic).

maxTwistPl(=*0.0*)
Coefficient of twisting friction (negative means elastic).

momentRotationLaw(=*false*)
use bending/twisting moment at contacts. See *Law2_ScGeom6D_CohFrictPhys_CohesionMoment::always_use_moment_law* for details.

moment_bending(=*Vector3r(0, 0, 0)*)
Bending moment

moment_twist(=*Vector3r(0, 0, 0)*)
Twist moment

normalAdhesion(=*0*)
tensile strength

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

shearAdhesion(=*0*)
cohesive part of the shear strength (a frictional term might be added depending on *CohFrictPhys::cohesionDisablesFriction*)

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

tangensOfFrictionAngle(=*NaN*)
tan of angle of friction

unp(=*0*)
plastic normal displacement, only used for tensile behaviour and if *CohFrictPhys::fragile* =false.

unpMax(=*0*)
maximum value of plastic normal displacement (counted positively), after that the interaction breaks even if *CohFrictPhys::fragile* =false. A negative value (i.e. -1) means no maximum.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**CpmPhys**(*inherits NormShearPhys → NormPhys → IPhys → Serializable*)
Representation of a single interaction of the Cpm type: storage for relevant parameters.

Evolution of the contact is governed by *Law2_ScGeom_CpmPhys_Cpm*, that includes damage effects and chages of parameters inside CpmPhys. See *cpm-model* for details.

E(=*NaN*)
normal modulus (stiffness / crossSection) [Pa]

Fn
Magnitude of normal force *(auto-updated)*

Fs
Magnitude of shear force *(auto-updated)*

G(=*NaN*)
shear modulus [Pa]

crossSection(=*NaN*)
equivalent cross-section associated with this contact [m^2]

cummBetaCount = **0**

cummBetaIter = **0**

damLaw(=*1*)
Law for softening part of uniaxial tension. 0 for linear, 1 for exponential (default)

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself,

top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

dmgOverstress(*=0*)
damage viscous overstress (at previous step or at current step)

dmgRateExp(*=0*)
exponent in the rate-dependent damage evolution

dmgStrain(*=0*)
damage strain (at previous or current step)

dmgTau(*=-1*)
characteristic time for damage (if non-positive, the law without rate-dependence is used)

epsCrackOnset(*=NaN*)
strain at which the material starts to behave non-linearly

epsFracture(*=NaN*)
strain at which the bond is fully broken [-]

epsN
Current normal strain *(auto-updated)*

epsNPl
normal plastic strain (initially zero) *(auto-updated)*

epsT
Current shear strain *(auto-updated)*

epsTPl
shear plastic strain (initially zero) *(auto-updated)*

static funcG(*(float)kappaD*, *(float)epsCrackOnset*, *(float)epsFracture*[, *(bool)neverDamage=False*[, *(int)damLaw=1*]]) → float :
Damage evolution law, evaluating the ω parameter. κ_D is historically maximum strain, *epsCrackOnset* (ε_0) = *CpmPhys.epsCrackOnset*, *epsFracture* = *CpmPhys.epsFracture*; if *neverDamage* is `True`, the value returned will always be 0 (no damage). TODO

static funcGInv(*(float)omega*, *(float)epsCrackOnset*, *(float)epsFracture*[, *(bool)neverDamage=False*[, *(int)damLaw=1*]]) → float :
Inversion of damage evolution law, evaluating the κ_D parameter. ω is damage, for other parameters see funcG function

isCohesive(*=false*)
if not cohesive, interaction is deleted when distance is greater than zero.

isoPrestress(*=0*)
"prestress" of this link (used to simulate isotropic stress)

kappaD
Up to now maximum normal strain (semi-norm), non-decreasing in time *(auto-updated)*

kn(*=0*)
Normal stiffness

ks(*=0*)
Shear stiffness

neverDamage(*=false*)
the damage evolution function will always return virgin state

normalForce(*=Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

omega
Damage internal variable *(auto-updated)*

plRateExp(*=0*)
exponent in the rate-dependent viscoplasticity

plTau(*=-1*)
characteristic time for viscoplasticity (if non-positive, no rate-dependence for shear)

refLength(*=NaN*)
initial length of interaction [m]

refPD(*=NaN*)
initial penetration depth of interaction [m] (used with ScGeom)

relDuctility(*=NaN*)
Relative ductility of bonds in normal direction

relResidualStrength
Relative residual strength *(auto-updated)*

setDamage(*(float)arg2*) → None
TODO

setRelResidualStrength(*(float)arg2*) → None
TODO

shearForce(*=Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

sigmaN
Current normal stress *(auto-updated)*

sigmaT
Current shear stress *(auto-updated)*

tanFrictionAngle(*=NaN*)
tangens of internal friction angle [-]

undamagedCohesion(*=NaN*)
virgin material cohesion [Pa]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**FrictPhys**(*inherits NormShearPhys → NormPhys → IPhys → Serializable*)
The simple linear elastic-plastic interaction with friction angle, like in the traditional *[CundallStrack1979]*

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

kn(*=0*)
Normal stiffness

ks(*=0*)
Shear stiffness

normalForce(*=Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

shearForce(*=Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

tangensOfFrictionAngle(=*NaN*)
tan of angle of friction

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**FrictViscoPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
Representation of a single interaction of the FrictViscoPM type, storage for relevant parameters

cn(=*NaN*)
Normal viscous constant defined as $\mathtt{n} = \mathtt{c}_{\mathtt{n,crit}}\beta_{\mathtt{n}}$.

cn_crit(=*NaN*)
Normal viscous constant for ctitical damping defined as $\mathtt{n} = \mathtt{C}_{\mathtt{n,crit}}\beta_{\mathtt{n}}$.

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

kn(=*0*)
Normal stiffness

ks(=*0*)
Shear stiffness

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

normalViscous(=*Vector3r::Zero()*)
Normal viscous component

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

tangensOfFrictionAngle(=*NaN*)
tan of angle of friction

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**InelastCohFrictPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)

cohesionBroken(=*false*)
is cohesion active? will be set false when a fragile contact is broken

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

isBroken(=*false*)
true if compression plastic fracture achieved

kDam(=*0*)
Damage coefficient on bending, computed from maximum bending moment reached and pure creep behaviour. Its values will vary between *InelastCohFrictPhys::kr* and *InelastCohFrictPhys::kRCrp* .

kRCrp(=*0.0*)
Bending creep stiffness

kRUnld(=*0.0*)
Bending plastic unload stiffness

kTCrp(=*0.0*)
Tension/compression creep stiffness

kTUnld(=*0.0*)
Tension/compression plastic unload stiffness

kTwCrp(=*0.0*)
Twist creep stiffness

kTwUnld(=*0.0*)
Twist plastic unload stiffness

kn(=*0*)
Normal stiffness

knC(=*0*)
compression stiffness

knT(=*0*)
tension stiffness

kr(=*0*)
bending stiffness

ks(=*0*)
shear stiffness

ktw(=*0*)
twist shear stiffness

maxBendMom(=*0.0*)
Plastic failure bending moment.

maxContract(=*0.0*)
Plastic failure contraction (shrinkage).

maxCrpRchdB(=*Vector3r(0, 0, 0)*)
maximal bending moment reached on plastic deformation.

maxCrpRchdC(=*Vector2r(0, 0)*)
maximal compression reached on plastic deformation. maxCrpRchdC[0] stores un and maxCrpRchdC[1] stores Fn.

maxCrpRchdT(=*Vector2r(0, 0)*)
maximal extension reached on plastic deformation. maxCrpRchdT[0] stores un and maxCrpRchdT[1] stores Fn.

maxCrpRchdTw(=*Vector2r(0, 0)*)
maximal twist reached on plastic deformation. maxCrpRchdTw[0] stores twist angle and maxCrpRchdTw[1] stores twist moment.

maxElB(=*0.0*)
Maximum bending elastic moment.

maxElC(=*0.0*)
Maximum compression elastic force.

maxElT(=*0.0*)
Maximum tension elastic force.

`maxElTw`(=*0.0*)
Maximum twist elastic moment.

`maxExten`(=*0.0*)
Plastic failure extension (stretching).

`maxTwist`(=*0.0*)
Plastic failure twist angle

`moment_bending`(=*Vector3r(0, 0, 0)*)
Bending moment

`moment_twist`(=*Vector3r(0, 0, 0)*)
Twist moment

`normalForce`(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

`onPlastB`(=*false*)
true if plasticity achieved on bending

`onPlastC`(=*false*)
true if plasticity achieved on compression

`onPlastT`(=*false*)
true if plasticity achieved on traction

`onPlastTw`(=*false*)
true if plasticity achieved on twisting

`pureCreep`(=*Vector3r(0, 0, 0)*)
Pure creep curve, used for comparison in calculation.

`shearAdhesion`(=*0*)
Maximum elastic shear force (cohesion).

`shearForce`(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

`tangensOfFrictionAngle`(=*NaN*)
tan of angle of friction

`twp`(=*0*)
plastic twist penetration depth describing the equilibrium state.

`unp`(=*0*)
plastic normal penetration depth describing the equilibrium state.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.JCFpmPhys`(*inherits NormShearPhys → NormPhys → IPhys → Serializable*)
Representation of a single interaction of the JCFpm type, storage for relevant parameters

`FnMax`(=*0*)
positiv value computed from *tensile strength* (or joint variant) to define the maximum admissible normal force in traction: Fn >= -FnMax. [N]

`FsMax`(=*0*)
computed from *cohesion* (or jointCohesion) to define the maximum admissible tangential force in shear, for Fn=0. [N]

`crackJointAperture`(=*0*)
Relative displacement between 2 spheres (in case of a crack it is equivalent of the crack aperture)

`crossSection`(=*0*)
crossSection=pi*Rmin^2. [m2]

`dict`() → dict
Return dictionary of attributes.

dilation(=*0*)
defines the normal displacement in the joint after sliding treshold. [m]

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

initD(=*0*)
equilibrium distance for interacting particles. Computed as the interparticular distance at first contact detection.

isBroken(=*false*)
flag for broken interactions

isCohesive(=*false*)
If false, particles interact in a frictional way. If true, particles are bonded regarding the given *cohesion* and *tensile strength* (or their jointed variants).

isOnJoint(=*false*)
defined as true when both interacting particles are *on joint* and are in opposite sides of the joint surface. In this case, mechanical parameters of the interaction are derived from the ''joint...'' material properties of the particles. Furthermore, the normal of the interaction may be re-oriented (see *Law2_ScGeom_JCFpmPhys_JointedCohesiveFrictionalPM.smoothJoint*).

jointCumulativeSliding(=*0*)
sliding distance for particles interacting on a joint. Used, when is true, to take into account dilatancy due to shearing. [-]

jointNormal(=*Vector3r::Zero()*)
normal direction to the joint, deduced from e.g. .

kn(=*0*)
Normal stiffness

ks(=*0*)
Shear stiffness

more(=*false*)
specifies if the interaction is crossed by more than 3 joints. If true, interaction is deleted (temporary solution).

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

tanDilationAngle(=*0*)
tangent of the angle defining the dilatancy of the joint surface (auto. computed from *JCFpmMat.jointDilationAngle*). [-]

tanFrictionAngle(=*0*)
tangent of Coulomb friction angle for this interaction (auto. computed). [-]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**LudingPhys**(*inherits* *FrictPhys* → *NormShearPhys* → *NormPhys* → *IPhys* → *Serializable*)
IPhys created from *LudingMat*, for use with *Law2_ScGeom_LudingPhys_Basic*.

DeltMax(=*NaN*)
Maximum overlap between particles for a collision

`DeltMin`(=*NaN*)
MinimalDelta value of delta

`DeltNull`(=*NaN*)
Force free overlap, plastic contact deformation

`DeltPMax`(=*NaN*)
Maximum overlap between particles for the limit case

`DeltPNull`(=*NaN*)
Max force free overlap, plastic contact deformation

`DeltPrev`(=*NaN*)
Previous value of delta

`G0`(=*NaN*)
Viscous damping

`PhiF`(=*NaN*)
Dimensionless plasticity depth

`dict`() → dict
Return dictionary of attributes.

`dispHierarchy`([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

`dispIndex`
Return class index of this instance.

`k1`(=*NaN*)
Slope of loading plastic branch

`k2`(=*NaN*)
Slope of unloading and reloading elastic branch

`kc`(=*NaN*)
Slope of irreversible, tensile adhesive branch

`kn`(=*0*)
Normal stiffness

`kp`(=*NaN*)
Slope of unloading and reloading limit elastic branch

`ks`(=*0*)
Shear stiffness

`normalForce`(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

`shearForce`(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

`tangensOfFrictionAngle`(=*NaN*)
tan of angle of friction

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.MindlinCapillaryPhys`(*inherits MindlinPhys → FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
Adds capillary physics to Mindlin's interaction physics.

`Delta1`(=*0.*)
Defines the surface area wetted by the meniscus on the smallest grains of radius R1 (R1<R2)

`Delta2`(=*0.*)
Defines the surface area wetted by the meniscus on the biggest grains of radius R2 (R1<R2)

Fs(=*Vector2r::Zero()*)
Shear force in local axes (computed incrementally)

adhesionForce(=*0.0*)
Force of adhesion as predicted by DMT

alpha(=*0.0*)
Constant coefficient to define contact viscous damping for non-linear elastic force-displacement relationship.

betan(=*0.0*)
Normal Damping Ratio. Fraction of the viscous damping coefficient (normal direction) equal to $\frac{c_n}{C_{n,crit}}$.

betas(=*0.0*)
Shear Damping Ratio. Fraction of the viscous damping coefficient (shear direction) equal to $\frac{c_s}{C_{s,crit}}$.

capillaryPressure(=*0.*)
Value of the capillary pressure Uc. Defined as Ugas-Uliquid, obtained from *corresponding Law2 parameter*

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

fCap(=*Vector3r::Zero()*)
Capillary Force produces by the presence of the meniscus. This is the force acting on particle #2

fusionNumber(=*0.*)
Indicates the number of meniscii that overlap with this one

isAdhesive(=*false*)
bool to identify if the contact is adhesive, that is to say if the contact force is attractive

isBroken(=*false*)
Might be set to true by the user to make liquid bridge inactive (capillary force is zero)

isSliding(=*false*)
check if the contact is sliding (useful to calculate the ratio of sliding contacts)

kn(=*0*)
Normal stiffness

kno(=*0.0*)
Constant value in the formulation of the normal stiffness

kr(=*0.0*)
Rotational stiffness

ks(=*0*)
Shear stiffness

kso(=*0.0*)
Constant value in the formulation of the tangential stiffness

ktw(=*0.0*)
Rotational stiffness

maxBendPl(=*0.0*)
Coefficient to determine the maximum plastic moment to apply at the contact

meniscus(=*false*)
True when a meniscus with a non-zero liquid volume (*vMeniscus*) has been computed for this interaction

momentBend(=*Vector3r::Zero()*)
Artificial bending moment to provide rolling resistance in order to account for some degree of interlocking between particles

momentTwist(=*Vector3r::Zero()*)
Artificial twisting moment (no plastic condition can be applied at the moment)

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

normalViscous(=*Vector3r::Zero()*)
Normal viscous component

prevU(=*Vector3r::Zero()*)
Previous local displacement; only used with *Law2_L3Geom_FrictPhys_HertzMindlin*.

radius(=*NaN*)
Contact radius (only computed with *Law2_ScGeom_MindlinPhys_Mindlin::calcEnergy*)

shearElastic(=*Vector3r::Zero()*)
Total elastic shear force

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

shearViscous(=*Vector3r::Zero()*)
Shear viscous component

tangensOfFrictionAngle(=*NaN*)
tan of angle of friction

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

usElastic(=*Vector3r::Zero()*)
Total elastic shear displacement (only elastic part)

usTotal(=*Vector3r::Zero()*)
Total elastic shear displacement (elastic+plastic part)

vMeniscus(=*0.*)
Volume of the meniscus

class yade.wrapper.**MindlinPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
Representation of an interaction of the Hertz-Mindlin type.

Fs(=*Vector2r::Zero()*)
Shear force in local axes (computed incrementally)

adhesionForce(=*0.0*)
Force of adhesion as predicted by DMT

alpha(=*0.0*)
Constant coefficient to define contact viscous damping for non-linear elastic force-displacement relationship.

betan(=*0.0*)
Normal Damping Ratio. Fraction of the viscous damping coefficient (normal direction) equal to $\frac{c_n}{C_{n,crit}}$.

betas(=*0.0*)
Shear Damping Ratio. Fraction of the viscous damping coefficient (shear direction) equal to $\frac{c_s}{C_{s,crit}}$.

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

isAdhesive(=*false*)
bool to identify if the contact is adhesive, that is to say if the contact force is attractive

isSliding(=*false*)
check if the contact is sliding (useful to calculate the ratio of sliding contacts)

kn(=*0*)
Normal stiffness

kno(=*0.0*)
Constant value in the formulation of the normal stiffness

kr(=*0.0*)
Rotational stiffness

ks(=*0*)
Shear stiffness

kso(=*0.0*)
Constant value in the formulation of the tangential stiffness

ktw(=*0.0*)
Rotational stiffness

maxBendPl(=*0.0*)
Coefficient to determine the maximum plastic moment to apply at the contact

momentBend(=*Vector3r::Zero()*)
Artificial bending moment to provide rolling resistance in order to account for some degree of interlocking between particles

momentTwist(=*Vector3r::Zero()*)
Artificial twisting moment (no plastic condition can be applied at the moment)

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

normalViscous(=*Vector3r::Zero()*)
Normal viscous component

prevU(=*Vector3r::Zero()*)
Previous local displacement; only used with *Law2_L3Geom_FrictPhys_HertzMindlin*.

radius(=*NaN*)
Contact radius (only computed with *Law2_ScGeom_MindlinPhys_Mindlin::calcEnergy*)

shearElastic(=*Vector3r::Zero()*)
Total elastic shear force

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

shearViscous(=*Vector3r::Zero()*)
Shear viscous component

tangensOfFrictionAngle(=*NaN*)
tan of angle of friction

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

usElastic(=*Vector3r::Zero()*)
Total elastic shear displacement (only elastic part)

usTotal(=*Vector3r::Zero()*)
Total elastic shear displacement (elastic+plastic part)

class yade.wrapper.**NormPhys**(*inherits IPhys → Serializable*)
Abstract class for interactions that have normal stiffness.

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

kn(=*0*)
Normal stiffness

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**NormShearPhys**(*inherits NormPhys → IPhys → Serializable*)
Abstract class for interactions that have shear stiffnesses, in addition to normal stiffness. This class is used in the PFC3d-style stiffness timestepper.

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

kn(=*0*)
Normal stiffness

ks(=*0*)
Shear stiffness

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**NormalInelasticityPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
Physics (of interaction) for using *Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity* : with inelastic unloadings

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

forMaxMoment(=*1.0*)
parameter stored for each interaction, and allowing to compute the maximum value of the exchanged torque : TorqueMax= forMaxMoment * NormalForce

kn(=*0*)
Normal stiffness

knLower(=*0.0*)
the stifness corresponding to a virgin load for example

kr(=*0.0*)
the rolling stiffness of the interaction

ks(=*0*)
Shear stiffness

moment_bending(=*Vector3r(0, 0, 0)*)
Bending moment. Defined here, being initialized as it should be, to be used in *Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity*

moment_twist(=*Vector3r(0, 0, 0)*)
Twist moment. Defined here, being initialized as it should be, to be used in *Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity*

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

previousFn(=*0.0*)
the value of the normal force at the last time step

previousun(=*0.0*)
the value of this un at the last time step

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

tangensOfFrictionAngle(=*NaN*)
tan of angle of friction

unMax(=*0.0*)
the maximum value of penetration depth of the history of this interaction

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**PolyhedraPhys**(*inherits* *FrictPhys* → *NormShearPhys* → *NormPhys* → *IPhys* → *Serializable*)
Simple elastic material with friction for volumetric constitutive laws

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

kn(=*0*)
Normal stiffness

ks(=*0*)
Shear stiffness

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

tangensOfFrictionAngle(=*NaN*)
tan of angle of friction

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**ViscElCapPhys**(*inherits* *ViscElPhys* → *FrictPhys* → *NormShearPhys* → *NormPhys* → *IPhys* → *Serializable*)
IPhys created from *ViscElCapMat*, for use with *Law2_ScGeom_ViscElCapPhys_Basic*.

Capillar(=*false*)
True, if capillar forces need to be added.

CapillarType(=*None_Capillar*)
Different types of capillar interaction: Willett_numeric, Willett_analytic, Weigert, Rabinovich, Lambert, Soulie

SPHmode(=*false*)
True, if SPH-mode is enabled.

Vb(=*0.0*)
Liquid bridge volume [m^3]

cn(=*NaN*)
Normal viscous constant

cs(=*NaN*)
Shear viscous constant

dcap(=*0.0*)
Damping coefficient for the capillary phase [-]

dict() → dict
Return dictionary of attributes.

dispHierarchy(*[(bool)names=True]*) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

gamma(=*0.0*)
Surface tension [N/m]

h(=*-1*)
Core radius. See Mueller *[Mueller2003]* .

kn(=*0*)
Normal stiffness

ks(=*0*)
Shear stiffness

liqBridgeActive(=*false*)
Whether liquid bridge is active at the moment

liqBridgeCreated(=*false*)
Whether liquid bridge was created, only after a normal contact of spheres

mR(=*0.0*)
Rolling resistance, see *[Zhou1999536]*.

mRtype(=*1*)
Rolling resistance type, see *[Zhou1999536]*. mRtype=1 - equation (3) in *[Zhou1999536]*; mRtype=2 - equation (4) in *[Zhou1999536]*

mu(=*-1*)
Viscosity. See Mueller *[Mueller2003]* .

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

sCrit(=*false*)
Critical bridge length [m]

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

tangensOfFrictionAngle(=*NaN*)
tan of angle of friction

theta(=*0.0*)
Contact angle [rad]

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**ViscElPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
IPhys created from *ViscElMat*, for use with *Law2_ScGeom_ViscElPhys_Basic*.

SPHmode(=*false*)
True, if SPH-mode is enabled.

cn(=*NaN*)
Normal viscous constant

cs(=*NaN*)
Shear viscous constant

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

h(=*-1*)
Core radius. See Mueller *[Mueller2003]* .

kn(=*0*)
Normal stiffness

ks(=*0*)
Shear stiffness

mR(=*0.0*)
Rolling resistance, see *[Zhou1999536]*.

mRtype(=*1*)
Rolling resistance type, see *[Zhou1999536]*. mRtype=1 - equation (3) in *[Zhou1999536]*; mRtype=2 - equation (4) in *[Zhou1999536]*

mu(=*-1*)
Viscosity. See Mueller *[Mueller2003]* .

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

tangensOfFrictionAngle(=*NaN*)
tan of angle of friction

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**ViscoFrictPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
Temporary version of *FrictPhys* for compatibility with e.g. *Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity*

creepedShear(=*Vector3r(0, 0, 0)*)
Creeped force (parallel)

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

kn(=*0*)
Normal stiffness

ks(=*0*)
Shear stiffness

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

tangensOfFrictionAngle(=*NaN*)
tan of angle of friction

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**WirePhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
Representation of a single interaction of the WirePM type, storage for relevant parameters

dL(=*0.*)
Additional wire length for considering the distortion for *WireMat* type=2 (see *[Thoeni2013]*).

dict() → dict
Return dictionary of attributes.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

displForceValues(=*uninitalized*)
Defines the values for force-displacement curve.

initD(=*0.*)
Equilibrium distance for particles. Computed as the initial inter-particular distance when particle are linked.

isDoubleTwist(=*false*)
If true the properties of the interaction will be defined as a double-twisted wire.

isLinked(=*false*)
If true particles are linked and will interact. Interactions are linked automatically by the definition of the corresponding interaction radius. The value is false if the wire breaks (no more interaction).

isShifted(=*false*)
If true *WireMat* type=2 and the force-displacement curve will be shifted.

kn(=*0*)
Normal stiffness

ks(=*0*)
Shear stiffness

limitFactor(=*0.*)
This value indicates on how far from failing the wire is, e.g. actual normal displacement divided by admissible normal displacement.

normalForce(=*Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

plastD
Plastic part of the inter-particular distance of the previous step.

Note: Only elastic displacements are reversible (the elastic stiffness is used for unloading) and compressive forces are inadmissible. The compressive stiffness is assumed to be equal to zero.

---

shearForce(=*Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

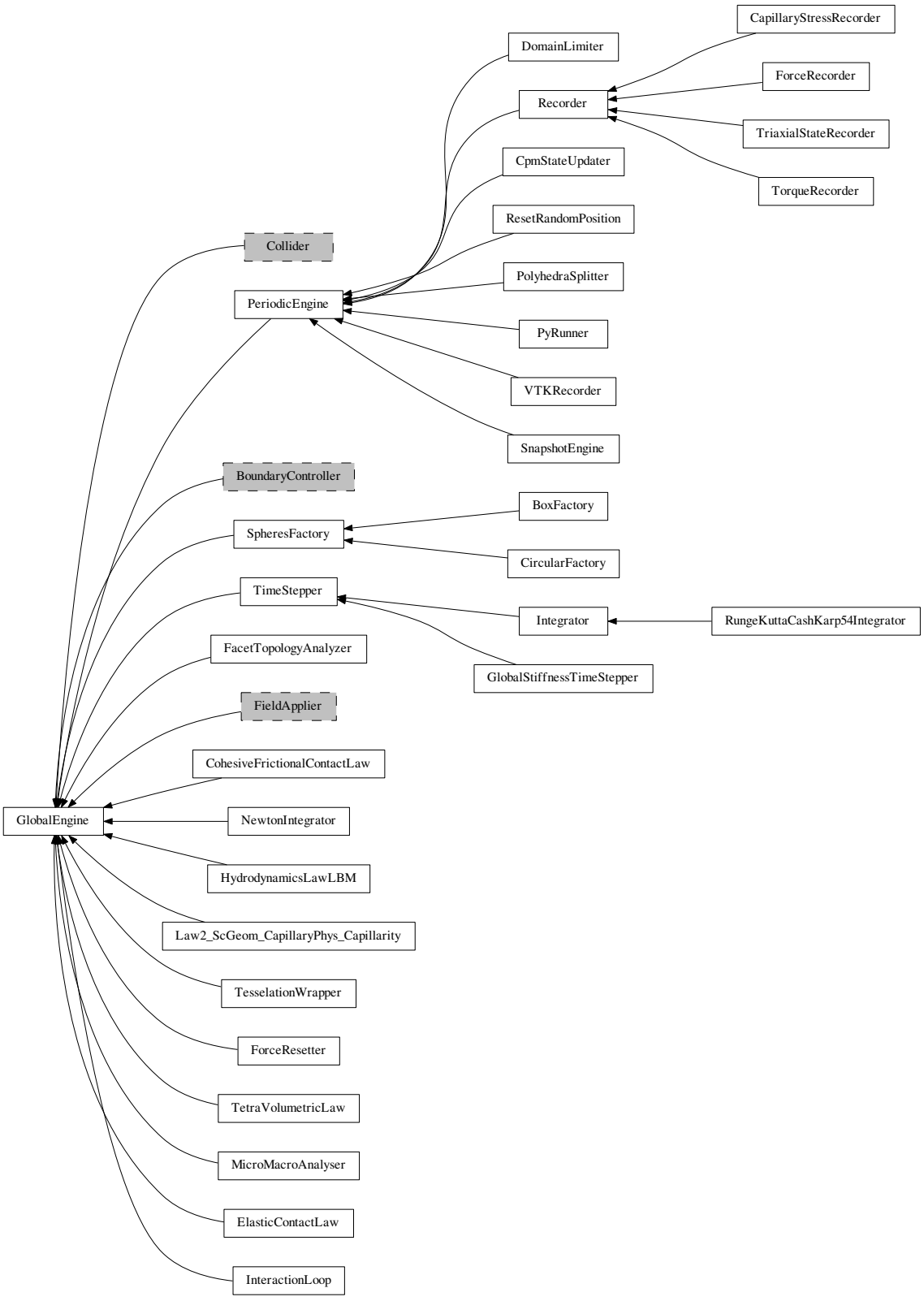
stiffnessValues(=*uninitalized*)
Defines the values for the various stiffnesses (the elastic stiffness is stored as kn).

tangensOfFrictionAngle(=*NaN*)
tan of angle of friction

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

8.3 Global engines

8.3.1 GlobalEngine



class `yade.wrapper.GlobalEngine`(*inherits Engine → Serializable*)

Engine that will generally affect the whole simulation (contrary to PartialEngine).

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**BoxFactory**(*inherits SpheresFactory → GlobalEngine → Engine → Serializable*)
Box geometry of the SpheresFactory region, given by extents and center

PSDcalculateMass(=*true*)
PSD-Input is in mass (true), otherwise the number of particles will be considered.

PSDcum(=*uninitalized*)
PSD-dispersion, cumulative procent meanings [-]

PSDsizes(=*uninitalized*)
PSD-dispersion, sizes of cells, Diameter [m]

blockedDOFs(=*""*)
Blocked degress of freedom

center(=*Vector3r(NaN, NaN, NaN)*)
Center of the region

color(=*Vector3r(-1, -1, -1)*)
Use the color for newly created particles, if specified

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

exactDiam(=*true*)
If true, the particles only with the defined in PSDsizes diameters will be created. Otherwise the diameter will be randomly chosen in the range [PSDsizes[i-1]:PSDsizes[i]], in this case the length of PSDsizes should be more on 1, than the length of PSDcum.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

extents(=*Vector3r(NaN, NaN, NaN)*)
Extents of the region

goalMass(=*0*)
Total mass that should be attained at the end of the current step. *(auto-updated)*

ids(=*uninitalized*)
ids of created bodies

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

mask(=*-1*)
groupMask to apply for newly created spheres

massFlowRate(=*NaN*)
Mass flow rate [kg/s]

materialId(=*-1*)
Shared material id to use for newly created spheres (can be negative to count from the end)

maxAttempt(=*5000*)
Maximum number of attempts to position a new sphere randomly.

maxMass(=*-1*)
Maximal mass at which to stop generating new particles regardless of massFlowRate. if maxMass=-1 - this parameter is ignored.

maxParticles(=*100*)
The number of particles at which to stop generating new ones regardless of massFlowRate. if maxParticles=-1 - this parameter is ignored .

normal(=*Vector3r(NaN, NaN, NaN)*)
Orientation of the region's geometry, direction of particle's velocites if normalVel is not set.

normalVel(=*Vector3r(NaN, NaN, NaN)*)
Direction of particle's velocites.

numParticles(=*0*)
Cummulative number of particles produces so far *(auto-updated)*

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

rMax(=*NaN*)
Maximum radius of generated spheres (uniform distribution)

rMin(=*NaN*)
Minimum radius of generated spheres (uniform distribution)

silent(=*false*)
If true no complain about excessing maxAttempt but disable the factory (by set massFlowRate=0).

stopIfFailed(=*true*)
If true, the SpheresFactory stops (sets massFlowRate=0), when maximal number of attempts to insert particle exceed.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

totalMass(*=0*)
Mass of spheres that was produced so far. *(auto-updated)*

totalVolume(*=0*)
Volume of spheres that was produced so far. *(auto-updated)*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

vAngle(*=NaN*)
Maximum angle by which the initial sphere velocity deviates from the normal.

vMax(*=NaN*)
Maximum velocity norm of generated spheres (uniform distribution)

vMin(*=NaN*)
Minimum velocity norm of generated spheres (uniform distribution)

class yade.wrapper.**CapillaryStressRecorder**(*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable*)
Records information from capillary meniscii on samples submitted to triaxial compressions. Classical sign convention (tension positiv) is used for capillary stresses. -> New formalism needs to be tested!!!

addIterNum(*=false*)
Adds an iteration number to the file name, when the file was created. Useful for creating new files at each call (false by default)

dead(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

file(*=uninitalized*)
Name of file to save to; must not be empty.

firstIterRun(*=0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

initRun(*=false*)
Run the first time we are called as well.

iterLast(*=0*)
Tracks step number of last run *(auto-updated)*.

iterPeriod(*=0*, *deactivated*)
Periodicity criterion using step number (deactivated if <= 0)

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

nDo(*=-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

nDone(*=0*)
Track number of executions (cummulative) *(auto-updated)*.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

realLast(=*0*)
Tracks real time of last run *(auto-updated)*.

realPeriod(=*0*, *deactivated*)
Periodicity criterion using real (wall clock, computation, human) time (deactivated if <=0)

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

truncate(=*false*)
Whether to delete current file contents, if any, when opening (false by default)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

virtLast(=*0*)
Tracks virtual time of last run *(auto-updated)*.

virtPeriod(=*0*, *deactivated*)
Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

class yade.wrapper.**CircularFactory**(*inherits* *SpheresFactory* → *GlobalEngine* → *Engine* → *Serializable*)
Circular geometry of the SpheresFactory region. It can be disk (given by radius and center), or cylinder (given by radius, length and center).

PSDcalculateMass(=*true*)
PSD-Input is in mass (true), otherwise the number of particles will be considered.

PSDcum(=*uninitalized*)
PSD-dispersion, cumulative procent meanings [-]

PSDsizes(=*uninitalized*)
PSD-dispersion, sizes of cells, Diameter [m]

blockedDOFs(="")
Blocked degress of freedom

center(=*Vector3r(NaN, NaN, NaN)*)
Center of the region

color(=*Vector3r(-1, -1, -1)*)
Use the color for newly created particles, if specified

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

exactDiam(=*true*)
If true, the particles only with the defined in PSDsizes diameters will be created. Otherwise the diameter will be randomly chosen in the range [PSDsizes[i-1]:PSDsizes[i]], in this case the length of PSDsizes should be more on 1, than the length of PSDcum.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

goalMass(=*0*)
Total mass that should be attained at the end of the current step. *(auto-updated)*

ids(=*uninitalized*)
ids of created bodies

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

length(=*0*)
Length of the cylindrical region (0 by default)

mask(=*-1*)
groupMask to apply for newly created spheres

massFlowRate(=*NaN*)
Mass flow rate [kg/s]

materialId(=*-1*)
Shared material id to use for newly created spheres (can be negative to count from the end)

maxAttempt(=*5000*)
Maximum number of attempts to position a new sphere randomly.

maxMass(=*-1*)
Maximal mass at which to stop generating new particles regardless of massFlowRate. if maxMass=-1 - this parameter is ignored.

maxParticles(=*100*)
The number of particles at which to stop generating new ones regardless of massFlowRate. if maxParticles=-1 - this parameter is ignored .

normal(=*Vector3r(NaN, NaN, NaN)*)
Orientation of the region's geometry, direction of particle's velocites if normalVel is not set.

normalVel(=*Vector3r(NaN, NaN, NaN)*)
Direction of particle's velocites.

numParticles(=*0*)
Cummulative number of particles produces so far *(auto-updated)*

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

rMax(=*NaN*)
Maximum radius of generated spheres (uniform distribution)

rMin(=*NaN*)
Minimum radius of generated spheres (uniform distribution)

radius(=*NaN*)
Radius of the region

silent(=*false*)
If true no complain about excessing maxAttempt but disable the factory (by set massFlowRate=0).

stopIfFailed(=*true*)
If true, the SpheresFactory stops (sets massFlowRate=0), when maximal number of attempts to insert particle exceed.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

totalMass(*=0*)
Mass of spheres that was produced so far. *(auto-updated)*

totalVolume(*=0*)
Volume of spheres that was produced so far. *(auto-updated)*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

vAngle(*=NaN*)
Maximum angle by which the initial sphere velocity deviates from the normal.

vMax(*=NaN*)
Maximum velocity norm of generated spheres (uniform distribution)

vMin(*=NaN*)
Minimum velocity norm of generated spheres (uniform distribution)

class yade.wrapper.**CohesiveFrictionalContactLaw**(*inherits GlobalEngine → Engine → Serializable*)
[DEPRECATED] Loop over interactions applying *Law2_ScGeom6D_CohFrictPhys_CohesionMoment* on all interactions.

Note: Use *InteractionLoop* and *Law2_ScGeom6D_CohFrictPhys_CohesionMoment* instead of this class for performance reasons.

---

always_use_moment_law(*=false*)
If true, use bending/twisting moments at all contacts. If false, compute moments only for cohesive contacts.

creep_viscosity(*=false*)
creep viscosity [Pa.s/m]. probably should be moved to Ip2_CohFrictMat_CohFrictMat_-CohFrictPhys...

dead(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

neverErase(*=false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. *Law2_ScGeom_CapillaryPhys_Capillarity*)

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

shear_creep(=*false*)
activate creep on the shear force, using *CohesiveFrictionalContactLaw::creep_viscosity*.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

twist_creep(=*false*)
activate creep on the twisting moment, using *CohesiveFrictionalContactLaw::creep_viscosity*.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**CpmStateUpdater**(*inherits* *PeriodicEngine* → *GlobalEngine* → *Engine* → *Serializable*)
Update *CpmState* of bodies based on state variables in *CpmPhys* of interactions with this bod. In particular, bodies' colors and *CpmState::normDmg* depending on average *damage* of their interactions and number of interactions that were already fully broken and have disappeared is updated. This engine contains its own loop (2 loops, more precisely) over all bodies and should be run periodically to update colors during the simulation, if desired.

avgRelResidual(=*NaN*)
Average residual strength at last run.

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

firstIterRun(=*0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

initRun(=*false*)
Run the first time we are called as well.

iterLast(=*0*)
Tracks step number of last run *(auto-updated)*.

iterPeriod(=*0*, *deactivated*)
Periodicity criterion using step number (deactivated if <= 0)

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxOmega(=*NaN*)
Globally maximum damage parameter at last run.

nDo(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

nDone(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes

openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

realLast(*=0*)
Tracks real time of last run *(auto-updated)*.

realPeriod(*=0*, *deactivated*)
Periodicity criterion using real (wall clock, computation, human) time (deactivated if <=0)

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

virtLast(*=0*)
Tracks virtual time of last run *(auto-updated)*.

virtPeriod(*=0*, *deactivated*)
Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

class yade.wrapper.**DomainLimiter**(*inherits* *PeriodicEngine* → *GlobalEngine* → *Engine* → *Serializable*)
Delete particles that are out of axis-aligned box given by *lo* and *hi*.

dead(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

firstIterRun(*=0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

hi(*=Vector3r(0, 0, 0)*)
Upper corner of the domain.

initRun(*=false*)
Run the first time we are called as well.

iterLast(*=0*)
Tracks step number of last run *(auto-updated)*.

iterPeriod(*=0*, *deactivated*)
Periodicity criterion using step number (deactivated if <= 0)

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

lo(*=Vector3r(0, 0, 0)*)
Lower corner of the domain.

mDeleted(*=0*)
Mass of deleted particles.

mask(*=-1*)
If mask is defined, only particles with corresponding groupMask will be deleted.

nDeleted(*=0*)
Cummulative number of particles deleted.

`nDo`(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

`nDone`(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`realLast`(=*0*)
Tracks real time of last run *(auto-updated)*.

`realPeriod`(=*0*, *deactivated*)
Periodicity criterion using real (wall clock, computation, human) time (deactivated if <=0)

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`vDeleted`(=*0*)
Volume of deleted particles.

`virtLast`(=*0*)
Tracks virtual time of last run *(auto-updated)*.

`virtPeriod`(=*0*, *deactivated*)
Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

class `yade.wrapper.ElasticContactLaw`(*inherits* *GlobalEngine* → *Engine* → *Serializable*)
[DEPRECATED] Loop over interactions applying *Law2_ScGeom_FrictPhys_CundallStrack* on all interactions.

Note: Use *InteractionLoop* and *Law2_ScGeom_FrictPhys_CundallStrack* instead of this class for performance reasons.

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`neverErase`(=*false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. *Law2_ScGeom_CapillaryPhys_Capillarity*)

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can

depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**FacetTopologyAnalyzer**(*inherits GlobalEngine → Engine → Serializable*)
Initializer for filling adjacency geometry data for facets.

Common vertices and common edges are identified and mutual angle between facet faces is written to Facet instances. If facets don't move with respect to each other, this must be done only at the beginng.

commonEdgesFound(*=0*)
how many common edges were identified during last run. *(auto-updated)*

commonVerticesFound(*=0*)
how many common vertices were identified during last run. *(auto-updated)*

dead(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

projectionAxis(*=Vector3r::UnitX()*)
Axis along which to do the initial vertex sort

relTolerance(*=1e-4*)
maximum distance of 'identical' vertices, relative to minimum facet size

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**ForceRecorder**(*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable*)
Engine saves the resultant force affecting to bodies, listed in *ids*. For instance, can be useful for defining the forces, which affects to _buldozer_ during its work.

addIterNum(=*false*)
Adds an iteration number to the file name, when the file was created. Useful for creating new files at each call (false by default)

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

file(=*uninitalized*)
Name of file to save to; must not be empty.

firstIterRun(=*0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

ids(=*uninitalized*)
List of bodies whose state will be measured

initRun(=*false*)
Run the first time we are called as well.

iterLast(=*0*)
Tracks step number of last run *(auto-updated)*.

iterPeriod(=*0*, *deactivated*)
Periodicity criterion using step number (deactivated if <= 0)

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

nDo(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

nDone(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

realLast(=*0*)
Tracks real time of last run *(auto-updated)*.

realPeriod(=*0*, *deactivated*)
Periodicity criterion using real (wall clock, computation, human) time (deactivated if <=0)

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

totalForce(=*Vector3r::Zero()*)
Resultant force, returning by the function.

truncate(=*false*)
Whether to delete current file contents, if any, when opening (false by default)

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`virtLast`(*=0*)
Tracks virtual time of last run *(auto-updated)*.

`virtPeriod`(*=0*, *deactivated*)
Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

class `yade.wrapper.ForceResetter`(*inherits GlobalEngine → Engine → Serializable*)
Reset all forces stored in Scene::forces (`O.forces` in python). Typically, this is the first engine to be run at every step. In addition, reset those energies that should be reset, if energy tracing is enabled.

`dead`(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`label`(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.GlobalStiffnessTimeStepper`(*inherits TimeStepper → GlobalEngine → Engine → Serializable*)
An engine assigning the time-step as a fraction of the minimum eigen-period in the problem. The derivation is detailed in the chapter on DEM formulation. The viscEl option enables to evaluate the timestep in a similar way for the visco-elastic contact law *Law2_ScGeom_ViscElPhys_Basic*, more detail in *GlobalStiffnessTimestepper::viscEl*.

`active`(*=true*)
is the engine active?

`dead`(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`defaultDt`(*=-1*)
used as the initial value of the timestep (especially useful in the first steps when no contact exist). If negative, it will be defined by *utils.PWaveTimeStep* * *GlobalStiffnessTimeStepper::timestepSafetyCoefficient*

`densityScaling`(*=false*)
(auto-updated) don't modify this value if you don't plan to modify the scaling factor manually

for some bodies. In most cases, it is enough to set *NewtonIntegrator::densityScaling* and let this one be adjusted automatically.

`dict()` → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`maxDt`(=*Mathr::MAX_REAL*)
if positive, used as max value of the timestep whatever the computed value

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`previousDt`(=*1*)
last computed dt *(auto-updated)*

`targetDt`(=*1*)
if *NewtonIntegrator::densityScaling* is active, this value will be used as the simulation timestep and the scaling will use this value of dt as the target value. The value of targetDt is arbitrary and should have no effect in the result in general. However if some bodies have imposed velocities, for instance, they will move more or less per each step depending on this value.

`timeStepUpdateInterval`(=*1*)
dt update interval

`timestepSafetyCoefficient`(=*0.8*)
safety factor between the minimum eigen-period and the final assigned dt (less than 1)

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`viscEl`(=*false*)
To use with *ViscElPhys*. if True, evaluate separetly the minimum eigen-period in the problem considering only the elastic contribution on one hand (spring only), and only the viscous contribution on the other hand (dashpot only). Take then the minimum of the two and use the safety coefficient *GlobalStiffnessTimestepper::timestepSafetyCoefficient* to take into account the possible coupling between the two contribution.

class `yade.wrapper.HydrodynamicsLawLBM`(*inherits* *GlobalEngine* → *Engine* → *Serializable*)
Engine to simulate fluid flow (with the lattice Boltzmann method) with a coupling with the discrete element method. If you use this Engine, please cite and refer to F. Lominé et al. International Journal For Numerical and Analytical Method in Geomechanics, 2012, doi: 10.1002/nag.1109

`ConvergenceThreshold`(=*0.000001*)

`CstBodyForce`(=*Vector3r::Zero()*)
A constant body force (=that does not vary in time or space, otherwise the implementation introduces errors)

DemIterLbmIterRatio(=*-1*)
Ratio between DEM and LBM iterations for subcycling

EndTime(=*-1*)
the time to stop the simulation

EngineIsActivated(=*true*)
To activate (or not) the engine

IterMax(=*1*)
This variable can be used to do several LBM iterations during one DEM iteration.

IterPrint(=*1*)
Print info on screen every IterPrint iterations

IterSave(=*100*)
Data are saved every IterSave LBM iteration (or see TimeSave)

IterSubCyclingStart(=*-1*)
Iteration number when the subcycling process starts

LBMSavedData(=" ")
a list of data that will be saved. Can use velocity,velXY,forces,rho,bodies,nodeBD,newNode,observedptc,observednode,contacts,spheres,bz2

Nu(=*0.000001*)
Fluid kinematic viscosity

Nx(=*1000*)
The number of grid division in x direction

ObservedNode(=*-1*)
The identifier of the node that will be observed (-1 means none)

ObservedPtc(=*-1*)
The identifier of the particle that will be observed (-1 means the first one)

RadFactor(=*1.0*)
The radius of DEM particules seen by the LBM is the real radius of particules*RadFactor

Rho(=*1000.*)
Fluid density

SaveGridRatio(=*1*)
Grid data are saved every SaveGridRatio * IterSave LBM iteration (with SaveMode=1)

SaveMode(=*1*)
Save Mode (1-> default, 2-> in time (not yet implemented)

TimeSave(=*-1*)
Data are saved at constant time interval (or see IterSave)

VbCutOff(=*-1*)
the minimum boundary velocity that is taken into account

VelocityThreshold(=*-1.*)
Velocity threshold when removingCriterion=2

WallXm_id(=*2*)
Identifier of the X- wall

WallXp_id(=*3*)
Identifier of the X+ wall

WallYm_id(=*0*)
Identifier of the Y- wall

WallYp_id(=*1*)
Identifier of the Y+ wall

`WallZm_id`(=*4*)
Identifier of the Z- wall

`WallZp_id`(=*5*)
Identifier of the Z+ wall

`XmBCType`(=*1*)
Boundary condition for the wall in Xm (-1: unused, 1: pressure condition, 2: velocity condition).

`XmBcRho`(=*-1*)
(!!! not fully implemented !!) The density imposed at the boundary

`XmBcVel`(=*Vector3r::Zero()*)
(!!! not fully implemented !!) The velocity imposed at the boundary

`XmYmZmBCType`(=*-1*)
Boundary condition for the corner node XmYmZm (not used with d2q9, -1: unused, 1: pressure condition, 2: velocity condition).

`XmYmZpBCType`(=*2*)
Boundary condition for the corner node XmYmZp (-1: unused, 1: pressure condition, 2: velocity condition).

`XmYpZmBCType`(=*-1*)
Boundary condition for the corner node XmYpZm (not used with d2q9, -1: unused, 1: pressure condition, 2: velocity condition).

`XmYpZpBCType`(=*2*)
Boundary condition for the corner node XmYpZp (-1: unused, 1: pressure condition, 2: velocity condition).

`XpBCType`(=*1*)
Boundary condition for the wall in Xp (-1: unused, 1: pressure condition, 2: velocity condition).

`XpBcRho`(=*-1*)
(!!! not fully implemented !!) The density imposed at the boundary

`XpBcVel`(=*Vector3r::Zero()*)
(!!! not fully implemented !!) The velocity imposed at the boundary

`XpYmZmBCType`(=*-1*)
Boundary condition for the corner node XpYmZm (not used with d2q9, -1: unused, 1: pressure condition, 2: velocity condition).

`XpYmZpBCType`(=*2*)
Boundary condition for the corner node XpYmZp (-1: unused, 1: pressure condition, 2: velocity condition).

`XpYpZmBCType`(=*-1*)
Boundary condition for the corner node XpYpZm (not used with d2q9, -1: unused, 1: pressure condition, 2: velocity condition).

`XpYpZpBCType`(=*2*)
Boundary condition for the corner node XpYpZp (-1: unused, 1: pressure condition, 2: velocity condition).

`YmBCType`(=*2*)
Boundary condition for the wall in Ym (-1: unused, 1: pressure condition, 2: velocity condition).

`YmBcRho`(=*-1*)
(!!! not fully implemented !!) The density imposed at the boundary

`YmBcVel`(=*Vector3r::Zero()*)
(!!! not fully implemented !!) The velocity imposed at the boundary

YpBCType(=*2*)
Boundary condition for the wall in Yp (-1: unused, 1: pressure condition, 2: velocity condition).

YpBcRho(=*-1*)
(!!! not fully implemented !!) The density imposed at the boundary

YpBcVel(=*Vector3r::Zero()*)
(!!! not fully implemented !!) The velocity imposed at the boundary

ZmBCType(=*-1*)
Boundary condition for the wall in Zm (-1: unused, 1: pressure condition, 2: velocity condition).

ZmBcRho(=*-1*)
(!!! not fully implemented !!) The density imposed at the boundary

ZmBcVel(=*Vector3r::Zero()*)
(!!! not fully implemented !!) The velocity imposed at the boundary

ZpBCType(=*-1*)
Boundary condition for the wall in Zp (-1: unused, 1: pressure condition, 2: velocity condition).

ZpBcVel(=*Vector3r::Zero()*)
(!!! not fully implemented !!) The velocity imposed at the boundary

applyForcesAndTorques(=*true*)
Switch to apply forces and torques

bc(=*" "*)
Boundary condition

dP(=*Vector3r(0., 0., 0.)*)
Pressure difference between input and output

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

defaultLbmInitMode(=*0*)
Switch between the two initialisation methods

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

model(=*"d2q9"*)
The LB model. Until now only d2q9 is implemented

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

periodicity(=*" "*)
periodicity

`removingCriterion`(*=0*)
Criterion to remove a sphere (1->based on particle position, 2->based on particle velocity

`tau`(*=0.6*)
Relaxation time

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`useWallXm`(*=false*)
Set true if you want that the LBM see the wall in Xm

`useWallXp`(*=false*)
Set true if you want that the LBM see the wall in Xp

`useWallYm`(*=true*)
Set true if you want that the LBM see the wall in Ym

`useWallYp`(*=true*)
Set true if you want that the LBM see the wall in Yp

`useWallZm`(*=false*)
Set true if you want that the LBM see the wall in Zm

`useWallZp`(*=false*)
Set true if you want that the LBM see the wall in Zp

`zpBcRho`(*=-1*)
(!!! not fully implemented !!) The density imposed at the boundary

class `yade.wrapper.Integrator`(*inherits TimeStepper → GlobalEngine → Engine → Serializable*)
Integration Engine Interface.

`active`(*=true*)
is the engine active?

`dead`(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`integrationsteps`(*=uninitalized*)
all integrationsteps count as all succesfull substeps

`label`(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`maxVelocitySq`(*=NaN*)
store square of max. velocity, for informative purposes; computed again at every step. *(auto-updated)*

`ompThreads`(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes

openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

slaves
List of lists of Engines to calculate the force acting on the particles; to obtain the derivatives of the states, engines inside will be run sequentially.

timeStepUpdateInterval(=*1*)
dt update interval

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.InteractionLoop`(*inherits* *GlobalEngine* → *Engine* → *Serializable*)
Unified dispatcher for handling interaction loop at every step, for parallel performance reasons.

Special constructor

Constructs from 3 lists of *Ig2*, *Ip2*, *Law* functors respectively; they will be passed to interal dispatchers, which you might retrieve.

---

callbacks(=*uninitalized*)
Callbacks which will be called for every *Interaction*, if activated.

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

eraseIntsInLoop(=*false*)
Defines if the interaction loop should erase pending interactions, else the collider takes care of that alone (depends on what collider is used).

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

geomDispatcher(=*new IGeomDispatcher*)
IGeomDispatcher object that is used for dispatch.

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

lawDispatcher(=*new LawDispatcher*)
LawDispatcher object used for dispatch.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

physDispatcher(=*new IPhysDispatcher*)
IPhysDispatcher object used for dispatch.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom_CapillaryPhys_Capillarity**(*inherits GlobalEngine → Engine → Serializable*)

This law allows one to take into account capillary forces/effects between spheres coming from the presence of interparticular liquid bridges (menisci).

The control parameter is the *capillary pressure* (or suction) Uc = Ugas - Uliquid. Liquid bridges properties (volume V, extent over interacting grains delta1 and delta2) are computed as a result of the defined capillary pressure and of the interacting geometry (spheres radii and interparticular distance).

References: in english *[Scholtes2009b]*; more detailed, but in french *[Scholtes2009d]*.

The law needs ascii files M(r=i) with i=R1/R2 to work (see https://yade-dem.org/wiki/CapillaryTriaxialTest). These ASCII files contain a set of results from the resolution of the Laplace-Young equation for different configurations of the interacting geometry, assuming a null wetting angle.

In order to allow capillary forces between distant spheres, it is necessary to enlarge the bounding boxes using *Bo1_Sphere_Aabb::aabbEnlargeFactor* and make the Ig2 define define distant interactions via *interactionDetectionFactor*. It is also necessary to disable interactions removal by the constitutive law (*Law2*). The only combinations of laws supported are currently capillary law + *Law2_ScGeom_FrictPhys_CundallStrack* and capillary law + *Law2_ScGeom_MindlinPhys_Mindlin* (and the other variants of Hertz-Mindlin).

See CapillaryPhys-example.py for an example script.

binaryFusion(=*true*)
If true, capillary forces are set to zero as soon as, at least, 1 overlap (menisci fusion) is detected. Otherwise *fCap* = *fCap* / (*fusionNumber* + 1)

capillaryPressure(=*0.*)
Value of the capillary pressure Uc defined as Uc=Ugas-Uliquid

createDistantMeniscii(=*false*)
Generate meniscii between distant spheres? Else only maintain the existing ones. For modeling a wetting path this flag should always be false. For a drying path it should be true for one step (initialization) then false, as in the logic of *[Scholtes2009c]*

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

fusionDetection(=*false*)
If true potential menisci overlaps are checked, computing *fusionNumber* for each capillary interaction, and reducing *fCap* according to *binaryFusion*

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be

typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

surfaceTension(*=0.073*)
Value of considered surface tension

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**MicroMacroAnalyser**(*inherits GlobalEngine → Engine → Serializable*)
compute fabric tensor, local porosity, local deformation, and other micromechanicaly defined quantities based on triangulation/tesselation of the packing.

compDeformation(*=false*)
Is the engine just saving states or also computing and outputing deformations for each increment?

compIncrt(*=false*)
Should increments of force and displacements be defined on [n,n+1]? If not, states will be saved with only positions and forces (no displacements).

dead(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

incrtNumber(*=1*)

interval(*=100*)
Number of timesteps between analyzed states.

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

nonSphereAsFictious(*=true*)
bodies that are not spheres will be used to defines bounds (else just skipped).

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

outputFile(*="MicroMacroAnalysis"*)
Base name for increment analysis output file.

stateFileName(*="state"*)
Base name of state files.

stateNumber(*=0*)
A number incremented and appended at the end of output files to reflect increment number.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**NewtonIntegrator**(*inherits GlobalEngine → Engine → Serializable*)
Engine integrating newtonian motion equations.

damping(*=0.2*)
damping coefficient for Cundall's non viscous damping (see numerical damping and *[Chareyre2005]*)

dead(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

densityScaling
if True, then density scaling *[Pfc3dManual30]* will be applied in order to have a critical timestep equal to *GlobalStiffnessTimeStepper::targetDt* for all bodies. This option makes the simulation unrealistic from a dynamic point of view, but may speedup quasistatic simulations. In rare situations, it could be useful to not set the scalling factor automatically for each body (which the time-stepper does). In such case revert *GlobalStiffnessTimeStepper.densityScaling* to False.

dict() → dict
Return dictionary of attributes.

exactAsphericalRot(*=true*)
Enable more exact body rotation integrator for *aspherical bodies only*, using formulation from *[Allen1989]*, pg. 89.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

gravity(*=Vector3r::Zero()*)
Gravitational acceleration (effectively replaces GravityEngine).

kinSplit(*=false*)
Whether to separately track translational and rotational kinetic energy.

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

mask(*=-1*)
If mask defined and the bitwise AND between mask and body's groupMask gives 0, the body will not move/rotate. Velocities and accelerations will be calculated not paying attention to this parameter.

maxVelocitySq(*=NaN*)
store square of max. velocity, for informative purposes; computed again at every step. *(auto-updated)*

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

prevVelGrad(=*Matrix3r::Zero()*)
Store previous velocity gradient (*Cell::velGrad*) to track acceleration. *(auto-updated)*

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

warnNoForceReset(=*true*)
Warn when forces were not resetted in this step by *ForceResetter*; this mostly points to *ForceResetter* being forgotten incidentally and should be disabled only with a good reason.

class `yade.wrapper.PeriodicEngine`(*inherits* *GlobalEngine* → *Engine* → *Serializable*)
Run Engine::action with given fixed periodicity real time (=wall clock time, computation time), virtual time (simulation time), iteration number), by setting any of those criteria (virtPeriod, realPeriod, iterPeriod) to a positive value. They are all negative (inactive) by default.

The number of times this engine is activated can be limited by setting nDo>0. If the number of activations will have been already reached, no action will be called even if an active period has elapsed.

If initRun is set (false by default), the engine will run when called for the first time; otherwise it will only start counting period (realLast etc interal variables) from that point, but without actually running, and will run only once a period has elapsed since the initial run.

This class should not be used directly; rather, derive your own engine which you want to be run periodically.

Derived engines should override Engine::action(), which will be called periodically. If the derived Engine overrides also Engine::isActivated, it should also take in account return value from PeriodicEngine::isActivated, since otherwise the periodicity will not be functional.

Example with *PyRunner*, which derives from PeriodicEngine; likely to be encountered in python scripts:

```
PyRunner(realPeriod=5,iterPeriod=10000,command='print O.iter')
```

will print iteration number every 10000 iterations or every 5 seconds of wall clock time, whiever comes first since it was last run.

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

firstIterRun(=*0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

initRun(=*false*)
Run the first time we are called as well.

iterLast(=*0*)
Tracks step number of last run *(auto-updated)*.

iterPeriod(=*0*, *deactivated*)
Periodicity criterion using step number (deactivated if <= 0)

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`nDo`(=*-1*, *deactivated*)
: Limit number of executions by this number (deactivated if negative)

`nDone`(=*0*)
: Track number of executions (cummulative) *(auto-updated)*.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`realLast`(=*0*)
: Tracks real time of last run *(auto-updated)*.

`realPeriod`(=*0*, *deactivated*)
: Periodicity criterion using real (wall clock, computation, human) time (deactivated if <=0)

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

`virtLast`(=*0*)
: Tracks virtual time of last run *(auto-updated)*.

`virtPeriod`(=*0*, *deactivated*)
: Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

class `yade.wrapper.PolyhedraSplitter`(*inherits* *PeriodicEngine* → *GlobalEngine* → *Engine* → *Serializable*)
: Engine that splits polyhedras.

`dead`(=*false*)
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
: Return dictionary of attributes.

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`firstIterRun`(=*0*)
: Sets the step number, at each an engine should be executed for the first time (disabled by default).

`initRun`(=*false*)
: Run the first time we are called as well.

`iterLast`(=*0*)
: Tracks step number of last run *(auto-updated)*.

`iterPeriod`(=*0*, *deactivated*)
: Periodicity criterion using step number (deactivated if <= 0)

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`nDo`(=*-1*, *deactivated*)
: Limit number of executions by this number (deactivated if negative)

`nDone`(=*0*)
: Track number of executions (cummulative) *(auto-updated)*.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`realLast`(=*0*)
: Tracks real time of last run *(auto-updated)*.

`realPeriod`(=*0*, *deactivated*)
: Periodicity criterion using real (wall clock, computation, human) time (deactivated if <=0)

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

`virtLast`(=*0*)
: Tracks virtual time of last run *(auto-updated)*.

`virtPeriod`(=*0*, *deactivated*)
: Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

class `yade.wrapper.PyRunner`(*inherits* *PeriodicEngine* → *GlobalEngine* → *Engine* → *Serializable*)
Execute a python command periodically, with defined (and adjustable) periodicity. See *PeriodicEngine* documentation for details.

`command`(=*""*)
: Command to be run by python interpreter. Not run if empty.

`dead`(=*false*)
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
: Return dictionary of attributes.

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`firstIterRun`(=*0*)
: Sets the step number, at each an engine should be executed for the first time (disabled by default).

`initRun`(=*false*)
: Run the first time we are called as well.

`iterLast`(=*0*)
: Tracks step number of last run *(auto-updated)*.

`iterPeriod`(=*0*, *deactivated*)
: Periodicity criterion using step number (deactivated if <= 0)

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`nDo`(=*-1*, *deactivated*)
: Limit number of executions by this number (deactivated if negative)

`nDone`(=*0*)
: Track number of executions (cummulative) *(auto-updated)*.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`realLast`(=*0*)
: Tracks real time of last run *(auto-updated)*.

`realPeriod`(=*0*, *deactivated*)
: Periodicity criterion using real (wall clock, computation, human) time (deactivated if <=0)

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

`virtLast`(=*0*)
: Tracks virtual time of last run *(auto-updated)*.

`virtPeriod`(=*0*, *deactivated*)
: Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

class `yade.wrapper.Recorder`(*inherits* *PeriodicEngine* → *GlobalEngine* → *Engine* → *Serializable*)

Engine periodically storing some data to (one) external file. In addition PeriodicEngine, it handles opening the file as needed. See *PeriodicEngine* for controlling periodicity.

`addIterNum`(=*false*)
: Adds an iteration number to the file name, when the file was created. Useful for creating new files at each call (false by default)

`dead`(=*false*)
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
: Return dictionary of attributes.

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`file`(=*uninitalized*)
: Name of file to save to; must not be empty.

`firstIterRun`(=*0*)
: Sets the step number, at each an engine should be executed for the first time (disabled by default).

`initRun`(=*false*)
Run the first time we are called as well.

`iterLast`(=*0*)
Tracks step number of last run *(auto-updated)*.

`iterPeriod`(=*0*, *deactivated*)
Periodicity criterion using step number (deactivated if <= 0)

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`nDo`(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

`nDone`(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`realLast`(=*0*)
Tracks real time of last run *(auto-updated)*.

`realPeriod`(=*0*, *deactivated*)
Periodicity criterion using real (wall clock, computation, human) time (deactivated if <=0)

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`truncate`(=*false*)
Whether to delete current file contents, if any, when opening (false by default)

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`virtLast`(=*0*)
Tracks virtual time of last run *(auto-updated)*.

`virtPeriod`(=*0*, *deactivated*)
Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

class `yade.wrapper.ResetRandomPosition`(*inherits* *PeriodicEngine* → *GlobalEngine* → *Engine* → *Serializable*)
Creates spheres during simulation, placing them at random positions. Every time called, one new sphere will be created and inserted in the simulation.

`angularVelocity`(=*Vector3r::Zero()*)
Mean angularVelocity of spheres.

`angularVelocityRange`(=*Vector3r::Zero()*)
Half size of a angularVelocity distribution interval. New sphere will have random angularVelocity within the range angularVelocity±angularVelocityRange.

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

factoryFacets(=*uninitalized*)
The geometry of the section where spheres will be placed; they will be placed on facets or in volume between them depending on *volumeSection* flag.

firstIterRun(=*0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

initRun(=*false*)
Run the first time we are called as well.

iterLast(=*0*)
Tracks step number of last run *(auto-updated)*.

iterPeriod(=*0*, *deactivated*)
Periodicity criterion using step number (deactivated if <= 0)

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxAttempts(=*20*)
Max attempts to place sphere. If placing the sphere in certain random position would cause an overlap with any other physical body in the model, SpheresFactory will try to find another position.

nDo(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

nDone(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

normal(=*Vector3r(0, 1, 0)*)
??

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

point(=*Vector3r::Zero()*)
??

realLast(=*0*)
Tracks real time of last run *(auto-updated)*.

realPeriod(=*0*, *deactivated*)
Periodicity criterion using real (wall clock, computation, human) time (deactivated if <=0)

subscribedBodies(=*uninitalized*)
Affected bodies.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

`velocity`(=*Vector3r::Zero()*)
Mean velocity of spheres.

`velocityRange`(=*Vector3r::Zero()*)
Half size of a velocities distribution interval. New sphere will have random velocity within the range velocity±velocityRange.

`virtLast`(=*0*)
Tracks virtual time of last run *(auto-updated)*.

`virtPeriod`(=*0*, *deactivated*)
Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

`volumeSection`(=*false*, *define factory by facets.*)
Create new spheres inside factory volume rather than on its surface.

class `yade.wrapper.RungeKuttaCashKarp54Integrator`(*inherits* *Integrator* → *TimeStepper* → *GlobalEngine* → *Engine* → *Serializable*)

RungeKuttaCashKarp54Integrator engine.

`__init__`() → None
object ___init___(tuple args, dict kwds)

___init___((list)arg2) → object : Construct from (possibly nested) list of slaves.

`a_dxdt`(=*1.0*)

`a_x`(=*1.0*)

`abs_err`(=*1e-6*)
Relative integration tolerance

`active`(=*true*)
is the engine active?

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`integrationsteps`(=*uninitalized*)
all integrationsteps count as all succesfull substeps

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`maxVelocitySq`(=*NaN*)
store square of max. velocity, for informative purposes; computed again at every step. *(auto-updated)*

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`rel_err`(=*1e-6*)
Absolute integration tolerance

slaves
List of lists of Engines to calculate the force acting on the particles; to obtain the derivatives of the states, engines inside will be run sequentially.

stepsize(*=1e-6*)
It is not important for an adaptive integration but important for the observer for setting the found states after integration

timeStepUpdateInterval(*=1*)
dt update interval

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**SnapshotEngine**(*inherits PeriodicEngine → GlobalEngine → Engine → Serializable*)
Periodically save snapshots of GLView(s) as .png files. Files are named *fileBase* + *counter* + '.png' (counter is left-padded by 0s, i.e. snap00004.png).

counter(*=0*)
Number that will be appended to fileBase when the next snapshot is saved (incremented at every save). *(auto-updated)*

dead(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

deadTimeout(*=3*)
Timeout for 3d operations (opening new view, saving snapshot); after timing out, throw exception (or only report error if *ignoreErrors*) and make myself *dead*. [s]

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

fileBase(*=""*)
Basename for snapshots

firstIterRun(*=0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

format(*="PNG"*)
Format of snapshots (one of JPEG, PNG, EPS, PS, PPM, BMP) QGLViewer documentation. File extension will be lowercased *format*. Validity of format is not checked.

ignoreErrors(*=true*)
Only report errors instead of throwing exceptions, in case of timeouts.

initRun(*=false*)
Run the first time we are called as well.

iterLast(*=0*)
Tracks step number of last run *(auto-updated)*.

iterPeriod(*=0*, *deactivated*)
Periodicity criterion using step number (deactivated if <= 0)

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

msecSleep(=*0*)
number of msec to sleep after snapshot (to prevent 3d hw problems) [ms]

nDo(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

nDone(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

plot(=*uninitalized*)
Name of field in *plot.imgData* to which taken snapshots will be appended automatically.

realLast(=*0*)
Tracks real time of last run *(auto-updated)*.

realPeriod(=*0*, *deactivated*)
Periodicity criterion using real (wall clock, computation, human) time (deactivated if <=0)

snapshots(=*uninitalized*)
Files that have been created so far

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

virtLast(=*0*)
Tracks virtual time of last run *(auto-updated)*.

virtPeriod(=*0*, *deactivated*)
Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

class `yade.wrapper.SpheresFactory`(*inherits* *GlobalEngine* → *Engine* → *Serializable*)
Engine for spitting spheres based on mass flow rate, particle size distribution etc. Initial velocity of particles is given by *vMin*, *vMax*, the *massFlowRate* determines how many particles to generate at each step. When *goalMass* is attained or positive *maxParticles* is reached, the engine does not produce particles anymore. Geometry of the region should be defined in a derived engine by overridden SpheresFactory::pickRandomPosition().

A sample script for this engine is in scripts/spheresFactory.py.

PSDcalculateMass(=*true*)
PSD-Input is in mass (true), otherwise the number of particles will be considered.

PSDcum(=*uninitalized*)
PSD-dispersion, cumulative procent meanings [-]

PSDsizes(=*uninitalized*)
PSD-dispersion, sizes of cells, Diameter [m]

blockedDOFs(=*""*)
Blocked degress of freedom

color(=*Vector3r(-1, -1, -1)*)
Use the color for newly created particles, if specified

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

exactDiam(=*true*)
If true, the particles only with the defined in PSDsizes diameters will be created. Otherwise the diameter will be randomly chosen in the range [PSDsizes[i-1]:PSDsizes[i]], in this case the length of PSDsizes should be more on 1, than the length of PSDcum.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

goalMass(=*0*)
Total mass that should be attained at the end of the current step. *(auto-updated)*

ids(=*uninitalized*)
ids of created bodies

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

mask(=*-1*)
groupMask to apply for newly created spheres

massFlowRate(=*NaN*)
Mass flow rate [kg/s]

materialId(=*-1*)
Shared material id to use for newly created spheres (can be negative to count from the end)

maxAttempt(=*5000*)
Maximum number of attempts to position a new sphere randomly.

maxMass(=*-1*)
Maximal mass at which to stop generating new particles regardless of massFlowRate. if maxMass=-1 - this parameter is ignored.

maxParticles(=*100*)
The number of particles at which to stop generating new ones regardless of massFlowRate. if maxParticles=-1 - this parameter is ignored .

normal(=*Vector3r(NaN, NaN, NaN)*)
Orientation of the region's geometry, direction of particle's velocites if normalVel is not set.

normalVel(=*Vector3r(NaN, NaN, NaN)*)
Direction of particle's velocites.

numParticles(=*0*)
Cummulative number of particles produces so far *(auto-updated)*

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

rMax(=*NaN*)
Maximum radius of generated spheres (uniform distribution)

rMin(=*NaN*)
Minimum radius of generated spheres (uniform distribution)

silent(=*false*)
If true no complain about excessing maxAttempt but disable the factory (by set massFlowRate=0).

stopIfFailed(=*true*)
If true, the SpheresFactory stops (sets massFlowRate=0), when maximal number of attempts to insert particle exceed.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

totalMass(=*0*)
Mass of spheres that was produced so far. *(auto-updated)*

totalVolume(=*0*)
Volume of spheres that was produced so far. *(auto-updated)*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

vAngle(=*NaN*)
Maximum angle by which the initial sphere velocity deviates from the normal.

vMax(=*NaN*)
Maximum velocity norm of generated spheres (uniform distribution)

vMin(=*NaN*)
Minimum velocity norm of generated spheres (uniform distribution)

class yade.wrapper.**TesselationWrapper**(*inherits GlobalEngine → Engine → Serializable*)
Handle the triangulation of spheres in a scene, build tesselation on request, and give access to computed quantities (see also the dedicated section in user manual). The calculation of microstrain is explained in *[Catalano2014a]*

See example usage in script example/tesselationWrapper/tesselationWrapper.py.

Below is an output of the *defToVtk* function visualized with paraview (in this case Yade's TesselationWrapper was used to process experimental data obtained on sand by Edward Ando at Grenoble University, 3SR lab.)

computeDeformations() → None
compute per-particle deformation. Get it with *TesselationWrapper::deformation* (id,i,j).

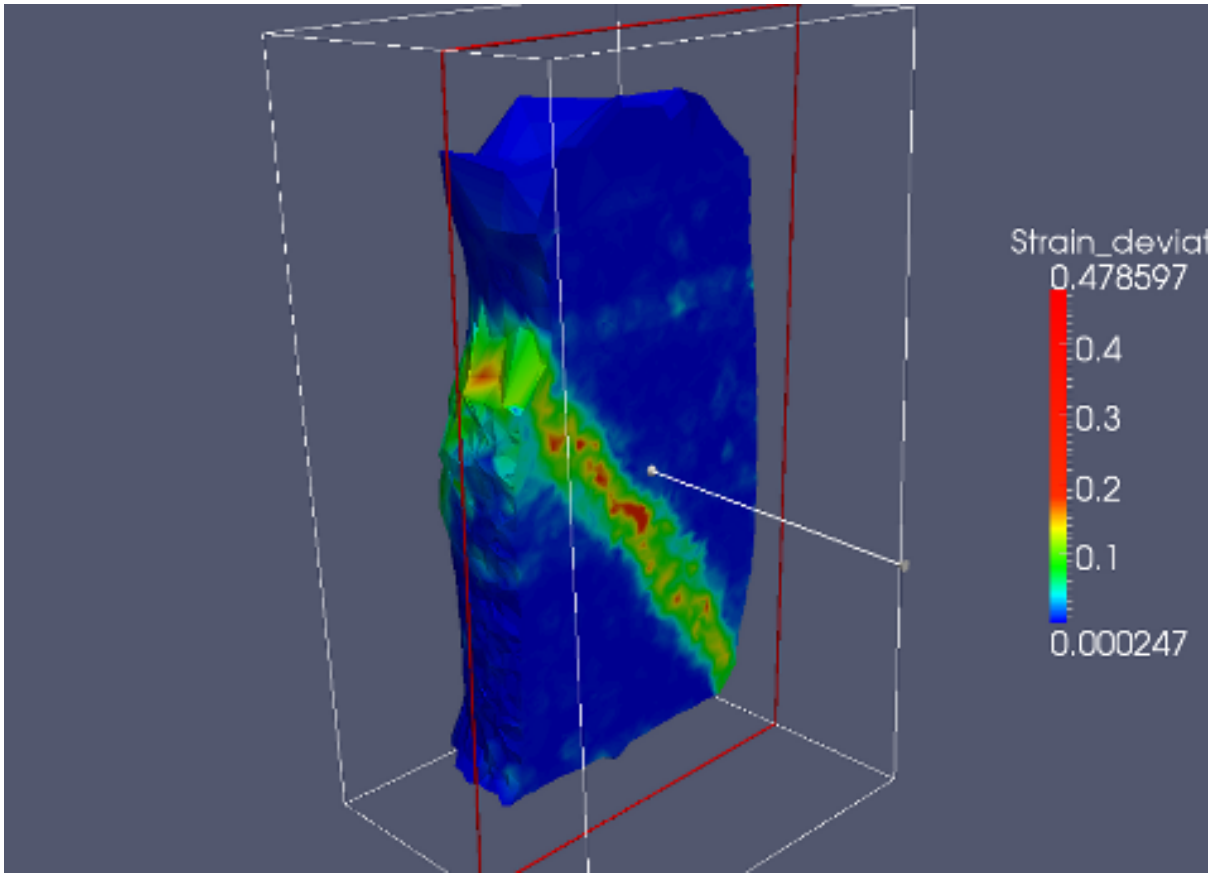
computeVolumes() → None
compute volumes of all Voronoi's cells.

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

defToVtk(*[(str)outputFile='def.vtk']*) → None
Write local deformations in vtk format from states 0 and 1.

defToVtkFromPositions(*[(str)input1='pos1'[, (str)input2='pos2'[, (str)outputFile='def.vtk'[, (bool)bz2=False]]]]*) → None
Write local deformations in vtk format from positions files (one sphere per line, with x,y,z,rad separated by spaces).

defToVtkFromStates(*[(str)input1='state1'[, (str)input2='state2'[, (str)outputFile='def.vtk'[, (bool)bz2=True]]]]*) → None
Write local deformations in vtk format from state files (since the file format is very special, consider using defToVtkFromPositions if the input files were not generated by TesselationWrapper).



deformation(*(int)id*, *(int)i*, *(int)j*) → float
Get particle deformation

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

far(*=10000.*)
Defines the radius of the large virtual spheres used to define nearly flat boundaries around the assembly. The radius will be the (scene's) bounding box size multiplied by 'far'. Higher values will minimize the error theoretically (since the infinite sphere really defines a plane), but it may increase numerical errors at some point. The default should give a resonable compromize.

getVolPoroDef([*(bool)deformation=False*]) → dict
Return a table with per-sphere computed quantities. Include deformations on the increment defined by states 0 and 1 if deformation=True (make sure to define states 0 and 1 consistently).

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

loadState([*(str)inputFile='state'*[, *(bool)state=0*[, *(bool)bz2=True*]]]) → None
Load a file with positions to define state 0 or 1.

n_spheres(*=0*)
(auto-computed)

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be

typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

saveState([*(str)outputFile='state'*[, *(bool)state=0*[, *(bool)bz2=True*]]]) → None
Save a file with positions, can be later reloaded in order to define state 0 or 1.

setState([*(bool)state=0*]) → None
Make the current state of the simulation the initial (0) or final (1) configuration for the definition of displacement increments, use only state=0 if you just want to get volmumes and porosity.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

triangulate([*(bool)reset=True*]) → None
triangulate spheres of the packing

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

volume([*(int)id=0*]) → float
Returns the volume of Voronoi's cell of a sphere.

class `yade.wrapper.TetraVolumetricLaw`(*inherits GlobalEngine → Engine → Serializable*)
Calculate physical response of 2 *tetrahedra* in interaction, based on penetration configuration given by *TTetraGeom*.

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.TimeStepper`(*inherits GlobalEngine → Engine → Serializable*)
Engine defining time-step (fundamental class)

`active`(=*true*)
: is the engine active?

`dead`(=*false*)
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
: Return dictionary of attributes.

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timeStepUpdateInterval`(=*1*)
: dt update interval

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

class `yade.wrapper.TorqueRecorder`(*inherits* *Recorder* → *PeriodicEngine* → *GlobalEngine* → *Engine* → *Serializable*)

Engine saves the total torque according to the given axis and ZeroPoint, the force is taken from bodies, listed in *ids* For instance, can be useful for defining the torque, which affects on ball mill during its work.

`addIterNum`(=*false*)
: Adds an iteration number to the file name, when the file was created. Useful for creating new files at each call (false by default)

`dead`(=*false*)
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
: Return dictionary of attributes.

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`file`(=*uninitalized*)
: Name of file to save to; must not be empty.

`firstIterRun`(=*0*)
: Sets the step number, at each an engine should be executed for the first time (disabled by default).

`ids`(=*uninitalized*)
List of bodies whose state will be measured

`initRun`(=*false*)
Run the first time we are called as well.

`iterLast`(=*0*)
Tracks step number of last run *(auto-updated)*.

`iterPeriod`(=*0*, *deactivated*)
Periodicity criterion using step number (deactivated if <= 0)

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`nDo`(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

`nDone`(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`realLast`(=*0*)
Tracks real time of last run *(auto-updated)*.

`realPeriod`(=*0*, *deactivated*)
Periodicity criterion using real (wall clock, computation, human) time (deactivated if <=0)

`rotationAxis`(=*Vector3r::UnitX()*)
Rotation axis

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`totalTorque`(=*0*)
Resultant torque, returning by the function.

`truncate`(=*false*)
Whether to delete current file contents, if any, when opening (false by default)

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`virtLast`(=*0*)
Tracks virtual time of last run *(auto-updated)*.

`virtPeriod`(=*0*, *deactivated*)
Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

`zeroPoint`(=*Vector3r::Zero()*)
Point of rotation center

class `yade.wrapper.TriaxialStateRecorder`(*inherits* *Recorder* → *PeriodicEngine* → *GlobalEngine* → *Engine* → *Serializable*)
Engine recording triaxial variables (see the variables list in the first line of the output file). This recorder needs *TriaxialCompressionEngine* or *ThreeDTriaxialEngine* present in the simulation).

`addIterNum`(=*false*)
Adds an iteration number to the file name, when the file was created. Useful for creating new files at each call (false by default)

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

file(=*uninitalized*)
Name of file to save to; must not be empty.

firstIterRun(=*0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

initRun(=*false*)
Run the first time we are called as well.

iterLast(=*0*)
Tracks step number of last run *(auto-updated)*.

iterPeriod(=*0*, *deactivated*)
Periodicity criterion using step number (deactivated if <= 0)

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

nDo(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

nDone(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

porosity(=*1*)
porosity of the packing [-]

realLast(=*0*)
Tracks real time of last run *(auto-updated)*.

realPeriod(=*0*, *deactivated*)
Periodicity criterion using real (wall clock, computation, human) time (deactivated if <=0)

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

truncate(=*false*)
Whether to delete current file contents, if any, when opening (false by default)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

virtLast(=*0*)
Tracks virtual time of last run *(auto-updated)*.

`virtPeriod`(=*0*, *deactivated*)
Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

class `yade.wrapper.VTKRecorder`(*inherits* *PeriodicEngine* → *GlobalEngine* → *Engine* → *Serializable*)
Engine recording snapshots of simulation into series of *.vtu files, readable by VTK-based post-processing programs such as Paraview. Both bodies (spheres and facets) and interactions can be recorded, with various vector/scalar quantities that are defined on them.

PeriodicEngine.initRun is initialized to `True` automatically.

`Key`(="")
Necessary if *recorders* contains 'cracks'. A string specifying the name of file 'cracks_____.txt' that is considered in this case (see *corresponding attribute*).

`ascii`(=*false*)
Store data as readable text in the XML file (sets vtkXMLWriter data mode to `vtkXMLWriter::Ascii`, while the default is `Appended`

`compress`(=*false*)
Compress output XML files [experimental].

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`fileName`(="")
Base file name; it will be appended with {spheres,intrs,facets}-243100.vtu (unless *multiblock* is `True`) depending on active recorders and step number (243100 in this case). It can contain slashes, but the directory must exist already.

`firstIterRun`(=*0*)
Sets the step number, at each an engine should be executed for the first time (disabled by default).

`initRun`(=*false*)
Run the first time we are called as well.

`iterLast`(=*0*)
Tracks step number of last run *(auto-updated)*.

`iterPeriod`(=*0*, *deactivated*)
Periodicity criterion using step number (deactivated if <= 0)

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`mask`(=*0*)
If mask defined, only bodies with corresponding groupMask will be exported. If 0, all bodies will be exported.

`multiblock`(=*false*)
Use multi-block (`.vtm`) files to store data, rather than separate `.vtu` files.

`nDo`(=*-1*, *deactivated*)
Limit number of executions by this number (deactivated if negative)

`nDone`(=*0*)
Track number of executions (cummulative) *(auto-updated)*.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`realLast`(=*0*)
: Tracks real time of last run *(auto-updated)*.

`realPeriod`(=*0*, *deactivated*)
: Periodicity criterion using real (wall clock, computation, human) time (deactivated if <=0)

`recorders`
: List of active recorders (as strings). `all` (the default value) enables all base and generic recorders.

Base recorders

Base recorders save the geometry (unstructured grids) on which other data is defined. They are implicitly activated by many of the other recorders. Each of them creates a new file (or a block, if *multiblock* is set).

`spheres` Saves positions and radii (`radii`) of *spherical* particles.

`facets` Save *facets* positions (vertices).

`boxes` Save *boxes* positions (edges).

`intr` Store interactions as lines between nodes at respective particles positions. Additionally stores magnitude of normal (`forceN`) and shear (`absForceT`) forces on interactions (the *geom*).

Generic recorders

Generic recorders do not depend on specific model being used and save commonly useful data.

`id` Saves id's (field `id`) of spheres; active only if `spheres` is active.

`mass` Saves masses (field `mass`) of spheres; active only if `spheres` is active.

`clumpId` Saves id's of clumps to which each sphere belongs (field `clumpId`); active only if `spheres` is active.

`colors` Saves colors of *spheres* and of *facets* (field `color`); only active if `spheres` or `facets` are activated.

`mask` Saves groupMasks of *spheres* and of *facets* (field `mask`); only active if `spheres` or `facets` are activated.

`materialId` Saves materialID of *spheres* and of *facets*; only active if `spheres` or `facets` are activated.

`coordNumber` Saves coordination number (number of neighbours) of *spheres* and of *facets*; only active if `spheres` or `facets` are activated.

`velocity` Saves linear and angular velocities of spherical particles as Vector3 and length(fields `linVelVec`, `linVelLen` and `angVelVec`, `angVelLen` respectively"); only effective with `spheres`.

`stress` Saves stresses of *spheres* and of *facets* as Vector3 and length; only active if `spheres` or `facets` are activated.

`force` Saves force and torque of *spheres*, *facets* and *boxes* as Vector3 and length (norm); only active if `spheres`, `facets` or `boxes` are activated.

`pericell` Saves the shape of the cell (simulation has to be periodic).

bstresses Saves per-particle principal stresses (sigI >= sigII >= sigIII) and associated principal directions (dirI/II/III). Per-particle stress tensors are given by *bodyStressTensors* (positive values for tensile states).

Specific recorders

The following should only be activated in appropriate cases, otherwise crashes can occur due to violation of type presuppositions.

cpm Saves data pertaining to the *concrete model*: `cpmDamage` (normalized residual strength averaged on particle), `cpmStress` (stress on particle); `intr` is activated automatically by `cpm`

wpm Saves data pertaining to the *wire particle model*: `wpmForceNFactor` shows the loading factor for the wire, e.g. normal force divided by threshold normal force.

jcfpm Saves data pertaining to the *rock (smooth)-jointed model*: `damage` is defined by *JCFpmState.tensBreak* + *JCFpmState.shearBreak*; `intr` is activated automatically by `jcfpm`, and *on joint* or *cohesive* interactions can be vizualized.

cracks Saves other data pertaining to the *rock model*: `cracks` shows locations where cohesive bonds failed during the simulation, with their types (0/1 for tensile/shear breakages), their sizes (0.5*(R1+R2)), and their normal directions. The *corresponding attribute* has to be activated, and Key attributes have to be consistent.

---

skipFacetIntr(=*true*)
Skip interactions that are not of sphere-sphere type (e.g. sphere-facet, sphere-box...), when saving interactions

skipNondynamic(=*false*)
Skip non-dynamic spheres (but not facets).

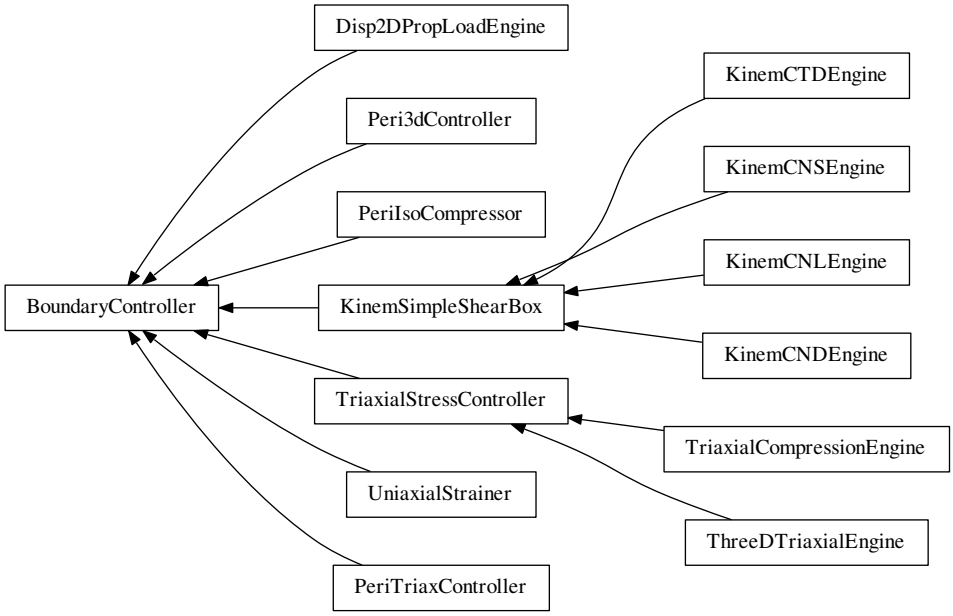
timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

virtLast(=*0*)
Tracks virtual time of last run *(auto-updated)*.

virtPeriod(=*0*, *deactivated*)
Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

8.3.2 BoundaryController



class `yade.wrapper.BoundaryController`(*inherits* *GlobalEngine* → *Engine* → *Serializable*)

Base for engines controlling boundary conditions of simulations. Not to be used directly.

`dead`(=*false*)
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
: Return dictionary of attributes.

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

class `yade.wrapper.Disp2DPropLoadEngine`(*inherits* *BoundaryController* → *GlobalEngine* → *Engine* → *Serializable*)

Disturbs a simple shear sample in a given displacement direction

This engine allows one to apply, on a simple shear sample, a loading controlled by du/dgamma = cste, which is equivalent to du + cste' * dgamma = 0 (proportionnal path loadings). To do so, the upper plate of the simple shear box is moved in a given direction (corresponding to a given du/dgamma), whereas lateral plates are moved so that the box remains closed. This engine can easily be used to perform directionnal probes, with a python script launching successivly the same .xml which contains this engine, after having modified the direction of loading (see *theta* attribute). That's why this Engine contains a *saveData* procedure which can save data on the state of the sample at the end of the loading (in case of successive loadings - for successive directions - through a python script, each line would correspond to one direction of loading).

`Key(=""")`
: string to add at the names of the saved files, and of the output file filled by *saveData*

`LOG(=false)`
: boolean controling the output of messages on the screen

`dead(=false)`
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
: Return dictionary of attributes.

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`id_boxback(=4)`
: the id of the wall at the back of the sample

`id_boxbas(=1)`
: the id of the lower wall

`id_boxfront(=5)`
: the id of the wall in front of the sample

`id_boxleft(=0)`
: the id of the left wall

`id_boxright(=2)`
: the id of the right wall

`id_topbox(=3)`
: the id of the upper wall

`label(=uninitalized)`
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`nbre_iter(=0)`
: the number of iterations of loading to perform

`ompThreads(=-1)`
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`theta(=0.0)`
: the angle, in a (gamma,h=-u) plane from the gamma - axis to the perturbation vector (trigo wise) [degrees]

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`v`(*=0.0*)
the speed at which the perturbation is imposed. In case of samples which are more sensitive to normal loadings than tangential ones, one possibility is to take v = V_shear - | (V_shear-V_comp)*sin(theta) | => v=V_shear in shear; V_comp in compression [m/s]

class `yade.wrapper.KinemCNDEngine`(*inherits* *KinemSimpleShearBox* → *BoundaryController* → *GlobalEngine* → *Engine* → *Serializable*)
To apply a Constant Normal Displacement (CND) shear for a parallelogram box

This engine, designed for simulations implying a simple shear box (*SimpleShear* Preprocessor or scripts/simpleShear.py), allows one to perform a constant normal displacement shear, by translating horizontally the upper plate, while the lateral ones rotate so that they always keep contact with the lower and upper walls.

`Key`(*=""*)
string to add at the names of the saved files

`LOG`(*=false*)
boolean controling the output of messages on the screen

`alpha`(*=Mathr::PI/2.0*)
the angle from the lower box to the left box (trigo wise). Measured by this Engine. Has to be saved, but not to be changed by the user.

`dead`(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`f0`(*=0.0*)
the (vertical) force acting on the upper plate on the very first time step (determined by the Engine). Controls of the loadings in case of *KinemCNSEngine* or *KinemCNLEngine* will be done according to this initial value [N]. Has to be saved, but not to be changed by the user.

`firstRun`(*=true*)
boolean set to false as soon as the engine has done its job one time : useful to know if initial height of, and normal force sustained by, the upper box are known or not (and thus if they have to be initialized). Has to be saved, but not to be changed by the user.

`gamma`(*=0.0*)
the current value of the tangential displacement

`gamma_save`(*=uninitalized*)
vector with the values of gamma at which a save of the simulation is performed [m]

`gammalim`(*=0.0*)
the value of the tangential displacement at wich the displacement is stopped [m]

`id_boxback`(*=4*)
the id of the wall at the back of the sample

`id_boxbas`(*=1*)
the id of the lower wall

`id_boxfront`(=*5*)
: the id of the wall in front of the sample

`id_boxleft`(=*0*)
: the id of the left wall

`id_boxright`(=*2*)
: the id of the right wall

`id_topbox`(=*3*)
: the id of the upper wall

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`max_vel`(=*1.0*)
: to limit the speed of the vertical displacements done to control σ (CNL or CNS cases) [m/s]

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`shearSpeed`(=*0.0*)
: the speed at which the shear is performed : speed of the upper plate [m/s]

`temoin_save`(=*uninitalized*)
: vector (same length as 'gamma_save' for ex), with 0 or 1 depending whether the save for the corresponding value of gamma has been done (1) or not (0). Has to be saved, but not to be changed by the user.

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

`wallDamping`(=*0.2*)
: the vertical displacements done to to control σ (CNL or CNS cases) are in fact damped, through this wallDamping

`y0`(=*0.0*)
: the height of the upper plate at the very first time step : the engine finds its value [m]. Has to be saved, but not to be changed by the user.

class `yade.wrapper.KinemCNLEngine`(*inherits* *KinemSimpleShearBox* → *BoundaryController* → *GlobalEngine* → *Engine* → *Serializable*)

To apply a constant normal stress shear (i.e. Constant Normal Load : CNL) for a parallelogram box (simple shear box : *SimpleShear* Preprocessor or scripts/simpleShear.py)

This engine allows one to translate horizontally the upper plate while the lateral ones rotate so that they always keep contact with the lower and upper walls.

In fact the upper plate can move not only horizontally but also vertically, so that the normal stress acting on it remains constant (this constant value is not chosen by the user but is the one that exists at the beginning of the simulation)

The right vertical displacements which will be allowed are computed from the rigidity Kn of the sample over the wall (so to cancel a deltaSigma, a normal dplt deltaSigma*S/(Kn) is set)

The movement is moreover controlled by the user via a *shearSpeed* which will be the speed of the upper wall, and by a maximum value of horizontal displacement *gammalim*, after which the shear stops.

Note: Not only the positions of walls are updated but also their speeds, which is all but useless considering the fact that in the contact laws these velocities of bodies are used to compute values of tangential relative displacements.

Warning: Because of this last point, if you want to use later saves of simulations executed with this Engine, but without that stopMovement was executed, your boxes will keep their speeds => you will have to cancel them 'by hand' in the .xml.

`Key(="")`
: string to add at the names of the saved files

`LOG(=false)`
: boolean controling the output of messages on the screen

`alpha(=Mathr::PI/2.0)`
: the angle from the lower box to the left box (trigo wise). Measured by this Engine. Has to be saved, but not to be changed by the user.

`dead(=false)`
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
: Return dictionary of attributes.

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`f0(=0.0)`
: the (vertical) force acting on the upper plate on the very first time step (determined by the Engine). Controls of the loadings in case of *KinemCNSEngine* or *KinemCNLEngine* will be done according to this initial value [N]. Has to be saved, but not to be changed by the user.

`firstRun(=true)`
: boolean set to false as soon as the engine has done its job one time : useful to know if initial height of, and normal force sustained by, the upper box are known or not (and thus if they have to be initialized). Has to be saved, but not to be changed by the user.

`gamma(=0.0)`
: current value of tangential displacement [m]

`gamma_save(=uninitalized)`
: vector with the values of gamma at which a save of the simulation is performed [m]

`gammalim(=0.0)`
: the value of tangential displacement (of upper plate) at wich the shearing is stopped [m]

`id_boxback(=4)`
: the id of the wall at the back of the sample

`id_boxbas(=1)`
: the id of the lower wall

`id_boxfront(=5)`
: the id of the wall in front of the sample

`id_boxleft(=0)`
: the id of the left wall

`id_boxright(=2)`
: the id of the right wall

`id_topbox`(=*3*)
the id of the upper wall

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`max_vel`(=*1.0*)
to limit the speed of the vertical displacements done to control σ (CNL or CNS cases) [m/s]

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`shearSpeed`(=*0.0*)
the speed at wich the shearing is performed : speed of the upper plate [m/s]

`temoin_save`(=*uninitalized*)
vector (same length as 'gamma_save' for ex), with 0 or 1 depending whether the save for the corresponding value of gamma has been done (1) or not (0). Has to be saved, but not to be changed by the user.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wallDamping`(=*0.2*)
the vertical displacements done to to control σ (CNL or CNS cases) are in fact damped, through this wallDamping

`y0`(=*0.0*)
the height of the upper plate at the very first time step : the engine finds its value [m]. Has to be saved, but not to be changed by the user.

class `yade.wrapper.KinemCNSEngine`(*inherits* *KinemSimpleShearBox* → *BoundaryController* → *GlobalEngine* → *Engine* → *Serializable*)

To apply a Constant Normal Stifness (CNS) shear for a parallelogram box (simple shear)

This engine, useable in simulations implying one deformable parallelepipedic box, allows one to translate horizontally the upper plate while the lateral ones rotate so that they always keep contact with the lower and upper walls. The upper plate can move not only horizontally but also vertically, so that the normal rigidity defined by DeltaF(upper plate)/DeltaU(upper plate) = constant (= *KnC* defined by the user).

The movement is moreover controlled by the user via a *shearSpeed* which is the horizontal speed of the upper wall, and by a maximum value of horizontal displacement *gammalim* (of the upper plate), after which the shear stops.

Note: not only the positions of walls are updated but also their speeds, which is all but useless considering the fact that in the contact laws these velocities of bodies are used to compute values of tangential relative displacements.

Warning: But, because of this last point, if you want to use later saves of simulations executed with this Engine, but without that stopMovement was executed, your boxes will keep their speeds => you will have to cancel them by hand in the .xml

`Key`(="")
string to add at the names of the saved files

`KnC`(=*10.0e6*)
: the normal rigidity chosen by the user [MPa/mm] - the conversion in Pa/m will be made

`LOG`(=*false*)
: boolean controling the output of messages on the screen

`alpha`(=*Mathr::PI/2.0*)
: the angle from the lower box to the left box (trigo wise). Measured by this Engine. Has to be saved, but not to be changed by the user.

`dead`(=*false*)
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
: Return dictionary of attributes.

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`f0`(=*0.0*)
: the (vertical) force acting on the upper plate on the very first time step (determined by the Engine). Controls of the loadings in case of *KinemCNSEngine* or *KinemCNLEngine* will be done according to this initial value [N]. Has to be saved, but not to be changed by the user.

`firstRun`(=*true*)
: boolean set to false as soon as the engine has done its job one time : useful to know if initial height of, and normal force sustained by, the upper box are known or not (and thus if they have to be initialized). Has to be saved, but not to be changed by the user.

`gamma`(=*0.0*)
: current value of tangential displacement [m]

`gammalim`(=*0.0*)
: the value of tangential displacement (of upper plate) at wich the shearing is stopped [m]

`id_boxback`(=*4*)
: the id of the wall at the back of the sample

`id_boxbas`(=*1*)
: the id of the lower wall

`id_boxfront`(=*5*)
: the id of the wall in front of the sample

`id_boxleft`(=*0*)
: the id of the left wall

`id_boxright`(=*2*)
: the id of the right wall

`id_topbox`(=*3*)
: the id of the upper wall

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`max_vel`(=*1.0*)
: to limit the speed of the vertical displacements done to control σ (CNL or CNS cases) [m/s]

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes

openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

shearSpeed(*=0.0*)
the speed at wich the shearing is performed : speed of the upper plate [m/s]

temoin_save(*=uninitalized*)
vector (same length as 'gamma_save' for ex), with 0 or 1 depending whether the save for the corresponding value of gamma has been done (1) or not (0). Has to be saved, but not to be changed by the user.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

wallDamping(*=0.2*)
the vertical displacements done to to control σ (CNL or CNS cases) are in fact damped, through this wallDamping

y0(*=0.0*)
the height of the upper plate at the very first time step : the engine finds its value [m]. Has to be saved, but not to be changed by the user.

class yade.wrapper.**KinemCTDEngine**(*inherits KinemSimpleShearBox → BoundaryController → GlobalEngine → Engine → Serializable*)
To compress a simple shear sample by moving the upper box in a vertical way only, so that the tangential displacement (defined by the horizontal gap between the upper and lower boxes) remains constant (thus, the CTD = Constant Tangential Displacement). The lateral boxes move also to keep always contact. All that until this box is submitted to a given stress (*targetSigma*). Moreover saves are executed at each value of stresses stored in the vector *sigma_save*, and at *targetSigma*

Key(*=""*)
string to add at the names of the saved files

LOG(*=false*)
boolean controling the output of messages on the screen

alpha(*=Mathr::PI/2.0*)
the angle from the lower box to the left box (trigo wise). Measured by this Engine. Has to be saved, but not to be changed by the user.

compSpeed(*=0.0*)
(vertical) speed of the upper box : >0 for real compression, <0 for unloading [m/s]

dead(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

f0(*=0.0*)
the (vertical) force acting on the upper plate on the very first time step (determined by the Engine). Controls of the loadings in case of *KinemCNSEngine* or *KinemCNLEngine* will be done according to this initial value [N]. Has to be saved, but not to be changed by the user.

firstRun(*=true*)
boolean set to false as soon as the engine has done its job one time : useful to know if initial

height of, and normal force sustained by, the upper box are known or not (and thus if they have to be initialized). Has to be saved, but not to be changed by the user.

`id_boxback`(=*4*)
: the id of the wall at the back of the sample

`id_boxbas`(=*1*)
: the id of the lower wall

`id_boxfront`(=*5*)
: the id of the wall in front of the sample

`id_boxleft`(=*0*)
: the id of the left wall

`id_boxright`(=*2*)
: the id of the right wall

`id_topbox`(=*3*)
: the id of the upper wall

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`max_vel`(=*1.0*)
: to limit the speed of the vertical displacements done to control σ (CNL or CNS cases) [m/s]

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`sigma_save`(=*uninitalized*)
: vector with the values of sigma at which a save of the simulation should be performed [kPa]

`targetSigma`(=*0.0*)
: the value of sigma at which the compression should stop [kPa]

`temoin_save`(=*uninitalized*)
: vector (same length as 'gamma_save' for ex), with 0 or 1 depending whether the save for the corresponding value of gamma has been done (1) or not (0). Has to be saved, but not to be changed by the user.

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

`wallDamping`(=*0.2*)
: the vertical displacements done to to control σ (CNL or CNS cases) are in fact damped, through this wallDamping

`y0`(=*0.0*)
: the height of the upper plate at the very first time step : the engine finds its value [m]. Has to be saved, but not to be changed by the user.

class `yade.wrapper.KinemSimpleShearBox`(*inherits BoundaryController → GlobalEngine → Engine → Serializable*)

This class is supposed to be a mother class for all Engines performing loadings on the simple shear box of *SimpleShear*. It is not intended to be used by itself, but its declaration and implentation will thus contain all what is useful for all these Engines. The script simpleShear.py illustrates the use of the various corresponding Engines.

`Key(="")`
string to add at the names of the saved files

`LOG(=false)`
boolean controling the output of messages on the screen

`alpha(=Mathr::PI/2.0)`
the angle from the lower box to the left box (trigo wise). Measured by this Engine. Has to be saved, but not to be changed by the user.

`dead(=false)`
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict() → dict`
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`f0(=0.0)`
the (vertical) force acting on the upper plate on the very first time step (determined by the Engine). Controls of the loadings in case of *KinemCNSEngine* or *KinemCNLEngine* will be done according to this initial value [N]. Has to be saved, but not to be changed by the user.

`firstRun(=true)`
boolean set to false as soon as the engine has done its job one time : useful to know if initial height of, and normal force sustained by, the upper box are known or not (and thus if they have to be initialized). Has to be saved, but not to be changed by the user.

`id_boxback(=4)`
the id of the wall at the back of the sample

`id_boxbas(=1)`
the id of the lower wall

`id_boxfront(=5)`
the id of the wall in front of the sample

`id_boxleft(=0)`
the id of the left wall

`id_boxright(=2)`
the id of the right wall

`id_topbox(=3)`
the id of the upper wall

`label(=uninitalized)`
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`max_vel(=1.0)`
to limit the speed of the vertical displacements done to control σ (CNL or CNS cases) [m/s]

`ompThreads(=-1)`
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`temoin_save(=uninitalized)`
vector (same length as 'gamma_save' for ex), with 0 or 1 depending whether the save for the

corresponding value of gamma has been done (1) or not (0). Has to be saved, but not to be changed by the user.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wallDamping`(*=0.2*)
the vertical displacements done to to control σ (CNL or CNS cases) are in fact damped, through this wallDamping

`y0`(*=0.0*)
the height of the upper plate at the very first time step : the engine finds its value [m]. Has to be saved, but not to be changed by the user.

class `yade.wrapper.Peri3dController`(*inherits BoundaryController → GlobalEngine → Engine → Serializable*)

Class for controlling independently all 6 components of "engineering" *stress* and *strain* of periodic *Cell*. *goal* are the goal values, while *stressMask* determines which components prescribe stress and which prescribe strain.

If the strain is prescribed, appropriate strain rate is directly applied. If the stress is prescribed, the strain predictor is used: from stress values in two previous steps the value of strain rate is prescribed so as the value of stress in the next step is as close as possible to the ideal one. Current algorithm is extremly simple and probably will be changed in future, but is roboust enough and mostly works fine.

Stress error (difference between actual and ideal stress) is evaluated in current and previous steps $(d\sigma_i, d\sigma_{i-1})$. Linear extrapolation is used to estimate error in the next step

$$d\sigma_{i+1} = 2d\sigma_i - d\sigma_{i-1}$$

According to this error, the strain rate is modified by *mod* parameter

$$d\sigma_{i+1} \begin{cases} > 0 \rightarrow \dot{\varepsilon}_{i+1} = \dot{\varepsilon}_i - \max(\mathrm{abs}(\dot{\varepsilon}_i)) \cdot \mathrm{mod} \\ < 0 \rightarrow \dot{\varepsilon}_{i+1} = \dot{\varepsilon}_i + \max(\mathrm{abs}(\dot{\varepsilon}_i)) \cdot \mathrm{mod} \end{cases}$$

According to this fact, the prescribed stress will (almost) never have exact prescribed value, but the difference would be very small (and decreasing for increasing *nSteps*. This approach works good if one of the dominant strain rates is prescribed. If all stresses are prescribed or if all goal strains is prescribed as zero, a good estimation is needed for the first step, therefore the compliance matrix is estimated (from user defined estimations of macroscopic material parameters *youngEstimation* and *poissonEstimation*) and respective strain rates is computed form prescribed stress rates and compliance matrix (the estimation of compliance matrix could be computed autamatically avoiding user inputs of this kind).

The simulation on rotated periodic cell is also supported. Firstly, the polar decomposition is performed on cell's transformation matrix *trsf* $\mathcal{T} = \mathbf{U}\mathbf{P}$, where $\mathbf{U}$ is orthogonal (unitary) matrix representing rotation and $\mathbf{P}$ is a positive semi-definite Hermitian matrix representing strain. A logarithm of $\mathbf{P}$ should be used to obtain realistic values at higher strain values (not implemented yet). A prescribed strain increment in global coordinates $dt \cdot \dot{\boldsymbol{\varepsilon}}$ is properly rotated to cell's local coordinates and added to $\mathbf{P}$

$$\mathbf{P}_{i+1} = \mathbf{P} + \mathbf{U}^T dt \cdot \dot{\boldsymbol{\varepsilon}} \mathbf{U}$$

The new value of *trsf* is computed at $\mathbf{T}_{i+1} = \mathbf{U}\mathbf{P}_{i+1}$. From current and next *trsf* the cell's velocity gradient *velGrad* is computed (according to its definition) as

$$\boldsymbol{V} = (\mathbf{T}_{i+1}\mathbf{T}^{-1} - \mathbf{I})/dt$$

Current implementation allow user to define independent loading "path" for each prescribed component. i.e. define the prescribed value as a function of time (or *progress* or steps). See *Paths*.

Examples examples/test/peri3dController_example1.py and examples/test/peri3dController_triaxialCompression.py explain usage and inputs of Peri3dController, examples/test/peri3dController_shear.py is an example of using shear components and also simulation on rotated cell.

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

doneHook(=*uninitalized*)
Python command (as string) to run when *nSteps* is achieved. If empty, the engine will be set *dead*.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

goal(=*Vector6r::Zero()*)
Goal state; only the upper triangular matrix is considered; each component is either prescribed stress or strain, depending on *stressMask*.

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

lenPe(=*0*)
Peri3dController internal variable

lenPs(=*0*)
Peri3dController internal variable

maxStrain(=*1e6*)
Maximal asolute value of *strain* allowed in the simulation. If reached, the simulation is considered as finished

maxStrainRate(=*1e3*)
Maximal absolute value of strain rate (both normal and shear components of *strain*)

mod(=*.1*)
Predictor modificator, by trail-and-error analysis the value 0.1 was found as the best.

nSteps(=*1000*)
Number of steps of the simulation.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

pathSizes(=*Vector6i::Zero()*)
Peri3dController internal variable

pathsCounter(=*Vector6i::Zero()*)
Peri3dController internal variable

pe(=*Vector6i::Zero()*)
Peri3dController internal variable

poissonEstimation(=*.25*)
Estimation of macroscopic Poisson's ratio, used used for the first simulation step

progress(=*0.*)
Actual progress of the simulation with Controller.

ps(=*Vector6i::Zero()*)
Peri3dController internal variable

strain(=*Vector6r::Zero()*)
Current strain (deformation) vector (ε_x,ε_y,ε_z,γ_{yz},γ_{zx},γ_{xy}) *(auto-updated).*

strainGoal(=*Vector6r::Zero()*)
Peri3dController internal variable

strainRate(=*Vector6r::Zero()*)
Current strain rate vector.

stress(=*Vector6r::Zero()*)
Current stress vector (σ_x,σ_y,σ_z,τ_{yz},τ_{zx},τ_{xy})|yupdate|.

stressGoal(=*Vector6r::Zero()*)
Peri3dController internal variable

stressIdeal(=*Vector6r::Zero()*)
Ideal stress vector at current time step.

stressMask(=*0*, *all strains*)
mask determining whether components of *goal* are strain (0) or stress (1). The order is 00,11,22,12,02,01 from the least significant bit. (e.g. 0b000011 is stress 00 and stress 11).

stressRate(=*Vector6r::Zero()*)
Current stress rate vector (that is prescribed, the actual one slightly differ).

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

xxPath
"Time function" (piecewise linear) for xx direction. Sequence of couples of numbers. First number is time, second number desired value of respective quantity (stress or strain). The last couple is considered as final state (equal to (*nSteps*, *goal*)), other values are relative to this state.

Example: nSteps=1000, goal[0]=300, xxPath=((2,3),(4,1),(5,2))

at step 400 (=5*1000/2) the value is 450 (=3*300/2),

at step 800 (=4*1000/5) the value is 150 (=1*300/2),

at step 1000 (=5*1000/5=nSteps) the value is 300 (=2*300/2=goal[0]).

See example scripts/test/peri3dController_example1 for illusration.

xyPath(=*vector<Vector2r>(1, Vector2r::Ones())*)
Time function for xy direction, see *xxPath*

youngEstimation(=*1e20*)
Estimation of macroscopic Young's modulus, used for the first simulation step

yyPath(=*vector<Vector2r>(1, Vector2r::Ones())*)
Time function for yy direction, see *xxPath*

yzPath(=*vector<Vector2r>(1, Vector2r::Ones())*)
Time function for yz direction, see *xxPath*

zxPath(=*vector<Vector2r>(1, Vector2r::Ones())*)
Time function for zx direction, see *xxPath*

zzPath(=*vector<Vector2r>(1, Vector2r::Ones())*)
Time function for zz direction, see *xxPath*

class `yade.wrapper.PeriIsoCompressor`(*inherits* *BoundaryController* → *GlobalEngine* → *Engine* → *Serializable*)

Compress/decompress cloud of spheres by controlling periodic cell size until it reaches prescribed average stress, then moving to next stress value in given stress series.

`charLen`(*=-1.*)
: Characteristic length, should be something like mean particle diameter (default -1=invalid value))

`currUnbalanced`
: Current value of unbalanced force

`dead`(*=false*)
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
: Return dictionary of attributes.

`doneHook`(*=""*)
: Python command to be run when reaching the last specified stress

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`globalUpdateInt`(*=20*)
: how often to recompute average stress, stiffness and unbalanced force

`keepProportions`(*=true*)
: Exactly keep proportions of the cell (stress is controlled based on average, not its components

`label`(*=uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`maxSpan`(*=-1.*)
: Maximum body span in terms of bbox, to prevent periodic cell getting too small. *(auto-computed)*

`maxUnbalanced`(*=1e-4*)
: if actual unbalanced force is smaller than this number, the packing is considered stable,

`ompThreads`(*=-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`sigma`
: Current stress value

`state`(*=0*)
: Where are we at in the stress series

`stresses`(*=uninitalized*)
: Stresses that should be reached, one after another

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

class `yade.wrapper.PeriTriaxController`(*inherits BoundaryController → GlobalEngine → Engine → Serializable*)

Engine for independently controlling stress or strain in periodic simulations.

`strainStress` contains absolute values for the controlled quantity, and `stressMask` determines meaning of those values (0 for strain, 1 for stress): e.g. `( 1<<0 | 1<<2 ) = 1 | 4 = 5` means that `strainStress[0]` and `strainStress[2]` are stress values, and `strainStress[1]` is strain.

See scripts/test/periodic-triax.py for a simple example.

`absStressTol`(*=1e3*)
: Absolute stress tolerance

`currUnbalanced`(*=NaN*)
: current unbalanced force (updated every globUpdate) *(auto-updated)*

`dead`(*=false*)
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
: Return dictionary of attributes.

`doneHook`(*=uninitalized*)
: python command to be run when the desired state is reached

`dynCell`(*=false*)
: Imposed stress can be controlled using the packing stiffness or by applying the laws of dynamic (dynCell=true). Don't forget to assign a *mass* to the cell.

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`externalWork`(*=0*)
: Work input from boundary controller.

`globUpdate`(*=5*)
: How often to recompute average stress, stiffness and unbalaced force.

`goal`
: Desired stress or strain values (depending on stressMask), strains defined as `strain(i)=log(Fii)`.

> **Warning:** Strains are relative to the *O.cell.refSize* (reference cell size), not the current one (e.g. at the moment when the new strain value is set).

`growDamping`(*=.25*)
: Damping of cell resizing (0=perfect control, 1=no control at all); see also `wallDamping` in *TriaxialStressController*.

`label`(*=uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`mass`(*=NaN*)
: mass of the cell (user set); if not set and *dynCell* is used, it will be computed as sum of masses of all particles.

`maxBodySpan`(*=Vector3r::Zero()*)
: maximum body dimension *(auto-computed)*

`maxStrainRate`(*=Vector3r(1, 1, 1)*)
: Maximum strain rate of the periodic cell.

`maxUnbalanced`(*=1e-4*)
: maximum unbalanced force.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

prevGrow(=*Vector3r::Zero()*)
previous cell grow

relStressTol(=*3e-5*)
Relative stress tolerance

stiff(=*Vector3r::Zero()*)
average stiffness (only every globUpdate steps recomputed from interactions) *(auto-updated)*

strain(=*Vector3r::Zero()*)
cell strain *(auto-updated)*

strainRate(=*Vector3r::Zero()*)
cell strain rate *(auto-updated)*

stress(=*Vector3r::Zero()*)
diagonal terms of the stress tensor

stressMask(=*0*, *all strains*)
mask determining strain/stress (0/1) meaning for goal components

stressTensor(=*Matrix3r::Zero()*)
average stresses, updated at every step (only every globUpdate steps recomputed from interactions if !dynCell)

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**ThreeDTriaxialEngine**(*inherits TriaxialStressController → BoundaryController → GlobalEngine → Engine → Serializable*)
The engine perform a triaxial compression with a control in direction 'i' in stress (if stressControl_i) else in strain.

For a stress control the imposed stress is specified by 'sigma_i' with a 'max_veli' depending on 'strainRatei'. To obtain the same strain rate in stress control than in strain control you need to set 'wallDamping = 0.8'. For a strain control the imposed strain is specified by 'strainRatei'. With this engine you can also perform internal compaction by growing the size of particles by using `TriaxialStressController::controlInternalStress`. For that, just switch on 'internalCompaction=1' and fix sigma_iso=value of mean pressure that you want at the end of the internal compaction.

> **Warning:** This engine is deprecated, please switch to TriaxialStressController if you expect long term support.

Key(=*""*)
A string appended at the end of all files, use it to name simulations.

UnbalancedForce(=*1*)
mean resultant forces divided by mean contact force

boxVolume
Total packing volume.

computeStressStrainInterval(=*10*)

currentStrainRate1(=*0*)
current strain rate in direction 1 - converging to *ThreeDTriaxialEngine::strainRate1* (./s)

currentStrainRate2(=*0*)
current strain rate in direction 2 - converging to *ThreeDTriaxialEngine::strainRate2* (./s)

currentStrainRate3(=*0*)
current strain rate in direction 3 - converging to *ThreeDTriaxialEngine::strainRate3* (./s)

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

depth(=*0*)
size of the box (2-axis) *(auto-updated)*

depth0(=*0*)
Reference size for strain definition. See *TriaxialStressController::depth*

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

externalWork(=*0*)
Energy provided by boundaries.|yupdate|

finalMaxMultiplier(=*1.00001*)
max multiplier of diameters during internal compaction (secondary precise adjustment - *TriaxialStressController::maxMultiplier* is used in the initial stage)

frictionAngleDegree(=*-1*)
Value of friction used in the simulation if (updateFrictionAngle)

goal1(=*0*)
prescribed stress/strain rate on axis 1, as defined by *TriaxialStressController::stressMask*

goal2(=*0*)
prescribed stress/strain rate on axis 2, as defined by *TriaxialStressController::stressMask*

goal3(=*0*)
prescribed stress/strain rate on axis 3, as defined by *TriaxialStressController::stressMask*

height(=*0*)
size of the box (1-axis) *(auto-updated)*

height0(=*0*)
Reference size for strain definition. See *TriaxialStressController::height*

internalCompaction(=*true*)
Switch between 'external' (walls) and 'internal' (growth of particles) compaction.

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxMultiplier(=*1.001*)
max multiplier of diameters during internal compaction (initial fast increase - *TriaxialStressController::finalMaxMultiplier* is used in a second stage)

max_vel(=*1*)
Maximum allowed walls velocity [m/s]. This value superseeds the one assigned by the stress controller if the later is higher. max_vel can be set to infinity in many cases, but sometimes helps stabilizing packings. Based on this value, different maxima are computed for each axis based on the dimensions of the sample, so that if each boundary moves at its maximum velocity, the strain rate will be isotropic (see e.g. *TriaxialStressController::max_vel1*).

max_vel1
see *TriaxialStressController::max_vel* *(auto-computed)*

max_vel2
see *TriaxialStressController::max_vel* *(auto-computed)*

max_vel3
see *TriaxialStressController::max_vel* *(auto-computed)*

meanStress(*=0*)
Mean stress in the packing. *(auto-updated)*

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

particlesVolume
Total volume of particles (clumps and spheres).

porosity
Porosity of the packing.

previousMultiplier(*=1*)
(auto-updated)

previousStress(*=0*)
(auto-updated)

radiusControlInterval(*=10*)

setContactProperties(*(float)arg2*) → None
Assign a new friction angle (degrees) to dynamic bodies and relative interactions

spheresVolume
Shorthand for *TriaxialStressController::particlesVolume*

stiffnessUpdateInterval(*=10*)
target strain rate (./s)

strain
Current strain in a vector (exx,eyy,ezz). The values reflect true (logarithmic) strain.

strainDamping(*=0.9997*)
factor used for smoothing changes in effective strain rate. If target rate is TR, then (1-damping)*(TR-currentRate) will be added at each iteration. With damping=0, rate=target all the time. With damping=1, it doesn't change.

strainRate
Current strain rate in a vector d/dt(exx,eyy,ezz).

strainRate1(*=0*)
target strain rate in direction 1 (./s, >0 for compression)

strainRate2(*=0*)
target strain rate in direction 2 (./s, >0 for compression)

strainRate3(*=0*)
target strain rate in direction 3 (./s, >0 for compression)

stress(*(int)id*) → Vector3
Returns the average stress on boundary 'id'. Here, 'id' refers to the internal numbering of boundaries, between 0 and 5.

stressControl_1(*=true*)
Switch to choose a stress or a strain control in directions 1

stressControl_2(=*true*)
Switch to choose a stress or a strain control in directions 2

stressControl_3(=*true*)
Switch to choose a stress or a strain control in directions 3

stressDamping(=*0.25*)
wall damping coefficient for the stress control - wallDamping=0 implies a (theoretical) perfect control, wallDamping=1 means no movement

stressMask(=*7*)
Bitmask determining wether the imposed *TriaxialStressController::goal's are stresses (0 for none, 7 for all, 1 for direction 1, 5 for directions 1 and 3, etc. :ydefault:'7*

thickness(=*-1*)
thickness of boxes (needed by some functions)

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updateFrictionAngle(=*false*)
Switch to activate the update of the intergranular frictionto the value *ThreeDTriaxialEngine::frictionAngleDegree*.

updatePorosity(=*false*)
If true porosity calculation will be updated once (will automatically reset to false after one calculation step). Can be used, when volume of particles changes during the simulation (e.g. when particles are erased or when clumps are created).

volumetricStrain(=*0*)
Volumetric strain (see *TriaxialStressController::strain*).|yupdate|

wall_back_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_back_id(=*4*)
id of boundary ; coordinate 2- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_bottom_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_bottom_id(=*2*)
id of boundary ; coordinate 1- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_front_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_front_id(=*5*)
id of boundary ; coordinate 2+ (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_left_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_left_id(=*0*)
id of boundary ; coordinate 0- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_right_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

`wall_right_id`(=*1*)
: id of boundary ; coordinate 0+ (default value is ok if aabbWalls are appended BEFORE spheres.)

`wall_top_activated`(=*true*)
: if true, this wall moves according to the target value (stress or strain rate).

`wall_top_id`(=*3*)
: id of boundary ; coordinate 1+ (default value is ok if aabbWalls are appended BEFORE spheres.)

`width`(=*0*)
: size of the box (0-axis) *(auto-updated)*

`width0`(=*0*)
: Reference size for strain definition. See *TriaxialStressController::width*

class `yade.wrapper.TriaxialCompressionEngine`(*inherits* *TriaxialStressController* → *BoundaryController* → *GlobalEngine* → *Engine* → *Serializable*)

The engine is a state machine with the following states; transitions my be automatic, see below.

1. STATE_ISO_COMPACTION: isotropic compaction (compression) until the prescribed mean pressue sigmaIsoCompaction is reached and the packing is stable. The compaction happens either by straining the walls (!internalCompaction) or by growing size of grains (internalCompaction).
2. STATE_ISO_UNLOADING: isotropic unloading from the previously reached state, until the mean pressure sigmaLateralConfinement is reached (and stabilizes).

   Note: this state will be skipped if sigmaLateralConfinement == sigmaIsoCompaction.

   ---

3. STATE_TRIAX_LOADING: confined uniaxial compression: constant sigmaLateralConfinement is kept at lateral walls (left, right, front, back), while top and bottom walls load the packing in their axis (by straining), until the value of epsilonMax (deformation along the loading axis) is reached. At this point, the simulation is stopped.
4. STATE_FIXED_POROSITY_COMPACTION: isotropic compaction (compression) until a chosen porosity value (parameter:fixedPorosity). The six walls move with a chosen translation speed (parameter StrainRate).
5. STATE_TRIAX_LIMBO: currently unused, since simulation is hard-stopped in the previous state.

Transition from COMPACTION to UNLOADING is done automatically if autoUnload==true;

Transition from (UNLOADING to LOADING) or from (COMPACTION to LOADING: if UNLOADING is skipped) is done automatically if autoCompressionActivation=true; Both autoUnload and autoCompressionActivation are true by default.

Note: Most of the algorithms used have been developed initialy for simulations reported in *[Chareyre2002a]* and *[Chareyre2005]*. They have been ported to Yade in a second step and used in e.g. *[Kozicki2008]*,[Scholtes2009b]_,[Jerier2010b].

Warning: This engine is deprecated, please switch to TriaxialStressController if you expect long term support.

`Key`(=*""*)
: A string appended at the end of all files, use it to name simulations.

`StabilityCriterion`(=*0.001*)
: tolerance in terms of *TriaxialCompressionEngine::UnbalancedForce* to consider the packing is stable

`UnbalancedForce(=1)`
mean resultant forces divided by mean contact force

`autoCompressionActivation(=true)`
Auto-switch from isotropic compaction (or unloading state if sigmaLateralConfinement<sigmaIsoCompaction) to deviatoric loading

`autoStopSimulation(=false)`
Stop the simulation when the sample reach STATE_LIMBO, or keep running

`autoUnload(=true)`
Auto-switch from isotropic compaction to unloading

`boxVolume`
Total packing volume.

`computeStressStrainInterval(=10)`

`currentState(=1)`
There are 5 possible states in which TriaxialCompressionEngine can be. See above *wrapper.TriaxialCompressionEngine*

`currentStrainRate(=0)`
current strain rate - converging to *TriaxialCompressionEngine::strainRate* (./s)

`dead(=false)`
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`depth(=0)`
size of the box (2-axis) *(auto-updated)*

`depth0(=0)`
Reference size for strain definition. See *TriaxialStressController::depth*

`dict() → dict`
Return dictionary of attributes.

`epsilonMax(=0.5)`
Value of axial deformation for which the loading must stop

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`externalWork(=0)`
Energy provided by boundaries.|yupdate|

`finalMaxMultiplier(=1.00001)`
max multiplier of diameters during internal compaction (secondary precise adjustment - *TriaxialStressController::maxMultiplier* is used in the initial stage)

`fixedPoroCompaction(=false)`
A special type of compaction with imposed final porosity *TriaxialCompressionEngine::fixedPorosity* (WARNING : can give unrealistic results!)

`fixedPorosity(=0)`
Value of porosity chosen by the user

`frictionAngleDegree(=-1)`
Value of friction assigned just before the deviatoric loading

`goal1(=0)`
prescribed stress/strain rate on axis 1, as defined by *TriaxialStressController::stressMask*

`goal2(=0)`
prescribed stress/strain rate on axis 2, as defined by *TriaxialStressController::stressMask*

goal3(=*0*)
prescribed stress/strain rate on axis 3, as defined by *TriaxialStressController::stressMask*

height(=*0*)
size of the box (1-axis) *(auto-updated)*

height0(=*0*)
Reference size for strain definition. See *TriaxialStressController::height*

internalCompaction(=*true*)
Switch between 'external' (walls) and 'internal' (growth of particles) compaction.

isAxisymetric(=*false*)
if true, sigma_iso is assigned to sigma1, 2 and 3 (applies at each iteration and overrides user-set values of s1,2,3)

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxMultiplier(=*1.001*)
max multiplier of diameters during internal compaction (initial fast increase - *TriaxialStressController::finalMaxMultiplier* is used in a second stage)

maxStress(=*0*)
Max absolute value of axial stress during the simulation (for post-processing)

max_vel(=*1*)
Maximum allowed walls velocity [m/s]. This value superseeds the one assigned by the stress controller if the later is higher. max_vel can be set to infinity in many cases, but sometimes helps stabilizing packings. Based on this value, different maxima are computed for each axis based on the dimensions of the sample, so that if each boundary moves at its maximum velocity, the strain rate will be isotropic (see e.g. *TriaxialStressController::max_vel1*).

max_vel1
see *TriaxialStressController::max_vel* *(auto-computed)*

max_vel2
see *TriaxialStressController::max_vel* *(auto-computed)*

max_vel3
see *TriaxialStressController::max_vel* *(auto-computed)*

meanStress(=*0*)
Mean stress in the packing. *(auto-updated)*

noFiles(=*false*)
If true, no files will be generated (*.xml, *.spheres,...)

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

particlesVolume
Total volume of particles (clumps and spheres).

porosity
Porosity of the packing.

previousMultiplier(=*1*)
(auto-updated)

previousSigmaIso(=*1*)
Previous value of inherited sigma_iso (used to detect manual changes of the confining pressure)

previousState(*=1*)
Previous state (used to detect manual changes of the state in .xml)

previousStress(*=0*)
(auto-updated)

radiusControlInterval(*=10*)

setContactProperties(*(float)arg2*) → None
Assign a new friction angle (degrees) to dynamic bodies and relative interactions

sigmaIsoCompaction(*=1*)
Prescribed isotropic pressure during the compaction phase (< 0 for real - compressive - compaction)

sigmaLateralConfinement(*=1*)
Prescribed confining pressure in the deviatoric loading (< 0 for classical compressive cases); might be different from *TriaxialCompressionEngine::sigmaIsoCompaction*

sigma_iso(*=0*)
prescribed confining stress (see :yref:TriaxialCompressionEngine::isAxisymetric‘)

spheresVolume
Shorthand for *TriaxialStressController::particlesVolume*

stiffnessUpdateInterval(*=10*)
target strain rate (./s)

strain
Current strain in a vector (exx,eyy,ezz). The values reflect true (logarithmic) strain.

strainDamping(*=0.99*)
coefficient used for smoother transitions in the strain rate. The rate reaches the target value like $\mathtt{d}^{\mathtt{n}}$ reaches 0, where $\mathtt{d}$ is the damping coefficient and $\mathtt{n}$ is the number of steps

strainRate(*=0*)
target strain rate (./s, >0 for compression)

stress(*(int)id*) → Vector3
Returns the average stress on boundary ‘id’. Here, ‘id’ refers to the internal numbering of boundaries, between 0 and 5.

stressDamping(*=0.25*)
wall damping coefficient for the stress control - wallDamping=0 implies a (theoretical) perfect control, wallDamping=1 means no movement

stressMask(*=7*)
Bitmask determining wether the imposed *TriaxialStressController::goal‘s are stresses (0 for none, 7 for all, 1 for direction 1, 5 for directions 1 and 3, etc. :ydefault:‘7*

testEquilibriumInterval(*=20*)
interval of checks for transition between phases, higher than 1 saves computation time.

thickness(*=-1*)
thickness of boxes (needed by some functions)

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

translationAxis(*=TriaxialStressController::normal[wall_bottom]*)
compression axis

uniaxialEpsilonCurr(*=1*)
Current value of axial deformation during confined loading (is reference to strain[1])

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updatePorosity(=*false*)
If true porosity calculation will be updated once (will automatically reset to false after one calculation step). Can be used, when volume of particles changes during the simulation (e.g. when particles are erased or when clumps are created).

volumetricStrain(=*0*)
Volumetric strain (see *TriaxialStressController::strain*).|yupdate|

wall_back_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_back_id(=*4*)
id of boundary ; coordinate 2- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_bottom_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_bottom_id(=*2*)
id of boundary ; coordinate 1- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_front_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_front_id(=*5*)
id of boundary ; coordinate 2+ (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_left_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_left_id(=*0*)
id of boundary ; coordinate 0- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_right_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_right_id(=*1*)
id of boundary ; coordinate 0+ (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_top_activated(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

wall_top_id(=*3*)
id of boundary ; coordinate 1+ (default value is ok if aabbWalls are appended BEFORE spheres.)

warn(=*0*)
counter used for sending a deprecation warning once

width(=*0*)
size of the box (0-axis) *(auto-updated)*

width0(=*0*)
Reference size for strain definition. See *TriaxialStressController::width*

class yade.wrapper.**TriaxialStressController**(*inherits* *BoundaryController* → *GlobalEngine* → *Engine* → *Serializable*)
An engine maintaining constant stresses or constant strain rates on some boundaries of a parallepipedic packing. The stress/strain control is defined for each axis using *TriaxialStressController::stressMask* (a bitMask) and target values are defined by goal1,goal2, and goal3. The sign conventions of continuum mechanics are used for strains and stresses (positive traction).

Note: The algorithms used have been developed initialy for simulations reported in

[Chareyre2002a] and *[Chareyre2005]*. They have been ported to Yade in a second step and used in e.g. *[Kozicki2008]*,[Scholtes2009b]_,[Jerier2010b].

---

boxVolume
Total packing volume.

computeStressStrainInterval(*=10*)

dead(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

depth(*=0*)
size of the box (2-axis) *(auto-updated)*

depth0(*=0*)
Reference size for strain definition. See *TriaxialStressController::depth*

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

externalWork(*=0*)
Energy provided by boundaries.|yupdate|

finalMaxMultiplier(*=1.00001*)
max multiplier of diameters during internal compaction (secondary precise adjustment - *TriaxialStressController::maxMultiplier* is used in the initial stage)

goal1(*=0*)
prescribed stress/strain rate on axis 1, as defined by *TriaxialStressController::stressMask*

goal2(*=0*)
prescribed stress/strain rate on axis 2, as defined by *TriaxialStressController::stressMask*

goal3(*=0*)
prescribed stress/strain rate on axis 3, as defined by *TriaxialStressController::stressMask*

height(*=0*)
size of the box (1-axis) *(auto-updated)*

height0(*=0*)
Reference size for strain definition. See *TriaxialStressController::height*

internalCompaction(*=true*)
Switch between 'external' (walls) and 'internal' (growth of particles) compaction.

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxMultiplier(*=1.001*)
max multiplier of diameters during internal compaction (initial fast increase - *TriaxialStressController::finalMaxMultiplier* is used in a second stage)

max_vel(*=1*)
Maximum allowed walls velocity [m/s]. This value superseeds the one assigned by the stress controller if the later is higher. max_vel can be set to infinity in many cases, but sometimes helps stabilizing packings. Based on this value, different maxima are computed for each axis based on the dimensions of the sample, so that if each boundary moves at its maximum velocity, the strain rate will be isotropic (see e.g. *TriaxialStressController::max_vel1*).

max_vel1
see *TriaxialStressController::max_vel* *(auto-computed)*

max_vel2
see *TriaxialStressController::max_vel* *(auto-computed)*

max_vel3
see *TriaxialStressController::max_vel* *(auto-computed)*

meanStress(*=0*)
Mean stress in the packing. *(auto-updated)*

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

particlesVolume
Total volume of particles (clumps and spheres).

porosity
Porosity of the packing.

previousMultiplier(*=1*)
(auto-updated)

previousStress(*=0*)
(auto-updated)

radiusControlInterval(*=10*)

spheresVolume
Shorthand for *TriaxialStressController::particlesVolume*

stiffnessUpdateInterval(*=10*)
target strain rate (./s)

strain
Current strain in a vector (exx,eyy,ezz). The values reflect true (logarithmic) strain.

strainDamping(*=0.99*)
coefficient used for smoother transitions in the strain rate. The rate reaches the target value like $\mathtt{d}^{\mathtt{n}}$ reaches 0, where $\mathtt{d}$ is the damping coefficient and $\mathtt{n}$ is the number of steps

strainRate
Current strain rate in a vector d/dt(exx,eyy,ezz).

stress(*(int)id*) → Vector3
Returns the average stress on boundary 'id'. Here, 'id' refers to the internal numbering of boundaries, between 0 and 5.

stressDamping(*=0.25*)
wall damping coefficient for the stress control - wallDamping=0 implies a (theoretical) perfect control, wallDamping=1 means no movement

stressMask(*=7*)
Bitmask determining wether the imposed *TriaxialStressController::goal's are stresses (0 for none, 7 for all, 1 for direction 1, 5 for directions 1 and 3, etc. :ydefault:'7*

thickness(*=-1*)
thickness of boxes (needed by some functions)

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

`updatePorosity`(=*false*)
If true porosity calculation will be updated once (will automatically reset to false after one calculation step). Can be used, when volume of particles changes during the simulation (e.g. when particles are erased or when clumps are created).

`volumetricStrain`(=*0*)
Volumetric strain (see *TriaxialStressController::strain*).|yupdate|

`wall_back_activated`(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

`wall_back_id`(=*4*)
id of boundary ; coordinate 2- (default value is ok if aabbWalls are appended BEFORE spheres.)

`wall_bottom_activated`(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

`wall_bottom_id`(=*2*)
id of boundary ; coordinate 1- (default value is ok if aabbWalls are appended BEFORE spheres.)

`wall_front_activated`(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

`wall_front_id`(=*5*)
id of boundary ; coordinate 2+ (default value is ok if aabbWalls are appended BEFORE spheres.)

`wall_left_activated`(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

`wall_left_id`(=*0*)
id of boundary ; coordinate 0- (default value is ok if aabbWalls are appended BEFORE spheres.)

`wall_right_activated`(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

`wall_right_id`(=*1*)
id of boundary ; coordinate 0+ (default value is ok if aabbWalls are appended BEFORE spheres.)

`wall_top_activated`(=*true*)
if true, this wall moves according to the target value (stress or strain rate).

`wall_top_id`(=*3*)
id of boundary ; coordinate 1+ (default value is ok if aabbWalls are appended BEFORE spheres.)

`width`(=*0*)
size of the box (0-axis) *(auto-updated)*

`width0`(=*0*)
Reference size for strain definition. See *TriaxialStressController::width*

class `yade.wrapper.UniaxialStrainer`(*inherits* *BoundaryController* → *GlobalEngine* → *Engine* → *Serializable*)
Axial displacing two groups of bodies in the opposite direction with given strain rate.

`absSpeed`(=*NaN*)
alternatively, absolute speed of boundary motion can be specified; this is effective only at the beginning and if strainRate is not set; changing absSpeed directly during simulation wil have no effect. [ms ^{1}]

`active`(=*true*)
Whether this engine is activated

asymmetry(=*0*, *symmetric*)
If 0, straining is symmetric for negIds and posIds; for 1 (or -1), only posIds are strained and negIds don't move (or vice versa)

avgStress(=*0*)
Current average stress *(auto-updated)* [Pa]

axis(=*2*)
The axis which is strained (0,1,2 for x,y,z)

blockDisplacements(=*false*)
Whether displacement of boundary bodies perpendicular to the strained axis are blocked or are free

blockRotations(=*false*)
Whether rotations of boundary bodies are blocked.

crossSectionArea(=*NaN*)
crossSection perpendicular to he strained axis; must be given explicitly [m^2]

currentStrainRate(=*NaN*)
Current strain rate (update automatically). *(auto-updated)*

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

idleIterations(=*0*)
Number of iterations that will pass without straining activity after stopStrain has been reached

initAccelTime(=*-200*)
Time for strain reaching the requested value (linear interpolation). If negative, the time is dt*(-initAccelTime), where dt is the timestep at the first iteration. [s]

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

limitStrain(=*0*, *disabled*)
Invert the sense of straining (sharply, without transition) one this value of strain is reached. Not effective if 0.

negIds(=*uninitalized*)
Bodies on which strain will be applied (on the negative end along the axis)

notYetReversed(=*true*)
Flag whether the sense of straining has already been reversed (only used internally).

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

originalLength(=*NaN*)
Distance of reference bodies in the direction of axis before straining started (computed automatically) [m]

`posIds`(=*uninitalized*)
Bodies on which strain will be applied (on the positive end along the axis)

`setSpeeds`(=*false*)
should we set speeds at the beginning directly, instead of increasing strain rate progressively?

`stopStrain`(=*NaN*)
Strain at which we will pause simulation; inactive (nan) by default; must be reached from below (in absolute value)

`strain`(=*0*)
Current strain value, elongation/originalLength *(auto-updated)* [-]

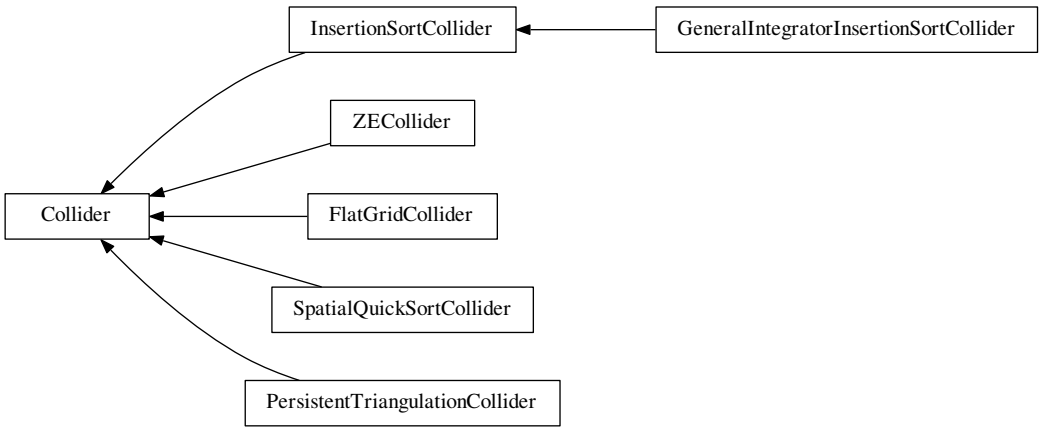
`strainRate`(=*NaN*)
Rate of strain, starting at 0, linearly raising to strainRate. [-]

`stressUpdateInterval`(=*10*)
How often to recompute stress on supports.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

8.3.3 Collider



class `yade.wrapper.Collider`(*inherits GlobalEngine → Engine → Serializable*)
Abstract class for finding spatial collisions between bodies.

Special constructor

Derived colliders (unless they override `pyHandleCustomCtorArgs`) can be given list of *BoundFunctors* which is used to initialize the internal *boundDispatcher* instance.

`avoidSelfInteractionMask`
This mask is used to avoid the interactions inside a group of particles. To do so, the particles must have the same mask and this mask have to be compatible with this one.

`boundDispatcher`(=*new BoundDispatcher*)
BoundDispatcher object that is used for creating *bounds* on collider's request as necessary.

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.FlatGridCollider`(*inherits Collider → GlobalEngine → Engine → Serializable*)

Non-optimized grid collider, storing grid as dense flat array. Each body is assigned to (possibly multiple) cells, which are arranged in regular grid between *aabbMin* and *aabbMax*, with cell size *step* (same in all directions). Bodies outsize (*aabbMin*, *aabbMax*) are handled gracefully, assigned to closest cells (this will create spurious potential interactions). *verletDist* determines how much is each body enlarged to avoid collision detection at every step.

Note: This collider keeps all cells in linear memory array, therefore will be memory-inefficient for sparse simulations.

Warning: objects *Body::bound* are not used, *BoundFunctors* are not used either: assigning cells to bodies is hard-coded internally. Currently handles *Shapes* are: *Sphere*.

Note: Periodic boundary is not handled (yet).

`aabbMax`(=*Vector3r::Zero()*)
Upper corner of grid (approximate, might be rouded up to *minStep*.

`aabbMin`(=*Vector3r::Zero()*)
Lower corner of grid.

`avoidSelfInteractionMask`
This mask is used to avoid the interactions inside a group of particles. To do so, the particles must have the same mask and this mask have to be compatible with this one.

`boundDispatcher`(=*new BoundDispatcher*)
BoundDispatcher object that is used for creating *bounds* on collider's request as necessary.

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`step`(=*0*)
Step in the grid (cell size)

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`verletDist`(=*0*)
Length by which enlarge space occupied by each particle; avoids running collision detection at every step.

class `yade.wrapper.GeneralIntegratorInsertionSortCollider`(*inherits* *InsertionSortCollider* → *Collider* → *GlobalEngine* → *Engine* → *Serializable*)
This class is the adaptive version of the InsertionSortCollider and changes the NewtonIntegrator dependency of the collider algorithms to the Integrator interface which is more general.

`allowBiggerThanPeriod`
If true, tests on bodies sizes will be disabled, and the simulation will run normaly even if bodies larger than period are found. It can be useful when the periodic problem include e.g. a floor modelized with wall/box/facet. Be sure you know what you are doing if you touch this flag. The result is undefined if one large body moves out of the (0,0,0) period.

`avoidSelfInteractionMask`
This mask is used to avoid the interactions inside a group of particles. To do so, the particles must have the same mask and this mask have to be compatible with this one.

`boundDispatcher`(=*new BoundDispatcher*)
BoundDispatcher object that is used for creating *bounds* on collider's request as necessary.

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`doSort`(=*false*)
Do forced resorting of interactions.

`dumpBounds`() → tuple
Return representation of the internal sort data. The format is `([...],[...],[...])` for 3 axes, where each `...` is a list of entries (bounds). The entry is a tuple with the fllowing items:

- coordinate (float)
- body id (int), but negated for negative bounds

- period numer (int), if the collider is in the periodic regime.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

fastestBodyMaxDist(*=-1*)
Normalized maximum displacement of the fastest body since last run; if >= 1, we could get out of bboxes and will trigger full run. *(auto-updated)*

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

minSweepDistFactor(*=0.1*)
Minimal distance by which enlarge all bounding boxes; superseeds computed value of verletDist when lower that (minSweepDistFactor x verletDist).

numReinit(*=0*)
Cummulative number of bound array re-initialization.

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

periodic
Whether the collider is in periodic mode (read-only; for debugging) *(auto-updated)*

sortAxis(*=0*)
Axis for the initial contact detection.

sortThenCollide(*=false*)
Separate sorting and colliding phase; it is MUCH slower, but all interactions are processed at every step; this effectively makes the collider non-persistent, not remembering last state. (The default behavior relies on the fact that inversions during insertion sort are overlaps of bounding boxes that just started/ceased to exist, and only processes those; this makes the collider much more efficient.)

strideActive
Whether striding is active (read-only; for debugging). *(auto-updated)*

targetInterv(*=50*)
(experimental) Target number of iterations between bound update, used to define a smaller sweep distance for slower grains if >0, else always use 1*verletDist. Useful in simulations with strong velocity contrasts between slow bodies and fast bodies.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updatingDispFactor(*=-1*)
(experimental) Displacement factor used to trigger bound update: the bound is updated only if updatingDispFactor*disp>sweepDist when >0, else all bounds are updated.

useless(*=uninitalized*)
for compatibility of scripts defining the old collider's attributes - see deprecated attributes

verletDist(*=-.5*, *Automatically initialized*)
Length by which to enlarge particle bounds, to avoid running collider at every step. Stride

disabled if zero. Negative value will trigger automatic computation, so that the real value will be *verletDist* × minimum spherical particle radius; if there are no spherical particles, it will be disabled. The actual length added to one bound can be only a fraction of verletDist when *InsertionSortCollider::targetInterv* is > 0.

class `yade.wrapper.InsertionSortCollider`(*inherits* *Collider* → *GlobalEngine* → *Engine* → *Serializable*)

Collider with O(n log(n)) complexity, using *Aabb* for bounds.

At the initial step, Bodies' bounds (along *sortAxis*) are first std::sort'ed along this (sortAxis) axis, then collided. The initial sort has $O(n^2)$ complexity, see Colliders' performance for some information (There are scripts in examples/collider-perf for measurements).

Insertion sort is used for sorting the bound list that is already pre-sorted from last iteration, where each inversion calls checkOverlap which then handles either overlap (by creating interaction if necessary) or its absence (by deleting interaction if it is only potential).

Bodies without bounding volume (such as clumps) are handled gracefully and never collide. Deleted bodies are handled gracefully as well.

This collider handles periodic boundary conditions. There are some limitations, notably:

1. No body can have Aabb larger than cell's half size in that respective dimension. You get exception if it does and gets in interaction. One way to explicitly by-pass this restriction is offered by `allowBiggerThanPeriod`, which can be turned on to insert a floor in the form of a very large box for instance (see examples/periodicSandPile.py).
2. No body can travel more than cell's distance in one step; this would mean that the simulation is numerically exploding, and it is only detected in some cases.

Stride can be used to avoid running collider at every step by enlarging the particle's bounds, tracking their displacements and only re-run if they might have gone out of that bounds (see Verlet list for brief description and background) . This requires cooperation from *NewtonIntegrator* as well as *BoundDispatcher*, which will be found among engines automatically (exception is thrown if they are not found).

If you wish to use strides, set `verletDist` (length by which bounds will be enlarged in all directions) to some value, e.g. 0.05 × typical particle radius. This parameter expresses the tradeoff between many potential interactions (running collider rarely, but with longer exact interaction resolution phase) and few potential interactions (running collider more frequently, but with less exact resolutions of interactions); it depends mainly on packing density and particle radius distribution.

If `targetInterv` is >1, not all particles will have their bound enlarged by `verletDist`; instead, they will have bounds increased by a length in order to trigger a new colliding after `targetInterv` iteration, assuming they move at almost constant velocity. Ideally in this method, all particles would reach their bounds at the sime iteration. This is of course not the case as soon as velocities fluctuate in time. *Bound::sweepLength* is tuned on the basis of the displacement recorded between the last two runs of the collider. In this situation, `verletDist` defines the maximum sweep length.

`allowBiggerThanPeriod`
: If true, tests on bodies sizes will be disabled, and the simulation will run normaly even if bodies larger than period are found. It can be useful when the periodic problem include e.g. a floor modelized with wall/box/facet. Be sure you know what you are doing if you touch this flag. The result is undefined if one large body moves out of the (0,0,0) period.

`avoidSelfInteractionMask`
: This mask is used to avoid the interactions inside a group of particles. To do so, the particles must have the same mask and this mask have to be compatible with this one.

`boundDispatcher`(=*new BoundDispatcher*)
: *BoundDispatcher* object that is used for creating *bounds* on collider's request as necessary.

`dead`(=*false*)
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
Return dictionary of attributes.

`doSort(=false)`
Do forced resorting of interactions.

`dumpBounds()` → tuple
Return representation of the internal sort data. The format is `([...],[...],[...])` for 3 axes, where each `...` is a list of entries (bounds). The entry is a tuple with the fllowing items:

- coordinate (float)
- body id (int), but negated for negative bounds
- period numer (int), if the collider is in the periodic regime.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`fastestBodyMaxDist(=-1)`
Normalized maximum displacement of the fastest body since last run; if >= 1, we could get out of bboxes and will trigger full run. *(auto-updated)*

`label(=uninitalized)`
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`minSweepDistFactor(=0.1)`
Minimal distance by which enlarge all bounding boxes; superseeds computed value of verletDist when lower that (minSweepDistFactor x verletDist).

`numReinit(=0)`
Cummulative number of bound array re-initialization.

`ompThreads(=-1)`
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`periodic`
Whether the collider is in periodic mode (read-only; for debugging) *(auto-updated)*

`sortAxis(=0)`
Axis for the initial contact detection.

`sortThenCollide(=false)`
Separate sorting and colliding phase; it is MUCH slower, but all interactions are processed at every step; this effectively makes the collider non-persistent, not remembering last state. (The default behavior relies on the fact that inversions during insertion sort are overlaps of bounding boxes that just started/ceased to exist, and only processes those; this makes the collider much more efficient.)

`strideActive`
Whether striding is active (read-only; for debugging). *(auto-updated)*

`targetInterv(=50)`
(experimental) Target number of iterations between bound update, used to define a smaller sweep distance for slower grains if >0, else always use 1*verletDist. Useful in simulations with strong velocity contrasts between slow bodies and fast bodies.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updatingDispFactor(*=-1*)
(experimental) Displacement factor used to trigger bound update: the bound is updated only if updatingDispFactor*disp>sweepDist when >0, else all bounds are updated.

useless(*=uninitalized*)
for compatibility of scripts defining the old collider's attributes - see deprecated attributes

verletDist(*=-.5*, *Automatically initialized*)
Length by which to enlarge particle bounds, to avoid running collider at every step. Stride disabled if zero. Negative value will trigger automatic computation, so that the real value will be *verletDist* × minimum spherical particle radius; if there are no spherical particles, it will be disabled. The actual length added to one bound can be only a fraction of verletDist when *InsertionSortCollider::targetInterv* is > 0.

class yade.wrapper.**PersistentTriangulationCollider**(*inherits Collider → GlobalEngine → Engine → Serializable*)
Collision detection engine based on regular triangulation. Handles spheres and flat boundaries (considered as infinite-sized bounding spheres).

avoidSelfInteractionMask
This mask is used to avoid the interactions inside a group of particles. To do so, the particles must have the same mask and this mask have to be compatible with this one.

boundDispatcher(*=new BoundDispatcher*)
BoundDispatcher object that is used for creating *bounds* on collider's request as necessary.

dead(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

haveDistantTransient(*=false*)
Keep distant interactions? If True, don't delete interactions once bodies don't overlap anymore; constitutive laws will be responsible for requesting deletion. If False, delete as soon as there is no object penetration.

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.SpatialQuickSortCollider`(*inherits Collider → GlobalEngine → Engine → Serializable*)

Collider using quicksort along axes at each step, using *Aabb* bounds.

Its performance is lower than that of *InsertionSortCollider* (see Colliders' performance), but the algorithm is simple enought to make it good for checking other collider's correctness.

`avoidSelfInteractionMask`
: This mask is used to avoid the interactions inside a group of particles. To do so, the particles must have the same mask and this mask have to be compatible with this one.

`boundDispatcher`(=*new BoundDispatcher*)
: *BoundDispatcher* object that is used for creating *bounds* on collider's request as necessary.

`dead`(=*false*)
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
: Return dictionary of attributes.

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

class `yade.wrapper.ZECollider`(*inherits Collider → GlobalEngine → Engine → Serializable*)

Collider with O(n log(n)) complexity, using a CGAL algorithm from Zomorodian and Edelsbrunner *[Kettner2011]* (http://www.cgal.org/Manual/beta/doc_html/cgal_manual/Box_intersection_d/Chapter_main.html)

`avoidSelfInteractionMask`
: This mask is used to avoid the interactions inside a group of particles. To do so, the particles must have the same mask and this mask have to be compatible with this one.

`boundDispatcher`(=*new BoundDispatcher*)
: *BoundDispatcher* object that is used for creating *bounds* on collider's request as necessary.

`dead`(=*false*)
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
: Return dictionary of attributes.

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

fastestBodyMaxDist(*=-1*)
Maximum displacement of the fastest body since last run; if >= verletDist, we could get out of bboxes and will trigger full run. DEPRECATED, was only used without bins. *(auto-updated)*

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

numReinit(*=0*)
Cummulative number of bound array re-initialization.

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

periodic
Whether the collider is in periodic mode (read-only; for debugging) *(auto-updated)*

sortAxis(*=0*)
Axis for the initial contact detection.

sortThenCollide(*=false*)
Separate sorting and colliding phase; it is MUCH slower, but all interactions are processed at every step; this effectively makes the collider non-persistent, not remembering last state. (The default behavior relies on the fact that inversions during insertion sort are overlaps of bounding boxes that just started/ceased to exist, and only processes those; this makes the collider much more efficient.)

strideActive
Whether striding is active (read-only; for debugging). *(auto-updated)*

targetInterv(*=30*)
(experimental) Target number of iterations between bound update, used to define a smaller sweep distance for slower grains if >0, else always use 1*verletDist. Useful in simulations with strong velocity contrasts between slow bodies and fast bodies.

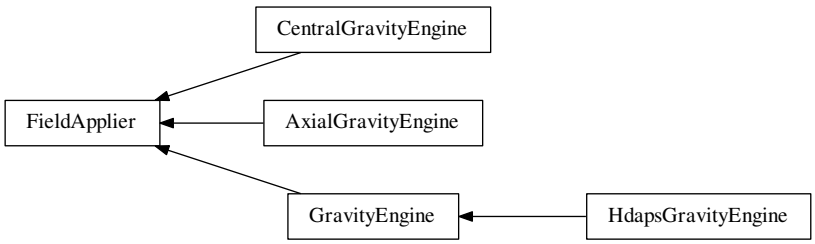
timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updatingDispFactor(*=-1*)
(experimental) Displacement factor used to trigger bound update: the bound is updated only if updatingDispFactor*disp>sweepDist when >0, else all bounds are updated.

verletDist(*=-.15*, *Automatically initialized*)
Length by which to enlarge particle bounds, to avoid running collider at every step. Stride disabled if zero. Negative value will trigger automatic computation, so that the real value will be *verletDist* × minimum spherical particle radius; if there are no spherical particles, it will be disabled.

8.3.4 FieldApplier



class `yade.wrapper.FieldApplier`(*inherits* *GlobalEngine* → *Engine* → *Serializable*)

Base for engines applying force files on particles. Not to be used directly.

`dead`(=*false*)
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
: Return dictionary of attributes.

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

class `yade.wrapper.AxialGravityEngine`(*inherits* *FieldApplier* → *GlobalEngine* → *Engine* → *Serializable*)

Apply acceleration (independent of distance) directed towards an axis.

`acceleration`(=*0*)
: Acceleration magnitude [kgms ^{2}]

`axisDirection`(=*Vector3r::UnitX()*)
: direction of the gravity axis (will be normalized automatically)

`axisPoint`(=*Vector3r::Zero()*)
: Point through which the axis is passing.

`dead`(=*false*)
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`mask`(=*0*)
If mask defined, only bodies with corresponding groupMask will be affected by this engine. If 0, all bodies will be affected.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.CentralGravityEngine`(*inherits FieldApplier → GlobalEngine → Engine → Serializable*)
Engine applying acceleration to all bodies, towards a central body.

`accel`(=*0*)
Acceleration magnitude [kgms ^{2}]

`centralBody`(=*Body::ID_NONE*)
The *body* towards which all other bodies are attracted.

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`mask`(=*0*)
If mask defined, only bodies with corresponding groupMask will be affected by this engine. If 0, all bodies will be affected.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes

openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

reciprocal(=*false*)
If true, acceleration will be applied on the central body as well.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.GravityEngine`(*inherits* *FieldApplier* → *GlobalEngine* → *Engine* → *Serializable*)
Engine applying constant acceleration to all bodies. DEPRECATED, use *Newton::gravity* unless you need energy tracking or selective gravity application using groupMask).

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

gravity(=*Vector3r::Zero()*)
Acceleration [kgms ^{2}]

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

mask(=*0*)
If mask defined, only bodies with corresponding groupMask will be affected by this engine. If 0, all bodies will be affected.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

warnOnce(=*true*)
For deprecation warning once.

class `yade.wrapper.HdapsGravityEngine`(*inherits* *GravityEngine* → *FieldApplier* → *GlobalEngine* → *Engine* → *Serializable*)
Read accelerometer in Thinkpad laptops (HDAPS and accordingly set gravity within the simulation. This code draws from hdaps-gl . See scripts/test/hdaps.py for an example.

accel(=*Vector2i::Zero()*)
reading from the sysfs file

calibrate(=*Vector2i::Zero()*)
Zero position; if NaN, will be read from the *hdapsDir* / calibrate.

calibrated(=*false*)
Whether *calibrate* was already updated. Do not set to `True` by hand unless you also give a meaningful value for *calibrate*.

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

gravity(=*Vector3r::Zero()*)
Acceleration [kgms ^{2}]

hdapsDir(=*"/sys/devices/platform/hdaps"*)
Hdaps directory; contains `position` (with accelerometer readings) and `calibration` (zero acceleration).

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

mask(=*0*)
If mask defined, only bodies with corresponding groupMask will be affected by this engine. If 0, all bodies will be affected.

msecUpdate(=*50*)
How often to update the reading.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

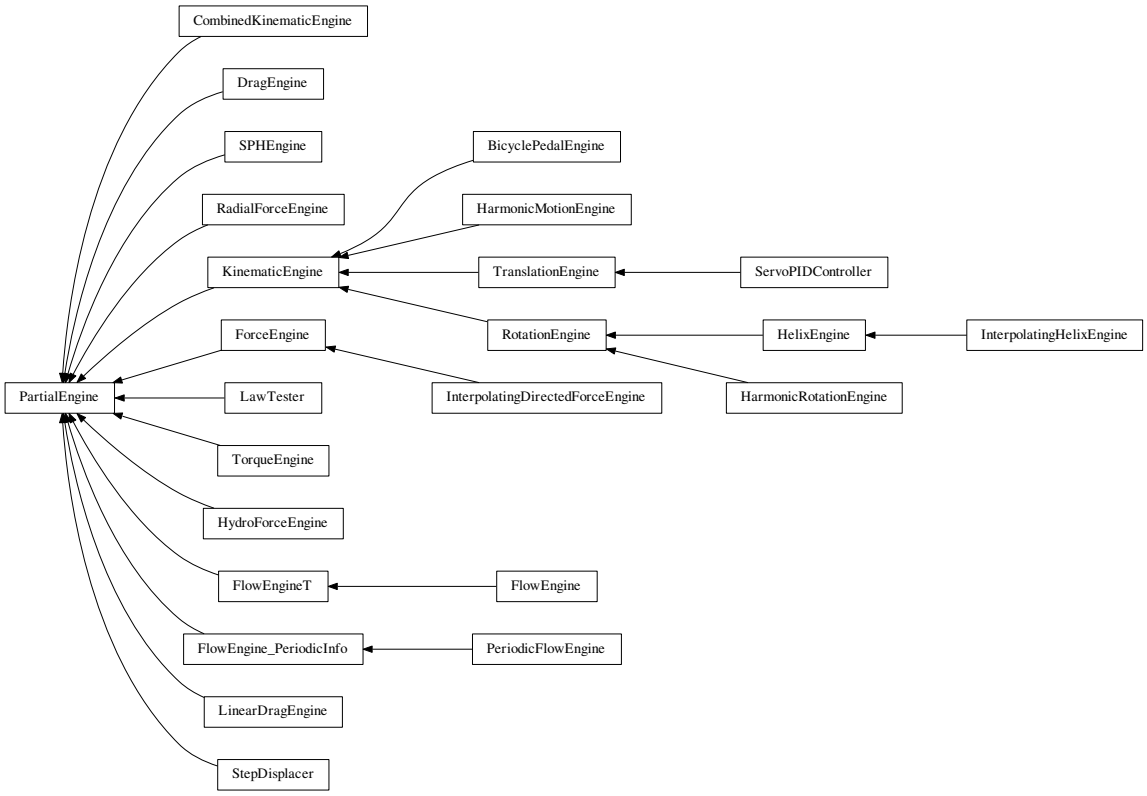
updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updateThreshold(=*4*)
Minimum difference of reading from the file before updating gravity, to avoid jitter.

warnOnce(=*true*)
For deprecation warning once.

zeroGravity(=*Vector3r(0, 0, -1)*)
Gravity if the accelerometer is in flat (zero) position.

8.4 Partial engines



class `yade.wrapper.PartialEngine`(*inherits Engine → Serializable*)

Engine affecting only particular bodies in the simulation, defined by *ids*.

`dead`(=*false*)
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
: Return dictionary of attributes.

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`ids`(=*uninitalized*)
: *Ids* of bodies affected by this PartialEngine.

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source

code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.BicyclePedalEngine`(*inherits KinematicEngine → PartialEngine → Engine → Serializable*)
Engine applying the linear motion of `bicycle pedal` e.g. moving points around the axis without rotation

`angularVelocity`(*=0*)
Angular velocity. [rad/s]

`dead`(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`fi`(*=Mathr::PI/2.0*)
Initial phase [radians]

`ids`(*=uninitalized*)
Ids of bodies affected by this PartialEngine.

`label`(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`radius`(*=-1.0*)
Rotation radius. [m]

`rotationAxis`(*=Vector3r::UnitX()*)
Axis of rotation (direction); will be normalized automatically.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.CombinedKinematicEngine`(*inherits PartialEngine → Engine → Serializable*)
Engine for applying combined displacements on pre-defined bodies. Constructed using `+` operator on regular *KinematicEngines*. The `ids` operated on are those of the first engine in the combination (assigned automatically).

`comb`(*=uninitalized*)
Kinematic engines that will be combined by this one, run in the order given.

`dead`(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`ids`(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.DragEngine`(*inherits PartialEngine → Engine → Serializable*)
Apply drag force on some particles at each step, decelerating them proportionally to their linear velocities. The applied force reads

$$F_d = -\frac{\nu}{|\nu|}\frac{1}{2}\rho|\nu|^2 C_d A$$

where ρ is the medium density (*density*), ν is particle's velocity, A is particle projected area (disc), C_d is the drag coefficient (0.47 for *Sphere*),

Note: Drag force is only applied to spherical particles, listed in ids.

`Cd`(=*0.47*)
Drag coefficient <http://en.wikipedia.org/wiki/Drag_coefficient>`_.

`Rho`(=*1.225*)
Density of the medium (fluid or air), by default - the density of the air.

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`ids`(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**FlowEngine**(*inherits FlowEngineT → PartialEngine → Engine → Serializable*)
An engine to solve flow problem in saturated granular media. Model description can be found in *[Chareyre2012a]* and *[Catalano2014a]*. See the example script FluidCouplingPFV/oedometer.py. More documentation to come.

OSI() → float
Return the number of interactions only between spheres.

avFlVelOnSph(*(int)idSph*) → object
compute a sphere-centered average fluid velocity

averagePressure() → float
Measure averaged pore pressure in the entire volume

averageSlicePressure(*(float)posY*) → float
Measure slice-averaged pore pressure at height posY

averageVelocity() → Vector3
measure the mean velocity in the period

blockCell(*(int)id*, *(bool)blockPressure*) → None
block cell 'id'. The cell will be excluded from the fluid flow problem and the conductivity of all incident facets will be null. If blockPressure=False, deformation is reflected in the pressure, else it is constantly 0.

blockHook(=*""*)
Python command to be run when remeshing. Anticipated usage: define blocked cells (see also *TemplateFlowEngine_FlowEngineT.blockCell*), or apply exotic types of boundary conditions which need to visit the newly built mesh

bndCondIsPressure(=*vector<bool>(6, false)*)
defines the type of boundary condition for each side. True if pressure is imposed, False for no-flux. Indexes can be retrieved with *FlowEngine::xmin* and friends.

bndCondValue(=*vector<double>(6, 0)*)
Imposed value of a boundary condition. Only applies if the boundary condition is imposed pressure, else the imposed flux is always zero presently (may be generalized to non-zero imposed fluxes in the future).

bodyNormalLubStress(*(int)idSph*) → Matrix3
Return the normal lubrication stress on sphere idSph.

bodyShearLubStress(*(int)idSph*) → Matrix3
Return the shear lubrication stress on sphere idSph.

boundaryPressure(=*vector<Real>()*)
values defining pressure along x-axis for the top surface. See also *FlowEngineT::boundaryXPos*

boundaryUseMaxMin(=*vector<bool>(6, true)*)
If true (default value) bounding sphere is added as function of max/min sphere coord, if false as function of yade wall position

boundaryVelocity(=*vector<Vector3r>(6, Vector3r::Zero())*)
velocity on top boundary, only change it using *FlowEngine::setBoundaryVel*

boundaryXPos(=*vector<Real>()*)
values of the x-coordinate for which pressure is defined. See also *FlowEngineT::boundaryPressure*

cholmodStats() → None
get statistics of cholmod solver activity

clampKValues(=*true*)
If true, clamp local permeabilities in [minKdivKmean,maxKdivKmean]*globalK. This clamping can avoid singular values in the permeability matrix and may reduce numerical errors in the solve phase. It will also hide junk values if they exist, or bias all values in very heterogeneous problems. So, use this with care.

clearImposedFlux() → None
Clear the list of points with flux imposed.

clearImposedPressure() → None
Clear the list of points with pressure imposed.

compTessVolumes() → None
Like TesselationWrapper::computeVolumes()

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

debug(=*false*)
Activate debug messages

defTolerance(=*0.05*)
Cumulated deformation threshold for which retriangulation of pore space is performed. If negative, the triangulation update will occure with a fixed frequency on the basis of *FlowEngine::meshUpdateInterval*

dict() → dict
Return dictionary of attributes.

doInterpolate(=*false*)
Force the interpolation of cell's info while remeshing. By default, interpolation would be done only for compressible fluids. It can be forced with this flag.

dt(=*0*)
timestep [s]

edgeSize() → float
Return the number of interactions.

emulateAction() → None
get scene and run action (may be used to manipulate an engine outside the timestepping loop).

eps(=*0.00001*)
roughness defined as a fraction of particles size, giving the minimum distance between particles in the lubrication model.

epsVolMax(=*0*)
Maximal absolute volumetric strain computed at each iteration. *(auto-updated)*

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

exportMatrix(`[`*(str)filename='matrix'*`]`) → None
Export system matrix to a file with all entries (even zeros will displayed).

`exportTriplets`(*[(str)filename='triplets']*) → None
Export system matrix to a file with only non-zero entries.

`first`(*=true*)
Controls the initialization/update phases

`fluidBulkModulus`(*=0.*)
Bulk modulus of fluid (inverse of compressibility) K=-dP*V/dV [Pa]. Flow is compressible if fluidBulkModulus > 0, else incompressible.

`fluidForce`(*(int)idSph*) → Vector3
Return the fluid force on sphere idSph.

`forceMetis`
If true, METIS is used for matrix preconditioning, else Cholmod is free to choose the best method (which may be METIS to, depending on the matrix). See `nmethods` in Cholmod documentation

`getBoundaryFlux`(*(int)boundary*) → float
Get total flux through boundary defined by its body id.

Note: The flux may be not zero even for no-flow condition. This artifact comes from cells which are incident to two or more boundaries (along the edges of the sample, typically). Such flux evaluation on impermeable boundary is just irrelevant, it does not imply that the boundary condition is not applied properly.

`getCell`(*(float)arg2*, *(float)arg3*, *(float)pos*) → int
get id of the cell containing (X,Y,Z).

`getCellBarycenter`(*(int)id*) → Vector3
get barycenter of cell 'id'.

`getCellCenter`(*(int)id*) → Vector3
get voronoi center of cell 'id'.

`getCellFlux`(*(int)cond*) → float
Get influx in cell associated to an imposed P (indexed using 'cond').

`getCellPImposed`(*(int)id*) → bool
get the status of cell 'id' wrt imposed pressure.

`getCellPressure`(*(int)id*) → float
get pressure in cell 'id'.

`getConstrictions`(*[(bool)all=True]*) → list
Get the list of constriction radii (inscribed circle) for all finite facets (if all==True) or all facets not incident to a virtual bounding sphere (if all==False). When all facets are returned, negative radii denote facet incident to one or more fictious spheres.

`getConstrictionsFull`(*[(bool)all=True]*) → list
Get the list of constrictions (inscribed circle) for all finite facets (if all==True), or all facets not incident to a fictious bounding sphere (if all==False). When all facets are returned, negative radii denote facet incident to one or more fictious spheres. The constrictions are returned in the format {{cell1,cell2}{rad,nx,ny,nz}}

`getPorePressure`(*(Vector3)pos*) → float
Measure pore pressure in position pos[0],pos[1],pos[2]

`getVertices`(*(int)id*) → list
get the vertices of a cell

`ids`(*=uninitalized*)
Ids of bodies affected by this PartialEngine.

`ignoredBody`(*=-1*)
Id of a sphere to exclude from the triangulation.)

imposeFlux(*(Vector3)pos*, *(float)p*) → None
Impose a flux in cell located at 'pos' (i.e. add a source term in the flow problem). Outflux positive, influx negative.

imposePressure(*(Vector3)pos*, *(float)p*) → int
Impose pressure in cell of location 'pos'. The index of the condition is returned (for multiple imposed pressures at different points).

imposePressureFromId(*(int)id*, *(float)p*) → int
Impose pressure in cell of index 'id' (after remeshing the same condition will apply for the same location, regardless of what the new cell index is at this location). The index of the condition itself is returned (for multiple imposed pressures at different points).

isActivated(=*true*)
Activates Flow Engine

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxKdivKmean(=*100*)
define the max K value (see *FlowEngine::clampKValues*)

meanKStat(=*false*)
report the local permeabilities' correction

meshUpdateInterval(=*1000*)
Maximum number of timesteps between re-triangulation events. See also *FlowEngine::defTolerance*.

metisUsed() → bool
check wether metis lib is effectively used

minKdivKmean(=*0.0001*)
define the min K value (see *FlowEngine::clampKValues*)

multithread(=*false*)
Build triangulation and factorize in the background (multi-thread mode)

nCells() → int
get the total number of finite cells in the triangulation.

normalLubForce(*(int)idSph*) → Vector3
Return the normal lubrication force on sphere idSph.

normalLubrication(=*false*)
compute normal lubrication force as developped by Brule

normalVect(*(int)idSph*) → Vector3
Return the normal vector between particles.

normalVelocity(*(int)idSph*) → Vector3
Return the normal velocity of the interaction.

numFactorizeThreads(=*1*)
number of openblas threads in the factorization phase

numSolveThreads(=*1*)
number of openblas threads in the solve phase.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

onlySpheresInteractions(*(int)interaction*) → int
Return the id of the interaction only between spheres.

pZero(*=0*)
The value used for initializing pore pressure. It is useless for incompressible fluid, but important for compressible model.

permeabilityFactor(*=1.0*)
permability multiplier

permeabilityMap(*=false*)
Enable/disable stocking of average permeability scalar in cell infos.

porosity(*=0*)
Porosity computed at each retriangulation *(auto-updated)*

pressureForce(*=true*)
compute the pressure field and associated fluid forces. WARNING: turning off means fluid flow is not computed at all.

pressureProfile(*(float)wallUpY*, *(float)wallDownY*) → None
Measure pore pressure in 6 equally-spaced points along the height of the sample

pumpTorque(*=false*)
Compute pump torque applied on particles

relax(*=1.9*)
Gauss-Seidel relaxation

saveVtk(*[(str)folder='./VTK']*) → None
Save pressure field in vtk format. Specify a folder name for output.

setCellPImposed(*(int)id*, *(bool)pImposed*) → None
make cell 'id' assignable with imposed pressure.

setCellPressure(*(int)id*, *(float)pressure*) → None
set pressure in cell 'id'.

setImposedPressure(*(int)cond*, *(float)p*) → None
Set pressure value at the point indexed 'cond'.

shearLubForce(*(int)idSph*) → Vector3
Return the shear lubrication force on sphere idSph.

shearLubTorque(*(int)idSph*) → Vector3
Return the shear lubrication torque on sphere idSph.

shearLubrication(*=false*)
compute shear lubrication force as developped by Brule (FIXME: ref.)

shearVelocity(*(int)idSph*) → Vector3
Return the shear velocity of the interaction.

sineAverage(*=0*)
Pressure value (average) when sinusoidal pressure is applied

sineMagnitude(*=0*)
Pressure value (amplitude) when sinusoidal pressure is applied (p)

slipBoundary(*=true*)
Controls friction condition on lateral walls

stiffness(*=10000*)
equivalent contact stiffness used in the lubrication model

surfaceDistanceParticle(*(int)interaction*) → float
Return the distance between particles.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

tolerance(=*1e-06*)
Gauss-Seidel tolerance

twistTorque(=*false*)
Compute twist torque applied on particles

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updateBCs() → None
tells the engine to update it's boundary conditions before running (especially useful when changing boundary pressure - should not be needed for point-wise imposed pressure)

updateTriangulation(=*0*)
If true the medium is retriangulated. Can be switched on to force retriangulation after some events (else it will be true periodicaly based on *FlowEngine::defTolerance* and *FlowEngine::meshUpdateInterval*. Of course, it costs CPU time.

useSolver(=*0*)
Solver to use 0=G-Seidel, 1=Taucs, 2-Pardiso, 3-CHOLMOD

viscosity(=*1.0*)
viscosity of the fluid

viscousNormalBodyStress(=*false*)
compute normal viscous stress applied on each body

viscousShear(=*false*)
compute viscous shear terms as developped by Donia Marzougui (FIXME: ref.)

viscousShearBodyStress(=*false*)
compute shear viscous stress applied on each body

volume(*(int)id=0*) → float
Returns the volume of Voronoi's cell of a sphere.

wallIds(=*vector<int>(6)*)
body ids of the boundaries (default values are ok only if aabbWalls are appended before spheres, i.e. numbered 0,...,5)

wallThickness(=*0*)
Walls thickness

waveAction(=*false*)
Allow sinusoidal pressure condition to simulate ocean waves

xmax(=*1*)
See *FlowEngine::xmin*.

xmin(=*0*)
Index of the boundary $\mathtt{x}_{\mathtt{min}}$. This index is not equal the the id of the corresponding body in general, it may be used to access the corresponding attributes (e.g. flow.bndCondValue[flow.xmin], flow.wallId[flow.xmin],...).

ymax(=*3*)
See *FlowEngine::xmin*.

ymin(=*2*)
See *FlowEngine::xmin*.

zmax(=*5*)
See *FlowEngine::xmin*.

zmin(=*4*)
See *FlowEngine::xmin*.

class yade.wrapper.**FlowEngineT**(*inherits PartialEngine → Engine → Serializable*)
A generic engine from wich more specialized engines can inherit. It is defined for the sole purpose of inserting the right data classes CellInfo and VertexInfo in the triangulation, and it should not

be used directly. Instead, look for specialized engines, e.g. *FlowEngine*, *PeriodicFlowEngine*, or *DFNFlowEngine*.

OSI() → float
Return the number of interactions only between spheres.

avFlVelOnSph(*(int)idSph*) → object
compute a sphere-centered average fluid velocity

averagePressure() → float
Measure averaged pore pressure in the entire volume

averageSlicePressure(*(float)posY*) → float
Measure slice-averaged pore pressure at height posY

averageVelocity() → Vector3
measure the mean velocity in the period

blockCell(*(int)id*, *(bool)blockPressure*) → None
block cell 'id'. The cell will be excluded from the fluid flow problem and the conductivity of all incident facets will be null. If blockPressure=False, deformation is reflected in the pressure, else it is constantly 0.

blockHook(=*""*)
Python command to be run when remeshing. Anticipated usage: define blocked cells (see also *TemplateFlowEngine_FlowEngineT.blockCell*), or apply exotic types of boundary conditions which need to visit the newly built mesh

bndCondIsPressure(=*vector<bool>(6, false)*)
defines the type of boundary condition for each side. True if pressure is imposed, False for no-flux. Indexes can be retrieved with *FlowEngine::xmin* and friends.

bndCondValue(=*vector<double>(6, 0)*)
Imposed value of a boundary condition. Only applies if the boundary condition is imposed pressure, else the imposed flux is always zero presently (may be generalized to non-zero imposed fluxes in the future).

bodyNormalLubStress(*(int)idSph*) → Matrix3
Return the normal lubrication stress on sphere idSph.

bodyShearLubStress(*(int)idSph*) → Matrix3
Return the shear lubrication stress on sphere idSph.

boundaryPressure(=*vector<Real>()*)
values defining pressure along x-axis for the top surface. See also *FlowEngineT::boundaryXPos*

boundaryUseMaxMin(=*vector<bool>(6, true)*)
If true (default value) bounding sphere is added as function of max/min sphere coord, if false as function of yade wall position

boundaryVelocity(=*vector<Vector3r>(6, Vector3r::Zero())*)
velocity on top boundary, only change it using *FlowEngine::setBoundaryVel*

boundaryXPos(=*vector<Real>()*)
values of the x-coordinate for which pressure is defined. See also *FlowEngineT::boundaryPressure*

cholmodStats() → None
get statistics of cholmod solver activity

clampKValues(=*true*)
If true, clamp local permeabilities in [minKdivKmean,maxKdivKmean]*globalK. This clamping can avoid singular values in the permeability matrix and may reduce numerical errors in the solve phase. It will also hide junk values if they exist, or bias all values in very heterogeneous problems. So, use this with care.

clearImposedFlux() → None
Clear the list of points with flux imposed.

`clearImposedPressure()` → None
Clear the list of points with pressure imposed.

`compTessVolumes()` → None
Like TesselationWrapper::computeVolumes()

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`debug`(=*false*)
Activate debug messages

`defTolerance`(=*0.05*)
Cumulated deformation threshold for which retriangulation of pore space is performed. If negative, the triangulation update will occure with a fixed frequency on the basis of *FlowEngine::meshUpdateInterval*

`dict()` → dict
Return dictionary of attributes.

`doInterpolate`(=*false*)
Force the interpolation of cell's info while remeshing. By default, interpolation would be done only for compressible fluids. It can be forced with this flag.

`dt`(=*0*)
timestep [s]

`edgeSize()` → float
Return the number of interactions.

`emulateAction()` → None
get scene and run action (may be used to manipulate an engine outside the timestepping loop).

`eps`(=*0.00001*)
roughness defined as a fraction of particles size, giving the minimum distance between particles in the lubrication model.

`epsVolMax`(=*0*)
Maximal absolute volumetric strain computed at each iteration. *(auto-updated)*

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`exportMatrix`(`[`*(str)filename='matrix'*`]`) → None
Export system matrix to a file with all entries (even zeros will displayed).

`exportTriplets`(`[`*(str)filename='triplets'*`]`) → None
Export system matrix to a file with only non-zero entries.

`first`(=*true*)
Controls the initialization/update phases

`fluidBulkModulus`(=*0.*)
Bulk modulus of fluid (inverse of compressibility) K=-dP*V/dV [Pa]. Flow is compressible if fluidBulkModulus > 0, else incompressible.

`fluidForce`(*(int)idSph*) → Vector3
Return the fluid force on sphere idSph.

`forceMetis`
If true, METIS is used for matrix preconditioning, else Cholmod is free to choose the best method (which may be METIS to, depending on the matrix). See `nmethods` in Cholmod documentation

`getBoundaryFlux(`*(int)boundary*`)` → float
Get total flux through boundary defined by its body id.

Note: The flux may be not zero even for no-flow condition. This artifact comes from cells which are incident to two or more boundaries (along the edges of the sample, typically). Such flux evaluation on impermeable boundary is just irrelevant, it does not imply that the boundary condition is not applied properly.

`getCell(`*(float)arg2*, *(float)arg3*, *(float)pos*`)` → int
get id of the cell containing (X,Y,Z).

`getCellBarycenter(`*(int)id*`)` → Vector3
get barycenter of cell 'id'.

`getCellCenter(`*(int)id*`)` → Vector3
get voronoi center of cell 'id'.

`getCellFlux(`*(int)cond*`)` → float
Get influx in cell associated to an imposed P (indexed using 'cond').

`getCellPImposed(`*(int)id*`)` → bool
get the status of cell 'id' wrt imposed pressure.

`getCellPressure(`*(int)id*`)` → float
get pressure in cell 'id'.

`getConstrictions(`[*(bool)all=True*]`)` → list
Get the list of constriction radii (inscribed circle) for all finite facets (if all==True) or all facets not incident to a virtual bounding sphere (if all==False). When all facets are returned, negative radii denote facet incident to one or more fictious spheres.

`getConstrictionsFull(`[*(bool)all=True*]`)` → list
Get the list of constrictions (inscribed circle) for all finite facets (if all==True), or all facets not incident to a fictious bounding sphere (if all==False). When all facets are returned, negative radii denote facet incident to one or more fictious spheres. The constrictions are returned in the format {{cell1,cell2}{rad,nx,ny,nz}}

`getPorePressure(`*(Vector3)pos*`)` → float
Measure pore pressure in position pos[0],pos[1],pos[2]

`getVertices(`*(int)id*`)` → list
get the vertices of a cell

`ids(`=*uninitalized*`)`
Ids of bodies affected by this PartialEngine.

`ignoredBody(`=*-1*`)`
Id of a sphere to exclude from the triangulation.)

`imposeFlux(`*(Vector3)pos*, *(float)p*`)` → None
Impose a flux in cell located at 'pos' (i.e. add a source term in the flow problem). Outflux positive, influx negative.

`imposePressure(`*(Vector3)pos*, *(float)p*`)` → int
Impose pressure in cell of location 'pos'. The index of the condition is returned (for multiple imposed pressures at different points).

`imposePressureFromId(`*(int)id*, *(float)p*`)` → int
Impose pressure in cell of index 'id' (after remeshing the same condition will apply for the same location, regardless of what the new cell index is at this location). The index of the condition itself is returned (for multiple imposed pressures at different points).

`isActivated(`=*true*`)`
Activates Flow Engine

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxKdivKmean(=*100*)
define the max K value (see *FlowEngine::clampKValues*)

meanKStat(=*false*)
report the local permeabilities' correction

meshUpdateInterval(=*1000*)
Maximum number of timesteps between re-triangulation events. See also *FlowEngine::defTolerance*.

metisUsed() → bool
check wether metis lib is effectively used

minKdivKmean(=*0.0001*)
define the min K value (see *FlowEngine::clampKValues*)

multithread(=*false*)
Build triangulation and factorize in the background (multi-thread mode)

nCells() → int
get the total number of finite cells in the triangulation.

normalLubForce(*(int)idSph*) → Vector3
Return the normal lubrication force on sphere idSph.

normalLubrication(=*false*)
compute normal lubrication force as developped by Brule

normalVect(*(int)idSph*) → Vector3
Return the normal vector between particles.

normalVelocity(*(int)idSph*) → Vector3
Return the normal velocity of the interaction.

numFactorizeThreads(=*1*)
number of openblas threads in the factorization phase

numSolveThreads(=*1*)
number of openblas threads in the solve phase.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

onlySpheresInteractions(*(int)interaction*) → int
Return the id of the interaction only between spheres.

pZero(=*0*)
The value used for initializing pore pressure. It is useless for incompressible fluid, but important for compressible model.

permeabilityFactor(=*1.0*)
permability multiplier

permeabilityMap(=*false*)
Enable/disable stocking of average permeability scalar in cell infos.

porosity(=*0*)
Porosity computed at each retriangulation *(auto-updated)*

pressureForce(*=true*)
compute the pressure field and associated fluid forces. WARNING: turning off means fluid flow is not computed at all.

pressureProfile(*(float)wallUpY*, *(float)wallDownY*) → None
Measure pore pressure in 6 equally-spaced points along the height of the sample

pumpTorque(*=false*)
Compute pump torque applied on particles

relax(*=1.9*)
Gauss-Seidel relaxation

saveVtk([*(str)folder='./VTK'*]) → None
Save pressure field in vtk format. Specify a folder name for output.

setCellPImposed(*(int)id*, *(bool)pImposed*) → None
make cell 'id' assignable with imposed pressure.

setCellPressure(*(int)id*, *(float)pressure*) → None
set pressure in cell 'id'.

setImposedPressure(*(int)cond*, *(float)p*) → None
Set pressure value at the point indexed 'cond'.

shearLubForce(*(int)idSph*) → Vector3
Return the shear lubrication force on sphere idSph.

shearLubTorque(*(int)idSph*) → Vector3
Return the shear lubrication torque on sphere idSph.

shearLubrication(*=false*)
compute shear lubrication force as developped by Brule (FIXME: ref.)

shearVelocity(*(int)idSph*) → Vector3
Return the shear velocity of the interaction.

sineAverage(*=0*)
Pressure value (average) when sinusoidal pressure is applied

sineMagnitude(*=0*)
Pressure value (amplitude) when sinusoidal pressure is applied (p)

slipBoundary(*=true*)
Controls friction condition on lateral walls

stiffness(*=10000*)
equivalent contact stiffness used in the lubrication model

surfaceDistanceParticle(*(int)interaction*) → float
Return the distance between particles.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

tolerance(*=1e-06*)
Gauss-Seidel tolerance

twistTorque(*=false*)
Compute twist torque applied on particles

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updateBCs() → None
tells the engine to update it's boundary conditions before running (especially useful when changing boundary pressure - should not be needed for point-wise imposed pressure)

updateTriangulation(=*0*)
If true the medium is retriangulated. Can be switched on to force retriangulation after some events (else it will be true periodicaly based on *FlowEngine::defTolerance* and *FlowEngine::meshUpdateInterval*. Of course, it costs CPU time.

useSolver(=*0*)
Solver to use 0=G-Seidel, 1=Taucs, 2-Pardiso, 3-CHOLMOD

viscosity(=*1.0*)
viscosity of the fluid

viscousNormalBodyStress(=*false*)
compute normal viscous stress applied on each body

viscousShear(=*false*)
compute viscous shear terms as developped by Donia Marzougui (FIXME: ref.)

viscousShearBodyStress(=*false*)
compute shear viscous stress applied on each body

volume(*(int)id=0*) → float
Returns the volume of Voronoi's cell of a sphere.

wallIds(=*vector<int>(6)*)
body ids of the boundaries (default values are ok only if aabbWalls are appended before spheres, i.e. numbered 0,...,5)

wallThickness(=*0*)
Walls thickness

waveAction(=*false*)
Allow sinusoidal pressure condition to simulate ocean waves

xmax(=*1*)
See *FlowEngine::xmin*.

xmin(=*0*)
Index of the boundary x_{min}. This index is not equal the the id of the corresponding body in general, it may be used to access the corresponding attributes (e.g. flow.bndCondValue[flow.xmin], flow.wallId[flow.xmin],...).

ymax(=*3*)
See *FlowEngine::xmin*.

ymin(=*2*)
See *FlowEngine::xmin*.

zmax(=*5*)
See *FlowEngine::xmin*.

zmin(=*4*)
See *FlowEngine::xmin*.

class yade.wrapper.**FlowEngine_PeriodicInfo**(*inherits PartialEngine → Engine → Serializable*)
A generic engine from wich more specialized engines can inherit. It is defined for the sole purpose of inserting the right data classes CellInfo and VertexInfo in the triangulation, and it should not be used directly. Instead, look for specialized engines, e.g. *FlowEngine*, *PeriodicFlowEngine*, or *DFNFlowEngine*.

OSI() → float
Return the number of interactions only between spheres.

avFlVelOnSph(*(int)idSph*) → object
compute a sphere-centered average fluid velocity

averagePressure() → float
Measure averaged pore pressure in the entire volume

averageSlicePressure(*(float)posY*) → float
Measure slice-averaged pore pressure at height posY

averageVelocity() → Vector3
measure the mean velocity in the period

blockCell(*(int)id*, *(bool)blockPressure*) → None
block cell 'id'. The cell will be excluded from the fluid flow problem and the conductivity of all incident facets will be null. If blockPressure=False, deformation is reflected in the pressure, else it is constantly 0.

blockHook(=*""*)
Python command to be run when remeshing. Anticipated usage: define blocked cells (see also *TemplateFlowEngine_FlowEngine_PeriodicInfo.blockCell*), or apply exotic types of boundary conditions which need to visit the newly built mesh

bndCondIsPressure(=*vector<bool>(6, false)*)
defines the type of boundary condition for each side. True if pressure is imposed, False for no-flux. Indexes can be retrieved with *FlowEngine::xmin* and friends.

bndCondValue(=*vector<double>(6, 0)*)
Imposed value of a boundary condition. Only applies if the boundary condition is imposed pressure, else the imposed flux is always zero presently (may be generalized to non-zero imposed fluxes in the future).

bodyNormalLubStress(*(int)idSph*) → Matrix3
Return the normal lubrication stress on sphere idSph.

bodyShearLubStress(*(int)idSph*) → Matrix3
Return the shear lubrication stress on sphere idSph.

boundaryPressure(=*vector<Real>()*)
values defining pressure along x-axis for the top surface. See also *FlowEngine_PeriodicInfo::boundaryXPos*

boundaryUseMaxMin(=*vector<bool>(6, true)*)
If true (default value) bounding sphere is added as function of max/min sphere coord, if false as function of yade wall position

boundaryVelocity(=*vector<Vector3r>(6, Vector3r::Zero())*)
velocity on top boundary, only change it using *FlowEngine::setBoundaryVel*

boundaryXPos(=*vector<Real>()*)
values of the x-coordinate for which pressure is defined. See also *FlowEngine_PeriodicInfo::boundaryPressure*

cholmodStats() → None
get statistics of cholmod solver activity

clampKValues(=*true*)
If true, clamp local permeabilities in [minKdivKmean,maxKdivKmean]*globalK. This clamping can avoid singular values in the permeability matrix and may reduce numerical errors in the solve phase. It will also hide junk values if they exist, or bias all values in very heterogeneous problems. So, use this with care.

clearImposedFlux() → None
Clear the list of points with flux imposed.

clearImposedPressure() → None
Clear the list of points with pressure imposed.

compTessVolumes() → None
Like TesselationWrapper::computeVolumes()

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

debug(=*false*)
Activate debug messages

defTolerance(=*0.05*)
Cumulated deformation threshold for which retriangulation of pore space is performed. If negative, the triangulation update will occure with a fixed frequency on the basis of *FlowEngine::meshUpdateInterval*

dict() → dict
Return dictionary of attributes.

doInterpolate(=*false*)
Force the interpolation of cell's info while remeshing. By default, interpolation would be done only for compressible fluids. It can be forced with this flag.

dt(=*0*)
timestep [s]

edgeSize() → float
Return the number of interactions.

emulateAction() → None
get scene and run action (may be used to manipulate an engine outside the timestepping loop).

eps(=*0.00001*)
roughness defined as a fraction of particles size, giving the minimum distance between particles in the lubrication model.

epsVolMax(=*0*)
Maximal absolute volumetric strain computed at each iteration. *(auto-updated)*

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

exportMatrix([*(str)filename='matrix'*]) → None
Export system matrix to a file with all entries (even zeros will displayed).

exportTriplets([*(str)filename='triplets'*]) → None
Export system matrix to a file with only non-zero entries.

first(=*true*)
Controls the initialization/update phases

fluidBulkModulus(=*0.*)
Bulk modulus of fluid (inverse of compressibility) K=-dP*V/dV [Pa]. Flow is compressible if fluidBulkModulus > 0, else incompressible.

fluidForce(*(int)idSph*) → Vector3
Return the fluid force on sphere idSph.

forceMetis
If true, METIS is used for matrix preconditioning, else Cholmod is free to choose the best method (which may be METIS to, depending on the matrix). See `nmethods` in Cholmod documentation

getBoundaryFlux(*(int)boundary*) → float
Get total flux through boundary defined by its body id.

Note: The flux may be not zero even for no-flow condition. This artifact comes from cells which are incident to two or more boundaries (along the edges of the sample, typically). Such flux evaluation on impermeable boundary is just irrelevant, it does not imply that the boundary condition is not applied properly.

---

getCell(*(float)arg2*, *(float)arg3*, *(float)pos*) → int
get id of the cell containing (X,Y,Z).

getCellBarycenter(*(int)id*) → Vector3
get barycenter of cell 'id'.

getCellCenter(*(int)id*) → Vector3
get voronoi center of cell 'id'.

getCellFlux(*(int)cond*) → float
Get influx in cell associated to an imposed P (indexed using 'cond').

getCellPImposed(*(int)id*) → bool
get the status of cell 'id' wrt imposed pressure.

getCellPressure(*(int)id*) → float
get pressure in cell 'id'.

getConstrictions([*(bool)all=True*]) → list
Get the list of constriction radii (inscribed circle) for all finite facets (if all==True) or all facets not incident to a virtual bounding sphere (if all==False). When all facets are returned, negative radii denote facet incident to one or more fictious spheres.

getConstrictionsFull([*(bool)all=True*]) → list
Get the list of constrictions (inscribed circle) for all finite facets (if all==True), or all facets not incident to a fictious bounding sphere (if all==False). When all facets are returned, negative radii denote facet incident to one or more fictious spheres. The constrictions are returned in the format {{cell1,cell2}{rad,nx,ny,nz}}

getPorePressure(*(Vector3)pos*) → float
Measure pore pressure in position pos[0],pos[1],pos[2]

getVertices(*(int)id*) → list
get the vertices of a cell

ids(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

ignoredBody(=*-1*)
Id of a sphere to exclude from the triangulation.)

imposeFlux(*(Vector3)pos*, *(float)p*) → None
Impose a flux in cell located at 'pos' (i.e. add a source term in the flow problem). Outflux positive, influx negative.

imposePressure(*(Vector3)pos*, *(float)p*) → int
Impose pressure in cell of location 'pos'. The index of the condition is returned (for multiple imposed pressures at different points).

imposePressureFromId(*(int)id*, *(float)p*) → int
Impose pressure in cell of index 'id' (after remeshing the same condition will apply for the same location, regardless of what the new cell index is at this location). The index of the condition itself is returned (for multiple imposed pressures at different points).

isActivated(=*true*)
Activates Flow Engine

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxKdivKmean(=*100*)
define the max K value (see *FlowEngine::clampKValues*)

meanKStat(=*false*)
report the local permeabilities' correction

meshUpdateInterval(*=1000*)
Maximum number of timesteps between re-triangulation events. See also *FlowEngine::defTolerance*.

metisUsed() → bool
check wether metis lib is effectively used

minKdivKmean(*=0.0001*)
define the min K value (see *FlowEngine::clampKValues*)

multithread(*=false*)
Build triangulation and factorize in the background (multi-thread mode)

nCells() → int
get the total number of finite cells in the triangulation.

normalLubForce(*(int)idSph*) → Vector3
Return the normal lubrication force on sphere idSph.

normalLubrication(*=false*)
compute normal lubrication force as developped by Brule

normalVect(*(int)idSph*) → Vector3
Return the normal vector between particles.

normalVelocity(*(int)idSph*) → Vector3
Return the normal velocity of the interaction.

numFactorizeThreads(*=1*)
number of openblas threads in the factorization phase

numSolveThreads(*=1*)
number of openblas threads in the solve phase.

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

onlySpheresInteractions(*(int)interaction*) → int
Return the id of the interaction only between spheres.

pZero(*=0*)
The value used for initializing pore pressure. It is useless for incompressible fluid, but important for compressible model.

permeabilityFactor(*=1.0*)
permability multiplier

permeabilityMap(*=false*)
Enable/disable stocking of average permeability scalar in cell infos.

porosity(*=0*)
Porosity computed at each retriangulation *(auto-updated)*

pressureForce(*=true*)
compute the pressure field and associated fluid forces. WARNING: turning off means fluid flow is not computed at all.

pressureProfile(*(float)wallUpY*, *(float)wallDownY*) → None
Measure pore pressure in 6 equally-spaced points along the height of the sample

pumpTorque(*=false*)
Compute pump torque applied on particles

relax(*=1.9*)
Gauss-Seidel relaxation

saveVtk([*(str)folder='./VTK'*]) → None
Save pressure field in vtk format. Specify a folder name for output.

setCellPImposed(*(int)id*, *(bool)pImposed*) → None
make cell 'id' assignable with imposed pressure.

setCellPressure(*(int)id*, *(float)pressure*) → None
set pressure in cell 'id'.

setImposedPressure(*(int)cond*, *(float)p*) → None
Set pressure value at the point indexed 'cond'.

shearLubForce(*(int)idSph*) → Vector3
Return the shear lubrication force on sphere idSph.

shearLubTorque(*(int)idSph*) → Vector3
Return the shear lubrication torque on sphere idSph.

shearLubrication(=*false*)
compute shear lubrication force as developped by Brule (FIXME: ref.)

shearVelocity(*(int)idSph*) → Vector3
Return the shear velocity of the interaction.

sineAverage(=*0*)
Pressure value (average) when sinusoidal pressure is applied

sineMagnitude(=*0*)
Pressure value (amplitude) when sinusoidal pressure is applied (p)

slipBoundary(=*true*)
Controls friction condition on lateral walls

stiffness(=*10000*)
equivalent contact stiffness used in the lubrication model

surfaceDistanceParticle(*(int)interaction*) → float
Return the distance between particles.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

tolerance(=*1e-06*)
Gauss-Seidel tolerance

twistTorque(=*false*)
Compute twist torque applied on particles

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updateBCs() → None
tells the engine to update it's boundary conditions before running (especially useful when changing boundary pressure - should not be needed for point-wise imposed pressure)

updateTriangulation(=*0*)
If true the medium is retriangulated. Can be switched on to force retriangulation after some events (else it will be true periodicaly based on *FlowEngine::defTolerance* and *FlowEngine::meshUpdateInterval*. Of course, it costs CPU time.

useSolver(=*0*)
Solver to use 0=G-Seidel, 1=Taucs, 2-Pardiso, 3-CHOLMOD

viscosity(=*1.0*)
viscosity of the fluid

viscousNormalBodyStress(=*false*)
compute normal viscous stress applied on each body

`viscousShear`(=*false*)
compute viscous shear terms as developped by Donia Marzougui (FIXME: ref.)

`viscousShearBodyStress`(=*false*)
compute shear viscous stress applied on each body

`volume`([*(int)id=0*]) → float
Returns the volume of Voronoi's cell of a sphere.

`wallIds`(=*vector<int>(6)*)
body ids of the boundaries (default values are ok only if aabbWalls are appended before spheres, i.e. numbered 0,...,5)

`wallThickness`(=*0*)
Walls thickness

`waveAction`(=*false*)
Allow sinusoidal pressure condition to simulate ocean waves

`xmax`(=*1*)
See *FlowEngine::xmin*.

`xmin`(=*0*)
Index of the boundary $\mathtt{x}_{\mathtt{min}}$. This index is not equal the the id of the corresponding body in general, it may be used to access the corresponding attributes (e.g. flow.bndCondValue[flow.xmin], flow.wallId[flow.xmin],...).

`ymax`(=*3*)
See *FlowEngine::xmin*.

`ymin`(=*2*)
See *FlowEngine::xmin*.

`zmax`(=*5*)
See *FlowEngine::xmin*.

`zmin`(=*4*)
See *FlowEngine::xmin*.

class `yade.wrapper.ForceEngine`(*inherits PartialEngine → Engine → Serializable*)
Apply contact force on some particles at each step.

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`force`(=*Vector3r::Zero()*)
Force to apply.

`ids`(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes

openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**HarmonicMotionEngine**(*inherits* *KinematicEngine* → *PartialEngine* → *Engine* → *Serializable*)
This engine implements the harmonic oscillation of bodies. http://en.wikipedia.org/wiki/Simple_harmonic_motion#Dynamics_of_simple_harmonic_motion

A(=*Vector3r::Zero()*)
Amplitude [m]

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

f(=*Vector3r::Zero()*)
Frequency [hertz]

fi(=*Vector3r(Mathr::PI/2.0, Mathr::PI/2.0, Mathr::PI/2.0)*)
Initial phase [radians]. By default, the body oscillates around initial position.

ids(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**HarmonicRotationEngine**(*inherits* *RotationEngine* → *KinematicEngine* → *PartialEngine* → *Engine* → *Serializable*)
This engine implements the harmonic-rotation oscillation of bodies. http://en.wikipedia.org/wiki/Simple_harmonic_motion#Dynamics_of_simple_harmonic_motion ; please, set dynamic=False for bodies, droven by this engine, otherwise amplitude will be 2x more, than awaited.

`A`(=*0*)
Amplitude [rad]

`angularVelocity`(=*0*)
Angular velocity. [rad/s]

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`f`(=*0*)
Frequency [hertz]

`fi`(=*Mathr::PI/2.0*)
Initial phase [radians]. By default, the body oscillates around initial position.

`ids`(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`rotateAroundZero`(=*false*)
If True, bodies will not rotate around their centroids, but rather around `zeroPoint`.

`rotationAxis`(=*Vector3r::UnitX()*)
Axis of rotation (direction); will be normalized automatically.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`zeroPoint`(=*Vector3r::Zero()*)
Point around which bodies will rotate if `rotateAroundZero` is True

class `yade.wrapper.HelixEngine`(*inherits* *RotationEngine* → *KinematicEngine* → *PartialEngine* → *Engine* → *Serializable*)
Engine applying both rotation and translation, along the same axis, whence the name HelixEngine

`angleTurned`(=*0*)
How much have we turned so far. *(auto-updated)* [rad]

`angularVelocity`(=*0*)
Angular velocity. [rad/s]

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`ids`(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`linearVelocity`(=*0*)
Linear velocity [m/s]

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`rotateAroundZero`(=*false*)
If True, bodies will not rotate around their centroids, but rather around `zeroPoint`.

`rotationAxis`(=*Vector3r::UnitX()*)
Axis of rotation (direction); will be normalized automatically.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`zeroPoint`(=*Vector3r::Zero()*)
Point around which bodies will rotate if `rotateAroundZero` is True

class `yade.wrapper.HydroForceEngine`(*inherits PartialEngine → Engine → Serializable*)

Apply drag and lift due to a fluid flow vector (1D) to each sphere + the buoyant weight.
The applied drag force reads

$$F_d = \frac{1}{2} C_d A \rho^f |\nu_f - \nu| \nu_f - \nu$$

where ρ is the medium density (*densFluid*), ν is particle's velocity, ν_f is the velocity of the fluid at the particle center(*vxFluid*), A is particle projected area (disc), C_d is the drag coefficient. The formulation of the drag coefficient depends on the local particle reynolds number and the solid volume fraction. The formulation of the drag is *[Dallavalle1948] [RevilBaudard2013]* with a correction of Richardson-Zaki [Richardson1954] to take into account the hindrance effect. This law is classical in sediment transport. It is possible to activate a fluctuation of the drag force for each particle which account for the turbulent fluctuation of the fluid velocity (*velFluct*). The model implemented for the turbulent velocity fluctuation is a simple discrete random walk which takes as input the Reynolds stress tensor R^f_{xz} as a function of the depth, and allows to recover the main property of the fluctuations by imposing $< u'_x u'_z > (z) =< R^f_{xz} > (z)/\rho^f$. It requires as input $< R^f_{xz} > (z)/\rho^f$ called *simplifiedReynoldStresses* in the code. The formulation of the lift is taken from *[Wiberg1985]* and is such that :

$$F_L = \frac{1}{2} C_L A \rho^f ((\nu_f - \nu)^2_{top} - (\nu_f - \nu)^2_{bottom})$$

Where the subscript top and bottom means evaluated at the top (respectively the bottom) of the sphere considered. This formulation of the lift account for the difference of pressure at the top and the bottom of the particle inside a turbulent shear flow. As this formulation is controversial when approaching the threshold of motion *[Schmeeckle2007]* it is possible to desactivate it with the variable *lift*. The buoyancy is taken into account through the buoyant weight :

$F_{buoyancy} = -\rho^f V^p g$

, where g is the gravity vector along the vertical, and V^p is the volume of the particle. This engine also evaluate the average particle velocity, solid volume fraction and drag force depth profiles, through the function averageProfile. This is done as the solid volume fraction depth profile is required for the drag calculation, and as the three are required for the independent fluid resolution.

`Cl`(=*0.2*)
Value of the lift coefficient taken from *[Wiberg1985]*

`activateAverage`(=*false*)
If true, activate the calculation of the average depth profiles of drag, solid volume fraction, and solid velocity for the application of the force (phiPart in hindrance function) and to use in python for the coupling with the fluid.

`averageDrag`(=*uninitalized*)
Discretized average drag depth profile. No role in the engine, output parameter. For practical reason, it can be evaluated directly inside the engine, calling from python the averageProfile() method of the engine, or puting *activateAverage* to True.

`averageDrag1`(=*uninitalized*)
Discretized average drag depth profile of particles of type 1. Evaluated when *twoSize* is set to True.

`averageDrag2`(=*uninitalized*)
Discretized average drag depth profile of particles of type 2. Evaluated when *twoSize* is set to True.

`averageProfile()` → None
Compute and store the particle velocity (*vxPart*, *vyPart*, *vzPart*) and solid volume fraction (*phiPart*) depth profile. For each defined cell z, the k component of the average particle velocity reads:

$< v_k >^z = \sum_p V^p v_k^p / \sum_p V^p$,

where the sum is made over the particles contained in the cell, v_k^p is the k component of the velocity associated to particle p, and V^p is the part of the volume of the particle p contained inside the cell. This definition allows to smooth the averaging, and is equivalent to taking into account the center of the particles only when there is a lot of particles in each cell. As for the solid volume fraction, it is evaluated in the same way: for each defined cell z, it reads:

$< \varphi >^z = \frac{1}{V_{cell}} \sum_p V^p$, where V_{cell} is the volume of the cell considered, and V^p is the volume of particle p contained in cell z. This function gives depth profiles of average velocity and solid volume fraction, returning the average quantities in each cell of height dz, from the reference horizontal plane at elevation *zRef* (input parameter) until the plane of elevation *zRef* plus *nCell* times *deltaZ* (input parameters). When the option *twoSize* is set to True, evaluate in addition the average drag (*averageDrag1* and *averageDrag2*) and solid volume fraction (*phiPart1* and *phiPart2*) depth profiles considering only the particles of radius respectively *radiusPart1* and *radiusPart2* in the averaging.

`bedElevation`(=*uninitalized*)
Elevation of the bed above which the fluid flow is turbulent and the particles undergo turbulent velocity fluctuation.

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

deltaZ(=*uninitalized*)
Height of the discretization cell.

densFluid(=*1000*)
Density of the fluid, by default - density of water

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

expoRZ(=*3.1*)
Value of the Richardson-Zaki exponent, for the drag correction due to hindrance

gravity(=*Vector3r(0, 0, -9.81)*)
Gravity vector (may depend on the slope).

ids(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

lift(=*false*)
Option to activate or not the evaluation of the lift

nCell(=*uninitalized*)
Number of cell in the depth

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

phiPart(=*uninitalized*)
Discretized solid volume fraction depth profile. Can be taken as input parameter, or evaluated directly inside the engine, calling from python the averageProfile() function, or puting *activateAverage* to True.

phiPart1(=*uninitalized*)
Discretized solid volume fraction depth profile of particles of type 1. Evaluated when *twoSize* is set to True.

phiPart2(=*uninitalized*)
Discretized solid volume fraction depth profile of particles of type 2. Evaluated when *twoSize* is set to True.

radiusPart1(=*0.*)
Radius of the particles of type 1. Useful only when *twoSize* is set to True.

radiusPart2(=*0.*)
Radius of the particles of type 2. Useful only when *twoSize* is set to True.

simplifiedReynoldStresses(=*uninitalized*)
Vector of size equal to *nCell* containing the Reynolds stresses divided by the fluid density in function of the depth. simplifiedReynoldStresses(z) $=< u_x' u_z' > (z)^2$

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

twoSize(=*false*)
Option to activate when considering two particle size in the simulation. When activated evaluate the average solid volume fraction and drag force for the two type of particles of diameter diameterPart1 and diameterPart2 independently.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

vCell(=*uninitalized*)
Volume of averaging cell

vFluctX(=*uninitalized*)
Vector associating a streamwise fluid velocity fluctuation to each particle. Fluctuation calculated in the C++ code from the discrete random walk model

vFluctZ(=*uninitalized*)
Vector associating a normal fluid velocity fluctuation to each particle. Fluctuation calculated in the C++ code from the discrete random walk model

velFluct(=*false*)
If true, activate the determination of turbulent fluid velocity fluctuation for the next time step only at the position of each particle, using a simple discrete random walk (DRW) model based on the Reynolds stresses profile (*simplifiedReynoldStresses*)

viscoDyn(=*1e-3*)
Dynamic viscosity of the fluid, by default - viscosity of water

vxFluid(=*uninitalized*)
Discretized streamwise fluid velocity depth profile

vxPart(=*uninitalized*)
Discretized streamwise solid velocity depth profile. Can be taken as input parameter, or evaluated directly inside the engine, calling from python the averageProfile() function, or puting *activateAverage* to True.

vyPart(=*uninitalized*)
Discretized spanwise solid velocity depth profile. No role in the engine, output parameter. For practical reason, it can be evaluated directly inside the engine, calling from python the averageProfile() method of the engine, or puting *activateAverage* to True.

vzPart(=*uninitalized*)
Discretized normal solid velocity depth profile. No role in the engine, output parameter. For practical reason, it can be evaluated directly inside the engine, calling from python the averageProfile() method of the engine, or puting *activateAverage* to True.

zRef(=*uninitalized*)
Position of the reference point which correspond to the first value of the fluid velocity, i.e. to the ground.

class yade.wrapper.**InterpolatingDirectedForceEngine**(*inherits ForceEngine → PartialEngine → Engine → Serializable*)
Engine for applying force of varying magnitude but constant direction on subscribed bodies. times and magnitudes must have the same length, direction (normalized automatically) gives the orientation.

As usual with interpolating engines: the first magnitude is used before the first time point, last magnitude is used after the last time point. Wrap specifies whether time wraps around the last time point to the first time point.

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
Return dictionary of attributes.

`direction`(=*Vector3r::UnitX()*)
Contact force direction (normalized automatically)

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`force`(=*Vector3r::Zero()*)
Force to apply.

`ids`(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`magnitudes`(=*uninitalized*)
Force magnitudes readings [N]

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`times`(=*uninitalized*)
Time readings [s]

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wrap`(=*false*)
wrap to the beginning of the sequence if beyond the last time point

class `yade.wrapper.InterpolatingHelixEngine`(*inherits* *HelixEngine* → *RotationEngine* → *KinematicEngine* → *PartialEngine* → *Engine* → *Serializable*)
Engine applying spiral motion, finding current angular velocity by linearly interpolating in times and velocities and translation by using slope parameter.

The interpolation assumes the margin value before the first time point and last value after the last time point. If wrap is specified, time will wrap around the last times value to the first one (note that no interpolation between last and first values is done).

`angleTurned`(=*0*)
How much have we turned so far. *(auto-updated)* [rad]

`angularVelocities`(=*uninitalized*)
List of angular velocities; manadatorily of same length as times. [rad/s]

`angularVelocity`(=*0*)
Angular velocity. [rad/s]

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`ids`(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`linearVelocity`(=*0*)
Linear velocity [m/s]

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`rotateAroundZero`(=*false*)
If True, bodies will not rotate around their centroids, but rather around `zeroPoint`.

`rotationAxis`(=*Vector3r::UnitX()*)
Axis of rotation (direction); will be normalized automatically.

`slope`(=*0*)
Axial translation per radian turn (can be negative) [m/rad]

`times`(=*uninitalized*)
List of time points at which velocities are given; must be increasing [s]

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wrap`(=*false*)
Wrap t if t>times_n, i.e. t_wrapped=t-N*(times_n-times_0)

`zeroPoint`(=*Vector3r::Zero()*)
Point around which bodies will rotate if `rotateAroundZero` is True

class `yade.wrapper.KinematicEngine`(*inherits* *PartialEngine* → *Engine* → *Serializable*)
Abstract engine for applying prescribed displacement.

Note: Derived classes should override the `apply` with given list of `ids` (not `action` with *PartialEngine.ids*), so that they work when combined together; *velocity* and *angular velocity* of all subscribed bodies is reset before the `apply` method is called, it should therefore only increment those quantities.

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
Return dictionary of attributes.

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`ids`(=*uninitalized*)
: *Ids* of bodies affected by this PartialEngine.

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

class `yade.wrapper.LawTester`(*inherits* *PartialEngine* → *Engine* → *Serializable*)

Prescribe and apply deformations of an interaction in terms of local mutual displacements and rotations. The loading path is specified either using *path* (as sequence of 6-vectors containing generalized displacements u_x, u_y, u_z, φ_x, φ_y, φ_z) or *disPath* (u_x, u_y, u_z) and *rotPath* (φ_x, φ_y, φ_z). Time function with time values (step numbers) corresponding to points on loading path is given by *pathSteps*. Loading values are linearly interpolated between given loading path points, and starting zero-value (the initial configuration) is assumed for both *path* and *pathSteps*. *hooks* can specify python code to run when respective point on the path is reached; when the path is finished, *doneHook* will be run.

LawTester should be placed between *InteractionLoop* and *NewtonIntegrator* in the simulation loop, since it controls motion via setting linear/angular velocities on particles; those velocities are integrated by *NewtonIntegrator* to yield an actual position change, which in turn causes *IGeom* to be updated (and *contact law* applied) when *InteractionLoop* is executed. Constitutive law generating forces on particles will not affect prescribed particle motion, since both particles have all *DoFs blocked* when first used with LawTester.

LawTester uses, as much as possible, *IGeom* to provide useful data (such as local coordinate system), but is able to compute those independently if absent in the respective *IGeom*:

IGeom	#DoFs	LawTester support level
L3Geom	3	full
L6Geom	6	full
ScGeom	3	emulate local coordinate system
ScGeom6D	6	emulate local coordinate system

Depending on *IGeom*, 3 (u_x, u_y, u_z) or 6 (u_x, u_y, u_z, φ_x, φ_y, φ_z) degrees of freedom (DoFs) are controlled with LawTester, by prescribing linear and angular velocities of both particles in contact. All DoFs controlled with LawTester are orthogonal (fully decoupled) and are controlled independently.

When 3 DoFs are controlled, *rotWeight* controls whether local shear is applied by moving particle on arc around the other one, or by rotating without changing position; although such rotation induces mutual rotation on the interaction, it is ignored with *IGeom* with only 3 DoFs. When 6 DoFs are controlled, only arc-displacement is applied for shear, since otherwise mutual rotation would occur.

idWeight distributes prescribed motion between both particles (resulting local deformation is the same if `id1` is moved towards `id2` or `id2` towards `id1`). This is true only for u_x, u_y, u_z, φ_x however ; bending rotations φ_y, φ_z are nevertheless always distributed regardless of `idWeight` to both spheres in inverse proportion to their radii, so that there is no shear induced.

LawTester knows current contact deformation from 2 sources: from its own internal data (which are used for prescribing the displacement at every step), which can be accessed in *uTest*, and from *IGeom* itself (depending on which data it provides), which is stored in *uGeom*. These two values should be identical (disregarding numerical percision), and it is a way to test whether *IGeom* and related functors compute what they are supposed to compute.

LawTester-operated interactions can be rendered with *GlExtra_LawTester* renderer.

See scripts/test/law-test.py for an example.

`dead`(=*false*)
: If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
: Return dictionary of attributes.

`disPath`(=*uninitalized*)
: Loading path, where each Vector3 contains desired normal displacement and two components of the shear displacement (in local coordinate system, which is being tracked automatically. If shorter than *rotPath*, the last value is repeated.

`displIsRel`(=*true*)
: Whether displacement values in *disPath* are normalized by reference contact length (r1+r2 for 2 spheres).

`doneHook`(=*uninitalized*)
: Python command (as string) to run when end of the path is achieved. If empty, the engine will be set *dead*.

`execCount`
: Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`hooks`(=*uninitalized*)
: Python commands to be run when the corresponding point in path is reached, before doing other things in that particular step. See also *doneHook*.

`idWeight`(=*1*)
: Float, usually ⟨0,1⟩, determining on how are displacements distributed between particles (0 for id1, 1 for id2); intermediate values will apply respective part to each of them. This parameter is ignored with 6-DoFs *IGeom*.

`ids`(=*uninitalized*)
: *Ids* of bodies affected by this PartialEngine.

`label`(=*uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
: Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

pathSteps(=*vector<int>(1, 1), (constant step)*)
Step number for corresponding values in *path*; if shorter than path, distance between last 2 values is used for the rest.

refLength(=*0*)
Reference contact length, for rendering only.

renderLength(=*0*)
Characteristic length for the purposes of rendering, set equal to the smaller radius.

rotPath(=*uninitalized*)
Rotational components of the loading path, where each item contains torsion and two bending rotations in local coordinates. If shorter than *path*, the last value is repeated.

rotWeight(=*1*)
Float ⟨0,1⟩ determining whether shear displacement is applied as rotation or displacement on arc (0 is displacement-only, 1 is rotation-only). Not effective when mutual rotation is specified.

step(=*1*)
Step number in which this engine is active; determines position in path, using pathSteps.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

trsf(=*uninitalized*)
Transformation matrix for the local coordinate system. *(auto-updated)*

uGeom(=*Vector6r::Zero()*)
Current generalized displacements (3 displacements, 3 rotations), as stored in the interation itself. They should corredpond to *uTest*, otherwise a bug is indicated.

uTest(=*Vector6r::Zero()*)
Current generalized displacements (3 displacements, 3 rotations), as they should be according to this *LawTester*. Should correspond to *uGeom*.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

uuPrev(=*Vector6r::Zero()*)
Generalized displacement values reached in the previous step, for knowing which increment to apply in the current step.

class yade.wrapper.**LinearDragEngine**(*inherits PartialEngine → Engine → Serializable*)
Apply viscous resistance or linear drag on some particles at each step, decelerating them proportionally to their linear velocities. The applied force reads

$$F_d = -b\nu$$

where b is the linear drag, ν is particle's velocity.

$$b = 6\pi\nu r$$

where ν is the medium viscosity, r is the Stokes radius of the particle (but in this case we accept it equal to sphere radius for simplification),

---

Note: linear drag is only applied to spherical particles, listed in ids.

---

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

ids(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

nu(=*0.001*)
Viscosity of the medium.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**PeriodicFlowEngine**(*inherits* *FlowEngine_PeriodicInfo* → *PartialEngine* → *Engine* → *Serializable*)
A variant of *FlowEngine* implementing periodic boundary conditions. The API is very similar.

OSI() → float
Return the number of interactions only between spheres.

avFlVelOnSph(*(int)idSph*) → object
compute a sphere-centered average fluid velocity

averagePressure() → float
Measure averaged pore pressure in the entire volume

averageSlicePressure(*(float)posY*) → float
Measure slice-averaged pore pressure at height posY

averageVelocity() → Vector3
measure the mean velocity in the period

blockCell(*(int)id*, *(bool)blockPressure*) → None
block cell 'id'. The cell will be excluded from the fluid flow problem and the conductivity of all incident facets will be null. If blockPressure=False, deformation is reflected in the pressure, else it is constantly 0.

blockHook(=*""*)
Python command to be run when remeshing. Anticipated usage: define blocked cells (see also *TemplateFlowEngine_FlowEngine_PeriodicInfo.blockCell*), or apply exotic types of boundary conditions which need to visit the newly built mesh

bndCondIsPressure(=*vector<bool>(6, false)*)
defines the type of boundary condition for each side. True if pressure is imposed, False for no-flux. Indexes can be retrieved with *FlowEngine::xmin* and friends.

bndCondValue(=*vector<double>(6, 0)*)
Imposed value of a boundary condition. Only applies if the boundary condition is imposed pressure, else the imposed flux is always zero presently (may be generalized to non-zero imposed fluxes in the future).

bodyNormalLubStress(*(int)idSph*) → Matrix3
Return the normal lubrication stress on sphere idSph.

bodyShearLubStress(*(int)idSph*) → Matrix3
Return the shear lubrication stress on sphere idSph.

boundaryPressure(=*vector<Real>()*)
values defining pressure along x-axis for the top surface. See also *FlowEngine_PeriodicInfo::boundaryXPos*

boundaryUseMaxMin(=*vector<bool>(6, true)*)
If true (default value) bounding sphere is added as function of max/min sphere coord, if false as function of yade wall position

boundaryVelocity(=*vector<Vector3r>(6, Vector3r::Zero())*)
velocity on top boundary, only change it using *FlowEngine::setBoundaryVel*

boundaryXPos(=*vector<Real>()*)
values of the x-coordinate for which pressure is defined. See also *FlowEngine_PeriodicInfo::boundaryPressure*

cholmodStats() → None
get statistics of cholmod solver activity

clampKValues(=*true*)
If true, clamp local permeabilities in [minKdivKmean,maxKdivKmean]*globalK. This clamping can avoid singular values in the permeability matrix and may reduce numerical errors in the solve phase. It will also hide junk values if they exist, or bias all values in very heterogeneous problems. So, use this with care.

clearImposedFlux() → None
Clear the list of points with flux imposed.

clearImposedPressure() → None
Clear the list of points with pressure imposed.

compTessVolumes() → None
Like TesselationWrapper::computeVolumes()

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

debug(=*false*)
Activate debug messages

defTolerance(=*0.05*)
Cumulated deformation threshold for which retriangulation of pore space is performed. If negative, the triangulation update will occure with a fixed frequency on the basis of *FlowEngine::meshUpdateInterval*

dict() → dict
Return dictionary of attributes.

doInterpolate(=*false*)
Force the interpolation of cell's info while remeshing. By default, interpolation would be done only for compressible fluids. It can be forced with this flag.

dt(=*0*)
timestep [s]

duplicateThreshold(=*0.06*)
distance from cell borders that will triger periodic duplication in the triangulation *(auto-updated)*

edgeSize() → float
Return the number of interactions.

emulateAction() → None
get scene and run action (may be used to manipulate an engine outside the timestepping loop).

eps(*=0.00001*)
roughness defined as a fraction of particles size, giving the minimum distance between particles in the lubrication model.

epsVolMax(*=0*)
Maximal absolute volumetric strain computed at each iteration. *(auto-updated)*

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

exportMatrix(*[(str)filename='matrix']*) → None
Export system matrix to a file with all entries (even zeros will displayed).

exportTriplets(*[(str)filename='triplets']*) → None
Export system matrix to a file with only non-zero entries.

first(*=true*)
Controls the initialization/update phases

fluidBulkModulus(*=0.*)
Bulk modulus of fluid (inverse of compressibility) K=-dP*V/dV [Pa]. Flow is compressible if fluidBulkModulus > 0, else incompressible.

fluidForce(*(int)idSph*) → Vector3
Return the fluid force on sphere idSph.

forceMetis
If true, METIS is used for matrix preconditioning, else Cholmod is free to choose the best method (which may be METIS to, depending on the matrix). See `nmethods` in Cholmod documentation

getBoundaryFlux(*(int)boundary*) → float
Get total flux through boundary defined by its body id.

Note: The flux may be not zero even for no-flow condition. This artifact comes from cells which are incident to two or more boundaries (along the edges of the sample, typically). Such flux evaluation on impermeable boundary is just irrelevant, it does not imply that the boundary condition is not applied properly.

---

getCell(*(float)arg2*, *(float)arg3*, *(float)pos*) → int
get id of the cell containing (X,Y,Z).

getCellBarycenter(*(int)id*) → Vector3
get barycenter of cell 'id'.

getCellCenter(*(int)id*) → Vector3
get voronoi center of cell 'id'.

getCellFlux(*(int)cond*) → float
Get influx in cell associated to an imposed P (indexed using 'cond').

getCellPImposed(*(int)id*) → bool
get the status of cell 'id' wrt imposed pressure.

getCellPressure(*(int)id*) → float
get pressure in cell 'id'.

getConstrictions(*[(bool)all=True]*) → list
Get the list of constriction radii (inscribed circle) for all finite facets (if all==True) or all facets not incident to a virtual bounding sphere (if all==False). When all facets are returned, negative radii denote facet incident to one or more fictious spheres.

getConstrictionsFull(*[(bool)all=True]*) → list
Get the list of constrictions (inscribed circle) for all finite facets (if all==True), or all facets

not incident to a fictious bounding sphere (if all==False). When all facets are returned, negative radii denote facet incident to one or more fictious spheres. The constrictions are returned in the format {{cell1,cell2}{rad,nx,ny,nz}}

getPorePressure(*(Vector3)pos*) → float
Measure pore pressure in position pos[0],pos[1],pos[2]

getVertices(*(int)id*) → list
get the vertices of a cell

gradP(=*Vector3r::Zero()*)
Macroscopic pressure gradient

ids(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

ignoredBody(=*-1*)
Id of a sphere to exclude from the triangulation.)

imposeFlux(*(Vector3)pos*, *(float)p*) → None
Impose a flux in cell located at 'pos' (i.e. add a source term in the flow problem). Outflux positive, influx negative.

imposePressure(*(Vector3)pos*, *(float)p*) → int
Impose pressure in cell of location 'pos'. The index of the condition is returned (for multiple imposed pressures at different points).

imposePressureFromId(*(int)id*, *(float)p*) → int
Impose pressure in cell of index 'id' (after remeshing the same condition will apply for the same location, regardless of what the new cell index is at this location). The index of the condition itself is returned (for multiple imposed pressures at different points).

isActivated(=*true*)
Activates Flow Engine

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

maxKdivKmean(=*100*)
define the max K value (see *FlowEngine::clampKValues*)

meanKStat(=*false*)
report the local permeabilities' correction

meshUpdateInterval(=*1000*)
Maximum number of timesteps between re-triangulation events. See also *FlowEngine::defTolerance*.

metisUsed() → bool
check wether metis lib is effectively used

minKdivKmean(=*0.0001*)
define the min K value (see *FlowEngine::clampKValues*)

multithread(=*false*)
Build triangulation and factorize in the background (multi-thread mode)

nCells() → int
get the total number of finite cells in the triangulation.

normalLubForce(*(int)idSph*) → Vector3
Return the normal lubrication force on sphere idSph.

normalLubrication(=*false*)
compute normal lubrication force as developped by Brule

normalVect(*(int)idSph*) → Vector3
Return the normal vector between particles.

normalVelocity(*(int)idSph*) → Vector3
Return the normal velocity of the interaction.

numFactorizeThreads(*=1*)
number of openblas threads in the factorization phase

numSolveThreads(*=1*)
number of openblas threads in the solve phase.

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

onlySpheresInteractions(*(int)interaction*) → int
Return the id of the interaction only between spheres.

pZero(*=0*)
The value used for initializing pore pressure. It is useless for incompressible fluid, but important for compressible model.

permeabilityFactor(*=1.0*)
permability multiplier

permeabilityMap(*=false*)
Enable/disable stocking of average permeability scalar in cell infos.

porosity(*=0*)
Porosity computed at each retriangulation *(auto-updated)*

pressureForce(*=true*)
compute the pressure field and associated fluid forces. WARNING: turning off means fluid flow is not computed at all.

pressureProfile(*(float)wallUpY*, *(float)wallDownY*) → None
Measure pore pressure in 6 equally-spaced points along the height of the sample

pumpTorque(*=false*)
Compute pump torque applied on particles

relax(*=1.9*)
Gauss-Seidel relaxation

saveVtk(*[(str)folder='./VTK']*) → None
Save pressure field in vtk format. Specify a folder name for output.

setCellPImposed(*(int)id*, *(bool)pImposed*) → None
make cell 'id' assignable with imposed pressure.

setCellPressure(*(int)id*, *(float)pressure*) → None
set pressure in cell 'id'.

setImposedPressure(*(int)cond*, *(float)p*) → None
Set pressure value at the point indexed 'cond'.

shearLubForce(*(int)idSph*) → Vector3
Return the shear lubrication force on sphere idSph.

shearLubTorque(*(int)idSph*) → Vector3
Return the shear lubrication torque on sphere idSph.

shearLubrication(*=false*)
compute shear lubrication force as developped by Brule (FIXME: ref.)

shearVelocity(*(int)idSph*) → Vector3
Return the shear velocity of the interaction.

sineAverage(*=0*)
Pressure value (average) when sinusoidal pressure is applied

sineMagnitude(*=0*)
Pressure value (amplitude) when sinusoidal pressure is applied (p)

slipBoundary(*=true*)
Controls friction condition on lateral walls

stiffness(*=10000*)
equivalent contact stiffness used in the lubrication model

surfaceDistanceParticle(*(int)interaction*) → float
Return the distance between particles.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

tolerance(*=1e-06*)
Gauss-Seidel tolerance

twistTorque(*=false*)
Compute twist torque applied on particles

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updateBCs() → None
tells the engine to update it's boundary conditions before running (especially useful when changing boundary pressure - should not be needed for point-wise imposed pressure)

updateTriangulation(*=0*)
If true the medium is retriangulated. Can be switched on to force retriangulation after some events (else it will be true periodicaly based on *FlowEngine::defTolerance* and *FlowEngine::meshUpdateInterval*. Of course, it costs CPU time.

useSolver(*=0*)
Solver to use 0=G-Seidel, 1=Taucs, 2-Pardiso, 3-CHOLMOD

viscosity(*=1.0*)
viscosity of the fluid

viscousNormalBodyStress(*=false*)
compute normal viscous stress applied on each body

viscousShear(*=false*)
compute viscous shear terms as developped by Donia Marzougui (FIXME: ref.)

viscousShearBodyStress(*=false*)
compute shear viscous stress applied on each body

volume(*[(int)id=0]*) → float
Returns the volume of Voronoi's cell of a sphere.

wallIds(*=vector<int>(6)*)
body ids of the boundaries (default values are ok only if aabbWalls are appended before spheres, i.e. numbered 0,...,5)

wallThickness(*=0*)
Walls thickness

waveAction(*=false*)
Allow sinusoidal pressure condition to simulate ocean waves

xmax(*=1*)
See *FlowEngine::xmin*.

xmin(=*0*)
Index of the boundary x_{min}. This index is not equal the the id of the corresponding body in general, it may be used to access the corresponding attributes (e.g. flow.bndCondValue[flow.xmin], flow.wallId[flow.xmin],...).

ymax(=*3*)
See *FlowEngine::xmin*.

ymin(=*2*)
See *FlowEngine::xmin*.

zmax(=*5*)
See *FlowEngine::xmin*.

zmin(=*4*)
See *FlowEngine::xmin*.

class yade.wrapper.**RadialForceEngine**(*inherits* *PartialEngine* → *Engine* → *Serializable*)
Apply force of given magnitude directed away from spatial axis.

axisDir(=*Vector3r::UnitX()*)
Axis direction (normalized automatically)

axisPt(=*Vector3r::Zero()*)
Point on axis

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

fNorm(=*0*)
Applied force magnitude

ids(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**RotationEngine**(*inherits* *KinematicEngine* → *PartialEngine* → *Engine* → *Serializable*)
Engine applying rotation (by setting angular velocity) to subscribed bodies. If *rotateAroundZero* is set, then each body is also displaced around *zeroPoint*.

`angularVelocity`(=*0*)
Angular velocity. [rad/s]

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`ids`(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`rotateAroundZero`(=*false*)
If True, bodies will not rotate around their centroids, but rather around `zeroPoint`.

`rotationAxis`(=*Vector3r::UnitX()*)
Axis of rotation (direction); will be normalized automatically.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`zeroPoint`(=*Vector3r::Zero()*)
Point around which bodies will rotate if `rotateAroundZero` is True

class `yade.wrapper.SPHEngine`(*inherits* *PartialEngine* → *Engine* → *Serializable*)
Apply given torque (momentum) value at every subscribed particle, at every step.

`KernFunctionDensity`(=*Lucy*)
Kernel function for density calculation (by default - Lucy). The following kernel functions are available: Lucy=1 (*[Lucy1977]* (27)), BSpline1=2 (*[Monaghan1985]* (21)), BSpline2=3 (*[Monaghan1985]* (22)).

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`h`(=*-1*)
Core radius. See Mueller *[Mueller2003]* .

`ids`(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

`k`(=*-1*)
Gas constant for SPH-interactions (only for SPH-model). See Mueller *[Mueller2003]* .

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`mask`(=*-1*)
Bitmask for SPH-particles.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`rho0`(=*-1*)
Rest density. See Mueller *[Mueller2003]* .

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.ServoPIDController`(*inherits* *TranslationEngine* → *KinematicEngine* → *PartialEngine* → *Engine* → *Serializable*)
PIDController servo-engine for applying prescribed force on bodies. http://en.wikipedia.org/wiki/PID_controller

`axis`(=*Vector3r::Zero()*)
Unit vector along which apply the velocity [-]

`curVel`(=*0.0*)
Current applied velocity [m/s]

`current`(=*Vector3r::Zero()*)
Current value for the controller [N]

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`errorCur`(=*0.0*)
Current error [N]

`errorPrev`(=*0.0*)
Previous error [N]

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`iTerm`(=*0.0*)
Integral term [N]

`ids`(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

`iterPeriod`(=*100.0*)
Periodicity criterion of velocity correlation [-]

`iterPrevStart`(=*-1.0*)
Previous iteration of velocity correlation [-]

`kD`(=*0.0*)
Derivative gain/coefficient for the PID-controller [-]

`kI`(=*0.0*)
Integral gain/coefficient for the PID-controller [-]

`kP`(=*0.0*)
Proportional gain/coefficient for the PID-controller [-]

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`maxVelocity`(=*0.0*)
Velocity [m/s]

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`target`(=*0.0*)
Target value for the controller [N]

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`translationAxis`(=*uninitalized*)
Direction [Vector3]

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`velocity`(=*uninitalized*)
Velocity [m/s]

class `yade.wrapper.StepDisplacer`(*inherits* *PartialEngine* → *Engine* → *Serializable*)
Apply generalized displacement (displacement or rotation) stepwise on subscribed bodies. Could be used for purposes of contact law tests (by moving one sphere compared to another), but in this case, see rather *LawTester*

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`ids`(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`mov`(=*Vector3r::Zero()*)
Linear displacement step to be applied per iteration, by addition to *State.pos*.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`rot`(=*Quaternionr::Identity()*)
Rotation step to be applied per iteration (via rotation composition with *State.ori*).

`setVelocities`(=*false*)
If false, positions and orientations are directly updated, without changing the speeds of concerned bodies. If true, only velocity and angularVelocity are modified. In this second case *integrator* is supposed to be used, so that, thanks to this Engine, the bodies will have the prescribed jump over one iteration (dt).

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.TorqueEngine`(*inherits PartialEngine → Engine → Serializable*)
Apply given torque (momentum) value at every subscribed particle, at every step.

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`ids`(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`moment`(=*Vector3r::Zero()*)
Torque value to be applied.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.TranslationEngine`(*inherits KinematicEngine → PartialEngine → Engine → Serializable*)
This engine is the base class for different engines, which require any kind of motion.

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`ids`(=*uninitalized*)
Ids of bodies affected by this PartialEngine.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

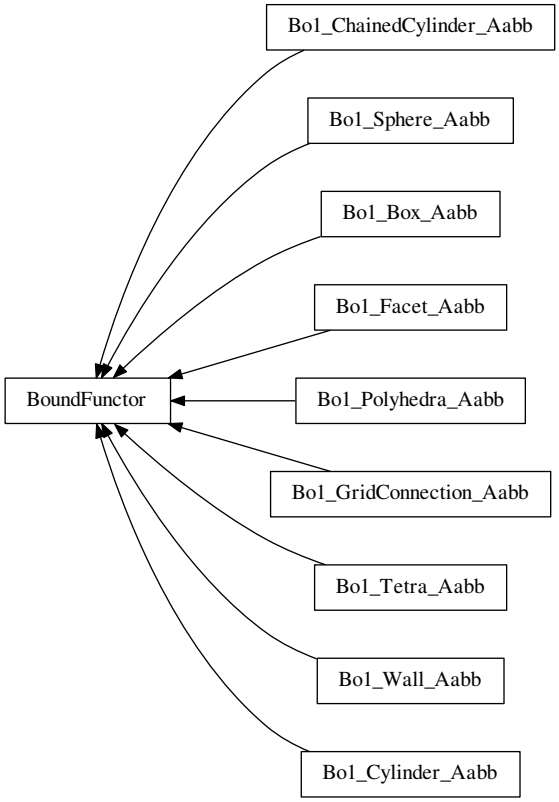
`translationAxis`(=*uninitalized*)
Direction [Vector3]

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`velocity`(=*uninitalized*)
Velocity [m/s]

8.5 Bounding volume creation

8.5.1 BoundFunctor



class `yade.wrapper.BoundFunctor`(*inherits Functor → Serializable*)
Functor for creating/updating *Body::bound*.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Bo1_Box_Aabb`(*inherits BoundFunctor → Functor → Serializable*)
Create/update an *Aabb* of a *Box*.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Bo1_ChainedCylinder_Aabb`(*inherits BoundFunctor → Functor → Serializable*)
Functor creating *Aabb* from *ChainedCylinder*.

`aabbEnlargeFactor`
Relative enlargement of the bounding box; deactivated if negative.

Note: This attribute is used to create distant interaction, but is only meaningful with an *IGeomFunctor* which will not simply discard such interactions: *Ig2_Cylinder_Cylinder_ScGeom::interactionDetectionFactor* should have the same value as *aabbEnlargeFactor*.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Bo1_Cylinder_Aabb`(*inherits BoundFunctor → Functor → Serializable*)
Functor creating *Aabb* from *Cylinder*.

`aabbEnlargeFactor`
Relative enlargement of the bounding box; deactivated if negative.

Note: This attribute is used to create distant interaction, but is only meaningful with an *IGeomFunctor* which will not simply discard such interactions: *Ig2_Cylinder_Cylinder_ScGeom::interactionDetectionFactor* should have the same value as *aabbEnlargeFactor*.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.Bo1_Facet_Aabb(*inherits BoundFunctor → Functor → Serializable*)
Creates/updates an *Aabb* of a *Facet*.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.Bo1_GridConnection_Aabb(*inherits BoundFunctor → Functor → Serializable*)
Functor creating *Aabb* from a *GridConnection*.

aabbEnlargeFactor(=*-1*, *deactivated*)
Relative enlargement of the bounding box; deactivated if negative.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.Bo1_Polyhedra_Aabb(*inherits BoundFunctor → Functor → Serializable*)
Create/update *Aabb* of a *Polyhedra*

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.Bo1_Sphere_Aabb(*inherits BoundFunctor → Functor → Serializable*)
Functor creating *Aabb* from *Sphere*.

aabbEnlargeFactor
Relative enlargement of the bounding box; deactivated if negative.

Note: This attribute is used to create distant interaction, but is only meaningful with an *IGeomFunctor* which will not simply discard such interactions: *Ig2_Sphere_Sphere_ScGeom::interactionDetectionFactor* should have the same value as *aabbEnlargeFactor*.

---

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Bo1_Tetra_Aabb**(*inherits BoundFunctor → Functor → Serializable*)
Create/update *Aabb* of a *Tetra*

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Bo1_Wall_Aabb**(*inherits BoundFunctor → Functor → Serializable*)
Creates/updates an *Aabb* of a *Wall*

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

8.5.2 BoundDispatcher

class yade.wrapper.**BoundDispatcher**(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling *functors* based on received argument type(s).

activated(=*true*)
Whether the engine is activated (only should be changed by the collider)

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

dispFunctor(*(Shape)arg2*) → BoundFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix([*(bool)names=True*]) → dict
Return dictionary with contents of the dispatch matrix.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

functors
Functors associated with this dispatcher.

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

minSweepDistFactor(=*0.2*)
Minimal distance by which enlarge all bounding boxes; superseeds computed value of sweepDist when lower that (minSweepDistFactor x sweepDist). Updated by the collider. *(auto-updated)*.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

sweepDist(=*0*)
Distance by which enlarge all bounding boxes, to prevent collider from being run at every step (only should be changed by the collider).

targetInterv(=*-1*)
see *InsertionSortCollider::targetInterv* *(auto-updated)*

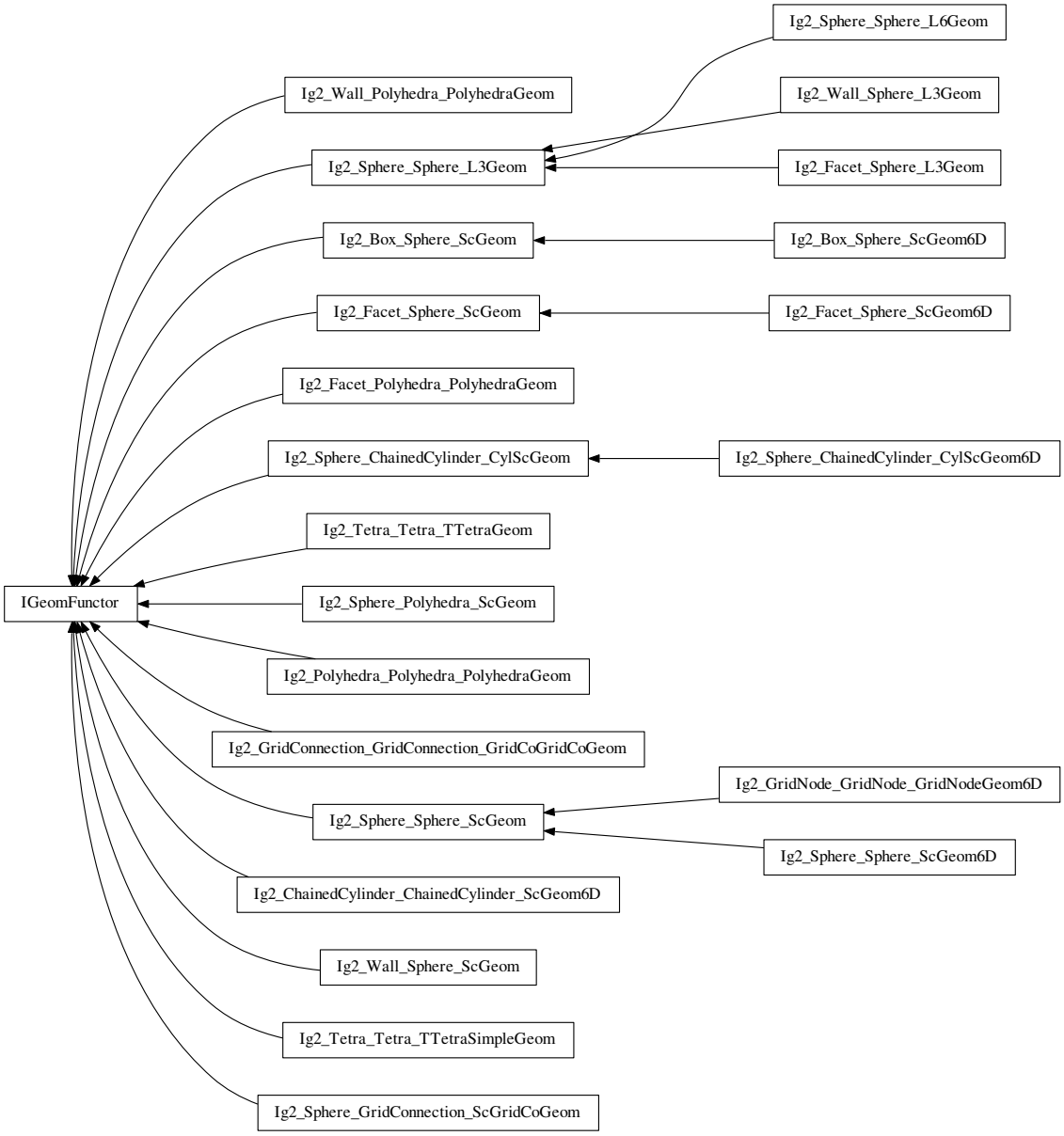
timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updatingDispFactor(=*-1*)
see *InsertionSortCollider::updatingDispFactor* *(auto-updated)*

8.6 Interaction Geometry creation

8.6.1 IGeomFunctor



class `yade.wrapper.IGeomFunctor`(*inherits Functor → Serializable*)

Functor for creating/updating *Interaction::geom* objects.

`bases`
: Ordered list of types (as strings) this functor accepts.

`dict()` → dict
: Return dictionary of attributes.

`label`(=*uninitalized*)
: Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
: Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the

source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Box_Sphere_ScGeom**(*inherits IGeomFunctor → Functor → Serializable*)
Create an interaction geometry *ScGeom* from *Box* and *Sphere*, representing the box with a projected virtual sphere of same radius.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Box_Sphere_ScGeom6D**(*inherits Ig2_Box_Sphere_ScGeom → IGeomFunctor → Functor → Serializable*)
Create an interaction geometry *ScGeom6D* from *Box* and *Sphere*, representing the box with a projected virtual sphere of same radius.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_ChainedCylinder_ChainedCylinder_ScGeom6D**(*inherits IGeomFunctor → Functor → Serializable*)
Create/update a *ScGeom* instance representing connexion between *chained cylinders*.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

halfLengthContacts(=*true*)
If True, Cylinders nodes interact like spheres of radius 0.5*length, else one node has size length while the other has size 0. The difference is mainly the locus of rotation definition.

interactionDetectionFactor(=*1*)
Enlarge both radii by this factor (if >1), to permit creation of distant interactions.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Facet_Polyhedra_PolyhedraGeom**(*inherits IGeomFunctor → Functor → Serializable*)
Create/update geometry of collision between Facet and Polyhedra

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Facet_Sphere_L3Geom**(*inherits Ig2_Sphere_Sphere_L3Geom → IGeomFunctor → Functor → Serializable*)
Incrementally compute *L3Geom* for contact between *Facet* and *Sphere*. Uses attributes of *Ig2_Sphere_Sphere_L3Geom*.

approxMask
Selectively enable geometrical approximations (bitmask); add the values for approximations to be enabled.

1	use previous transformation to transform velocities (which are known at mid-steps), instead of mid-step transformation computed as quaternion slerp at t=0.5.
2	do not take average (mid-step) normal when computing relative shear displacement, use previous value instead
4	do not re-normalize average (mid-step) normal, if used....

By default, the mask is zero, wherefore none of these approximations is used.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

distFactor(=*1*)
Create interaction if spheres are not futher than *distFactor* *(r1+r2). If negative, zero normal deformation will be set to be the initial value (otherwise, the geometrical distance is the ''zero" one).

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

noRatch(=*true*)
See *Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting*.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

trsfRenorm(=*100*)
How often to renormalize *trsf*; if non-positive, never renormalized (simulation might be unstable)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Facet_Sphere_ScGeom**(*inherits IGeomFunctor → Functor → Serializable*)
Create/update a *ScGeom* instance representing intersection of *Facet* and *Sphere*.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

shrinkFactor(=*0*, *no shrinking*)
The radius of the inscribed circle of the facet is decreased by the value of the sphere's radius multipled by *shrinkFactor*. From the definition of contact point on the surface made of facets, the given surface is not continuous and becomes in effect surface covered with triangular tiles, with gap between the separate tiles equal to the sphere's radius multiplied by 2×*shrinkFactor*. If zero, no shrinking is done.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Facet_Sphere_ScGeom6D**(*inherits Ig2_Facet_Sphere_ScGeom → IGeomFunctor → Functor → Serializable*)
Create an interaction geometry *ScGeom6D* from *Facet* and *Sphere*, representing the Facet with a projected virtual sphere of same radius.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

shrinkFactor(=*0*, *no shrinking*)
The radius of the inscribed circle of the facet is decreased by the value of the sphere's radius multipled by *shrinkFactor*. From the definition of contact point on the surface made of facets, the given surface is not continuous and becomes in effect surface covered with triangular tiles, with gap between the separate tiles equal to the sphere's radius multiplied by 2×*shrinkFactor*. If zero, no shrinking is done.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_GridConnection_GridConnection_GridCoGridCoGeom**(*inherits IGeomFunctor → Functor → Serializable*)

Create/update a *GridCoGridCoGeom* instance representing the geometry of a contact point between two *GridConnection* , including relative rotations.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_GridNode_GridNode_GridNodeGeom6D**(*inherits Ig2_Sphere_Sphere_ScGeom → IGeomFunctor → Functor → Serializable*)

Create/update a *GridNodeGeom6D* instance representing the geometry of a contact point between two *GridNode*, including relative rotations.

avoidGranularRatcheting
Define relative velocity so that ratcheting is avoided. It applies for sphere-sphere contacts. It eventualy also apply for sphere-emulating interactions (i.e. convertible into the ScGeom type), if the virtual sphere's motion is defined correctly (see e.g. *Ig2_Sphere_ChainedCylinder_CylScGeom*.

Short explanation of what we want to avoid :

Numerical ratcheting is best understood considering a small elastic cycle at a contact between two grains : assuming b1 is fixed, impose this displacement to b2 :

1.translation *dx* in the normal direction

2.rotation *a*

3.translation -*dx* (back to the initial position)

4.rotation -*a* (back to the initial orientation)

If the branch vector used to define the relative shear in rotation×branch is not constant (typically if it is defined from the vector center→contactPoint), then the shear displacement at the end of this cycle is not zero: rotations *a* and -*a* are multiplied by branches of different lengths.

It results in a finite contact force at the end of the cycle even though the positions and orientations are unchanged, in total contradiction with the elastic nature of the problem. It could also be seen as an *inconsistent energy creation or loss*. Given that DEM simulations tend to generate oscillations around equilibrium (damped mass-spring), it can have a significant impact on the evolution of the packings, resulting for instance in slow creep in iterations under constant load.

The solution adopted here to avoid ratcheting is as proposed by McNamara and co-workers. They analyzed the ratcheting problem in detail - even though they comment on the basis of a cycle that differs from the one shown above. One will find interesting discussions in e.g. *[McNamara2008]*, even though solution it suggests is not fully applied here (equations of motion are not incorporating alpha, in contradiction with what is suggested by McNamara et al.).

bases
Ordered list of types (as strings) this functor accepts.

creep(=*false*)
Substract rotational creep from relative rotation. The rotational creep *ScGeom6D::twistCreep* is a quaternion and has to be updated inside a constitutive law, see for instance *Law2_ScGeom6D_CohFrictPhys_CohesionMoment*.

dict() → dict
Return dictionary of attributes.

interactionDetectionFactor
Enlarge both radii by this factor (if >1), to permit creation of distant interactions.

InteractionGeometry will be computed when interactionDetectionFactor*(rad1+rad2) > distance.

Note: This parameter is functionally coupled with *Bo1_Sphere_Aabb::aabbEnlargeFactor*, which will create larger bounding boxes and should be of the same value.

---

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updateRotations(=*true*)
Precompute relative rotations. Turning this false can speed up simulations when rotations are not needed in constitutive laws (e.g. when spheres are compressed without cohesion and moment in early stage of a triaxial test), but is not foolproof. Change this value only if you know what you are doing.

class yade.wrapper.Ig2_Polyhedra_Polyhedra_PolyhedraGeom(*inherits IGeomFunctor → Functor → Serializable*)
Create/update geometry of collision between 2 Polyhedras

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.Ig2_Sphere_ChainedCylinder_CylScGeom(*inherits IGeomFunctor → Functor → Serializable*)
Create/update a *ScGeom* instance representing intersection of two *Spheres*.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

interactionDetectionFactor(=*1*)
Enlarge both radii by this factor (if >1), to permit creation of distant interactions.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Sphere_ChainedCylinder_CylScGeom6D**(*inherits Ig2_Sphere_ChainedCylinder_CylScGeom → IGeomFunctor → Functor → Serializable*)
Create/update a *ScGeom6D* instance representing the geometry of a contact point between two *Spheres*, including relative rotations.

bases
Ordered list of types (as strings) this functor accepts.

creep(=*false*)
Substract rotational creep from relative rotation. The rotational creep *ScGeom6D::twistCreep* is a quaternion and has to be updated inside a constitutive law, see for instance *Law2_ScGeom6D_CohFrictPhys_CohesionMoment*.

dict() → dict
Return dictionary of attributes.

interactionDetectionFactor(=*1*)
Enlarge both radii by this factor (if >1), to permit creation of distant interactions.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

updateRotations(=*false*)
Precompute relative rotations. Turning this false can speed up simulations when rotations are not needed in constitutive laws (e.g. when spheres are compressed without cohesion and moment in early stage of a triaxial test), but is not foolproof. Change this value only if you know what you are doing.

class yade.wrapper.**Ig2_Sphere_GridConnection_ScGridCoGeom**(*inherits IGeomFunctor → Functor → Serializable*)
Create/update a *ScGridCoGeom6D* instance representing the geometry of a contact point between a *GricConnection* and a *Sphere* including relative rotations.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

interactionDetectionFactor(=*1*)
Enlarge both radii by this factor (if >1), to permit creation of distant interactions.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Sphere_Polyhedra_ScGeom**(*inherits IGeomFunctor → Functor → Serializable*)
Create/update geometry of collision between Sphere and Polyhedra

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

edgeCoeff(*=1.0*)
multiplier of penetrationDepth when sphere contacts edge (simulating smaller volume of actual intersection or when several polyhedrons has common edge)

label(*=uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

vertexCoeff(*=1.0*)
multiplier of penetrationDepth when sphere contacts vertex (simulating smaller volume of actual intersection or when several polyhedrons has common vertex)

class yade.wrapper.**Ig2_Sphere_Sphere_L3Geom**(*inherits IGeomFunctor → Functor → Serializable*)
Functor for computing incrementally configuration of 2 *Spheres* stored in *L3Geom*; the configuration is positioned in global space by local origin $\mathbf{c}$ (contact point) and rotation matrix $\mathbf{T}$ (orthonormal transformation matrix), and its degrees of freedom are local displacement $\mathbf{u}$ (in one normal and two shear directions); with *Ig2_Sphere_Sphere_L6Geom* and *L6Geom*, there is additionally $\boldsymbol{\varphi}$. The first row of $\mathbf{T}$, i.e. local x-axis, is the contact normal noted $\mathbf{n}$ for brevity. Additionally, quasi-constant values of $\mathbf{u}_0$ (and $\boldsymbol{\varphi}_0$) are stored as shifted origins of $\mathbf{u}$ (and $\boldsymbol{\varphi}$); therefore, current value of displacement is always $\mathbf{u}^\circ - \mathbf{u}_0$.

Suppose two spheres with radii r_i, positions $\mathbf{x}_i$, velocities $\mathbf{v}_i$, angular velocities $\boldsymbol{\omega}_i$.

When there is not yet contact, it will be created if $u_N = |\mathbf{x}_2^\circ - \mathbf{x}_1^\circ| - |f_d|(r_1 + r2) < 0$, where f_d is *distFactor* (sometimes also called "interaction radius"). If $f_d > 0$, then $\mathbf{u}_{0x}$ will be initalized to u_N, otherwise to 0. In another words, contact will be created if spheres enlarged by $|f_d|$ touch, and the "equilibrium distance" (where $\mathbf{u}_x - \mathbf{u} - 0x$ is zero) will be set to the current distance if f_d is positive, and to the geometrically-touching distance if negative.

Local axes (rows of $\mathbf{T}$) are initially defined as follows:

- local x-axis is $\mathbf{n} = \mathbf{x}_l = \widehat{\mathbf{x}_2 - \mathbf{x}_1}$;
- local y-axis positioned arbitrarily, but in a deterministic manner: aligned with the xz plane (if $\mathbf{n}_y < \mathbf{n}_z$) or xy plane (otherwise);
- local z-axis $\mathbf{z}_l = \mathbf{x}_l \times \mathbf{y}_l$.

If there has already been contact between the two spheres, it is updated to keep track of rigid motion of the contact (one that does not change mutual configuration of spheres) and mutual configuration changes. Rigid motion transforms local coordinate system and can be decomposed in rigid translation (affecting $\mathbf{c}$), and rigid rotation (affecting $\mathbf{T}$), which can be split in rotation $\mathbf{o}_r$

perpendicular to the normal and rotation $\mathbf{o}_t$ (“twist”) parallel with the normal:

$$\mathbf{o}_r^{\ominus} = \mathbf{n}^- \times \mathbf{n}^\circ.$$

Since velocities are known at previous midstep $(t - \Delta t/2)$, we consider mid-step normal

$$\mathbf{n}^{\ominus} = \frac{\mathbf{n}^- + \mathbf{n}^\circ}{2}.$$

For the sake of numerical stability, $\mathbf{n}^{\ominus}$ is re-normalized after being computed, unless prohibited by *approxMask*. If *approxMask* has the appropriate bit set, the mid-normal is not compute, and we simply use $\mathbf{n}^{\ominus} \approx \mathbf{n}^-$.

Rigid rotation parallel with the normal is

$$\mathbf{o}_t^{\ominus} = \mathbf{n}^{\ominus} \left(\mathbf{n}^{\ominus} \cdot \frac{\boldsymbol{\omega}_1^{\ominus} + \boldsymbol{\omega}_2^{\ominus}}{2} \right) \Delta t.$$

Branch vectors $\mathbf{b}_1$, $\mathbf{b}_2$ (connecting $\mathbf{x}_1^\circ$, $\mathbf{x}_2^\circ$ with $\mathbf{c}^\circ$ are computed depending on *noRatch* (see *here*).

$$\mathbf{b}_1 = \begin{cases} r_1 \mathbf{n}^\circ & \text{with } \texttt{noRatch} \\ \mathbf{c}^\circ - \mathbf{x}_1^\circ & \text{otherwise} \end{cases}$$

$$\mathbf{b}_2 = \begin{cases} -r_2 \mathbf{n}^\circ & \text{with } \texttt{noRatch} \\ \mathbf{c}^\circ - \mathbf{x}_2^\circ & \text{otherwise} \end{cases}$$

Relative velocity at $\mathbf{c}^\circ$ can be computed as

$$\boldsymbol{v}_r^{\ominus} = (\tilde{\boldsymbol{v}}_2^{\ominus} + \boldsymbol{\omega}_2 \times \mathbf{b}_2) - (\boldsymbol{v}_1 + \boldsymbol{\omega}_1 \times \mathbf{b}_1)$$

where $\tilde{\boldsymbol{v}}_2$ is $\boldsymbol{v}_2$ without mean-field velocity gradient in periodic boundary conditions (see *Cell.homoDeform*). In the numerial implementation, the normal part of incident velocity is removed (since it is computed directly) with $\boldsymbol{v}_{r2}^{\ominus} = \boldsymbol{v}_r^{\ominus} - (\mathbf{n}^{\ominus} \cdot \boldsymbol{v}_r^{\ominus})\mathbf{n}^{\ominus}$.

Any vector $\mathbf{a}$ expressed in global coordinates transforms during one timestep as

$$\mathbf{a}^\circ = \mathbf{a}^- + \boldsymbol{v}_r^{\ominus} \Delta t - \mathbf{a}^- \times \mathbf{o}_r^{\ominus} - \mathbf{a}^- \times \mathbf{t}_r^{\ominus}$$

where the increments have the meaning of relative shear, rigid rotation normal to $\mathbf{n}$ and rigid rotation parallel with $\mathbf{n}$. Local coordinate system orientation, rotation matrix $\mathbf{T}$, is updated by rows, i.e.

$$\mathbf{T}^\circ = \begin{pmatrix} \mathbf{n}_x^\circ & \mathbf{n}_y^\circ & \mathbf{n}_z^\circ \\ \mathbf{T}_{1,\bullet}^- - \mathbf{T}_{1,\bullet}^- \times \mathbf{o}_r^{\ominus} - \mathbf{T}_{1,\bullet}^- \times \mathbf{o}_t^{\ominus} \\ \mathbf{T}_{2,\bullet}^- - \mathbf{T}_{2,\bullet}^- \times \mathbf{o}_r^{\ominus} - \mathbf{T}_{\bullet}^- \times \mathbf{o}_t^{\ominus} \end{pmatrix}$$

This matrix is re-normalized (unless prevented by *approxMask*) and mid-step transformation is computed using quaternion spherical interpolation as

$$\mathbf{T}^{\ominus} = \mathrm{Slerp}\left(\mathbf{T}^-; \mathbf{T}^\circ; t = 1/2\right).$$

Depending on *approxMask*, this computation can be avoided by approximating $\mathbf{T}^{\ominus} = \mathbf{T}^-$.

Finally, current displacement is evaluated as

$$\mathbf{u}^\circ = u^- + \mathbf{T}^{\ominus} \boldsymbol{v}_r^{\ominus} \Delta t.$$

For the normal component, non-incremental evaluation is preferred, giving

$$\mathbf{u}_x^\circ = |\mathbf{x}_2^\circ - \mathbf{x}_1^\circ| - (r_1 + r_2)$$

If this functor is called for *L6Geom*, local rotation is updated as

$$\boldsymbol{\varphi}^\circ = \boldsymbol{\varphi}^- + \mathbf{T}^{\ominus} \Delta t (\boldsymbol{\omega}_2 - \boldsymbol{\omega}_1)$$

approxMask
Selectively enable geometrical approximations (bitmask); add the values for approximations to be enabled.

1	use previous transformation to transform velocities (which are known at mid-steps), instead of mid-step transformation computed as quaternion slerp at t=0.5.
2	do not take average (mid-step) normal when computing relative shear displacement, use previous value instead
4	do not re-normalize average (mid-step) normal, if used....

By default, the mask is zero, wherefore none of these approximations is used.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

distFactor(=*1*)
Create interaction if spheres are not futher than *distFactor* *(r1+r2). If negative, zero normal deformation will be set to be the initial value (otherwise, the geometrical distance is the ''zero" one).

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

noRatch(=*true*)
See *Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting*.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

trsfRenorm(=*100*)
How often to renormalize *trsf*; if non-positive, never renormalized (simulation might be unstable)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Sphere_Sphere_L6Geom**(*inherits Ig2_Sphere_Sphere_L3Geom → IGeomFunctor → Functor → Serializable*)
Incrementally compute *L6Geom* for contact of 2 spheres.

approxMask
Selectively enable geometrical approximations (bitmask); add the values for approximations to be enabled.

1	use previous transformation to transform velocities (which are known at mid-steps), instead of mid-step transformation computed as quaternion slerp at t=0.5.
2	do not take average (mid-step) normal when computing relative shear displacement, use previous value instead
4	do not re-normalize average (mid-step) normal, if used....

By default, the mask is zero, wherefore none of these approximations is used.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

distFactor(=*1*)
Create interaction if spheres are not futher than *distFactor* *(r1+r2). If negative, zero normal deformation will be set to be the initial value (otherwise, the geometrical distance is the ''zero" one).

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

noRatch(=*true*)
See *Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting*.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

trsfRenorm(=*100*)
How often to renormalize *trsf*; if non-positive, never renormalized (simulation might be unstable)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Sphere_Sphere_ScGeom**(*inherits IGeomFunctor → Functor → Serializable*)
Create/update a *ScGeom* instance representing the geometry of a contact point between two *Spheres* s.

avoidGranularRatcheting
Define relative velocity so that ratcheting is avoided. It applies for sphere-sphere contacts. It eventualy also apply for sphere-emulating interactions (i.e. convertible into the ScGeom type), if the virtual sphere's motion is defined correctly (see e.g. *Ig2_Sphere_ChainedCylinder_CylScGeom*.

Short explanation of what we want to avoid :

Numerical ratcheting is best understood considering a small elastic cycle at a contact between two grains : assuming b1 is fixed, impose this displacement to b2 :

1.translation *dx* in the normal direction

2.rotation *a*

3.translation *-dx* (back to the initial position)

4.rotation *-a* (back to the initial orientation)

If the branch vector used to define the relative shear in rotation×branch is not constant (typically if it is defined from the vector center→contactPoint), then the shear displacement at the end of this cycle is not zero: rotations *a* and *-a* are multiplied by branches of different lengths.

It results in a finite contact force at the end of the cycle even though the positions and orientations are unchanged, in total contradiction with the elastic nature of the problem. It could also be seen as an *inconsistent energy creation or loss*. Given that DEM simulations tend to generate oscillations around equilibrium (damped mass-spring), it can have a significant impact on the evolution of the packings, resulting for instance in slow creep in iterations under constant load.

The solution adopted here to avoid ratcheting is as proposed by McNamara and co-workers. They analyzed the ratcheting problem in detail - even though they comment on the basis of a cycle that differs from the one shown above. One will find interesting discussions in e.g. *[McNamara2008]*, even though solution it suggests is not fully applied here (equations of motion are not incorporating alpha, in contradiction with what is suggested by McNamara et al.).

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

interactionDetectionFactor
Enlarge both radii by this factor (if >1), to permit creation of distant interactions.

InteractionGeometry will be computed when interactionDetectionFactor*(rad1+rad2) > distance.

Note: This parameter is functionally coupled with *Bo1_Sphere_Aabb::aabbEnlargeFactor*, which will create larger bounding boxes and should be of the same value.

---

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Sphere_Sphere_ScGeom6D**(*inherits* *Ig2_Sphere_Sphere_ScGeom* → *IGeomFunctor* → *Functor* → *Serializable*)
Create/update a *ScGeom6D* instance representing the geometry of a contact point between two *Spheres*, including relative rotations.

avoidGranularRatcheting
Define relative velocity so that ratcheting is avoided. It applies for sphere-sphere contacts. It eventualy also apply for sphere-emulating interactions (i.e. convertible into the ScGeom type), if the virtual sphere's motion is defined correctly (see e.g. *Ig2_Sphere_ChainedCylinder_CylScGeom*.

Short explanation of what we want to avoid :

Numerical ratcheting is best understood considering a small elastic cycle at a contact between two grains : assuming b1 is fixed, impose this displacement to b2 :

1.translation *dx* in the normal direction

2.rotation *a*

3.translation -*dx* (back to the initial position)

4.rotation -*a* (back to the initial orientation)

If the branch vector used to define the relative shear in rotation×branch is not constant (typically if it is defined from the vector center→contactPoint), then the shear displacement at the end of this cycle is not zero: rotations *a* and -*a* are multiplied by branches of different lengths.

It results in a finite contact force at the end of the cycle even though the positions and orientations are unchanged, in total contradiction with the elastic nature of the problem. It could also be seen as an *inconsistent energy creation or loss*. Given that DEM simulations tend to generate oscillations around equilibrium (damped mass-spring), it can have a significant impact on the evolution of the packings, resulting for instance in slow creep in iterations under constant load.

The solution adopted here to avoid ratcheting is as proposed by McNamara and co-workers. They analyzed the ratcheting problem in detail - even though they comment on the basis of a cycle that differs from the one shown above. One will find interesting discussions in e.g. *[McNamara2008]*, even though solution it suggests is not fully applied here (equations of motion are not incorporating alpha, in contradiction with what is suggested by McNamara et al.).

bases
Ordered list of types (as strings) this functor accepts.

`creep`(=*false*)
Substract rotational creep from relative rotation. The rotational creep *ScGeom6D::twistCreep* is a quaternion and has to be updated inside a constitutive law, see for instance *Law2_ScGeom6D_CohFrictPhys_CohesionMoment*.

`dict`() → dict
Return dictionary of attributes.

`interactionDetectionFactor`
Enlarge both radii by this factor (if >1), to permit creation of distant interactions.

InteractionGeometry will be computed when interactionDetectionFactor*(rad1+rad2) > distance.

Note: This parameter is functionally coupled with *Bo1_Sphere_Aabb::aabbEnlargeFactor*, which will create larger bounding boxes and should be of the same value.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`updateRotations`(=*true*)
Precompute relative rotations. Turning this false can speed up simulations when rotations are not needed in constitutive laws (e.g. when spheres are compressed without cohesion and moment in early stage of a triaxial test), but is not foolproof. Change this value only if you know what you are doing.

class `yade.wrapper.Ig2_Tetra_Tetra_TTetraGeom`(*inherits IGeomFunctor → Functor → Serializable*)
Create/update geometry of collision between 2 *tetrahedra* (*TTetraGeom* instance)

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ig2_Tetra_Tetra_TTetraSimpleGeom`(*inherits IGeomFunctor → Functor → Serializable*)
EXPERIMANTAL. Create/update geometry of collision between 2 *tetrahedra* (*TTetraSimpleGeom* instance)

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Wall_Polyhedra_PolyhedraGeom**(*inherits IGeomFunctor → Functor → Serializable*)
Create/update geometry of collision between Wall and Polyhedra

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Wall_Sphere_L3Geom**(*inherits Ig2_Sphere_Sphere_L3Geom → IGeomFunctor → Functor → Serializable*)
Incrementally compute *L3Geom* for contact between *Wall* and *Sphere*. Uses attributes of *Ig2_Sphere_Sphere_L3Geom*.

approxMask
Selectively enable geometrical approximations (bitmask); add the values for approximations to be enabled.

1	use previous transformation to transform velocities (which are known at mid-steps), instead of mid-step transformation computed as quaternion slerp at t=0.5.
2	do not take average (mid-step) normal when computing relative shear displacement, use previous value instead
4	do not re-normalize average (mid-step) normal, if used....

By default, the mask is zero, wherefore none of these approximations is used.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

distFactor(=*1*)
Create interaction if spheres are not futher than *distFactor* *(r1+r2). If negative, zero normal deformation will be set to be the initial value (otherwise, the geometrical distance is the ''zero" one).

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

noRatch(=*true*)
See *Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting*.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

trsfRenorm(*=100*)
How often to renormalize *trsf*; if non-positive, never renormalized (simulation might be unstable)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ig2_Wall_Sphere_ScGeom**(*inherits IGeomFunctor → Functor → Serializable*)
Create/update a *ScGeom* instance representing intersection of *Wall* and *Sphere*.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(*=uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

noRatch(*=true*)
Avoid granular ratcheting

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

8.6.2 IGeomDispatcher

class yade.wrapper.**IGeomDispatcher**(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling *functors* based on received argument type(s).

dead(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

dispFunctor(*(Shape)arg2*, *(Shape)arg3*) → IGeomFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix([*(bool)names=True*]) → dict
Return dictionary with contents of the dispatch matrix.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==True).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==True).

functors
Functors associated with this dispatcher.

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

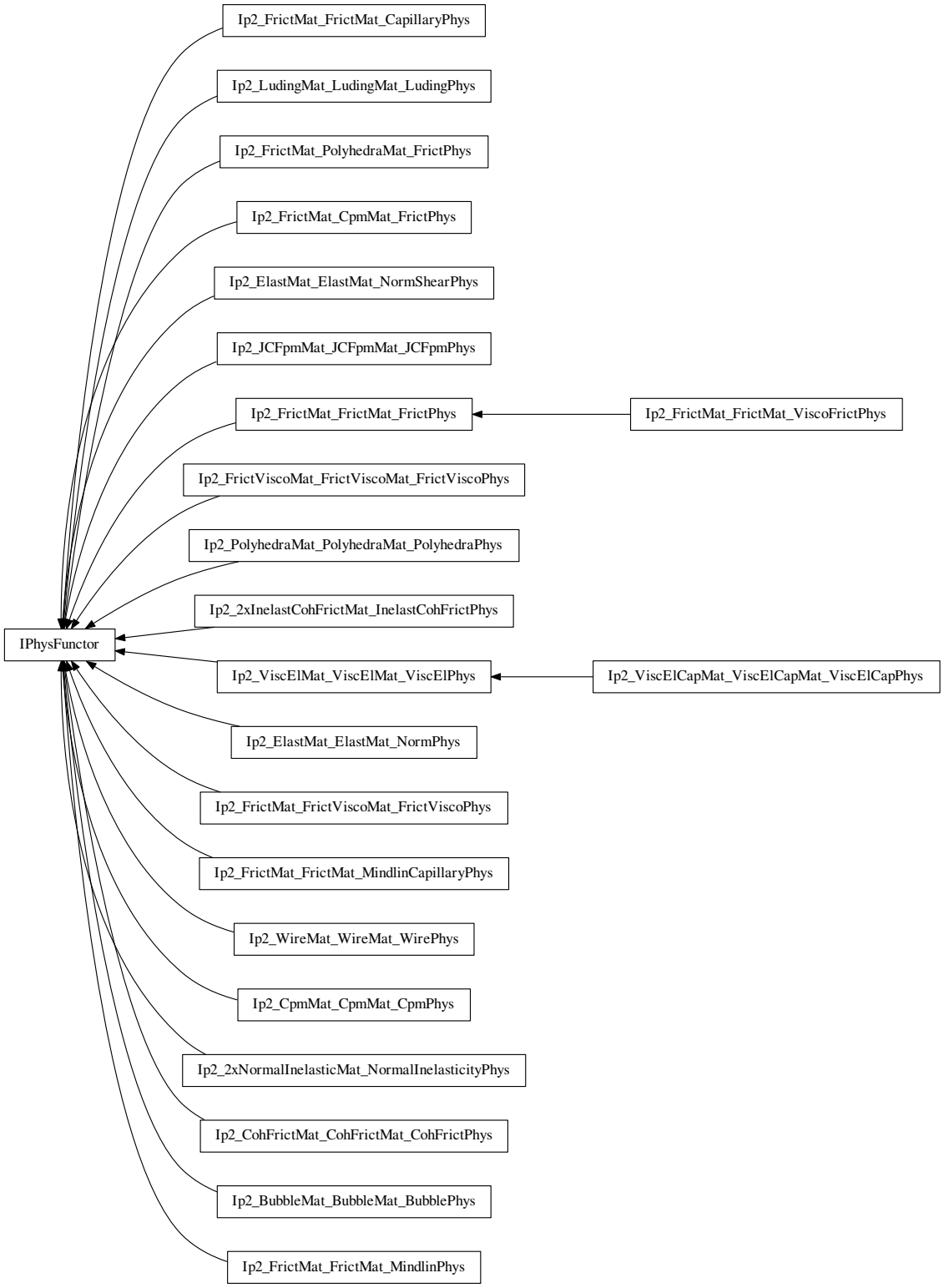
ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

8.7 Interaction Physics creation

8.7.1 IPhysFunctor



class `yade.wrapper.IPhysFunctor`(*inherits* *Functor* → *Serializable*)

Functor for creating/updating *Interaction::phys* objects.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_2xInelastCohFrictMat_InelastCohFrictPhys**(*inherits IPhysFunctor → Functor → Serializable*)
Generates cohesive-frictional interactions with moments. Used in the contact law *Law2_ScGeom6D_InelastCohFrictPhys_CohesionMoment*.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_2xNormalInelasticMat_NormalInelasticityPhys**(*inherits IPhysFunctor → Functor → Serializable*)
Computes interaction attributes (of *NormalInelasticityPhys* type) from *NormalInelasticMat* material parameters. For simulations using *Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity*. Note that, as for others *Ip2 functors*, most of the attributes are computed only once, when the interaction is new.

bases
Ordered list of types (as strings) this functor accepts.

betaR(=*0.12*)
Parameter for computing the torque-stifness : T-stifness = betaR * Rmoy^2

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.Ip2_BubbleMat_BubbleMat_BubblePhys(*inherits IPhysFunctor → Functor → Serializable*)

Generates bubble interactions.Used in the contact law Law2_ScGeom_BubblePhys_Bubble.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.Ip2_CohFrictMat_CohFrictMat_CohFrictPhys(*inherits IPhysFunctor → Functor → Serializable*)

Generates cohesive-frictional interactions with moments, used in the contact law *Law2_ScGeom6D_CohFrictPhys_CohesionMoment*. The normal/shear stiffness and friction definitions are the same as in *Ip2_FrictMat_FrictMat_FrictPhys*, check the documentation there for details.

Adhesions related to the normal and the shear components are calculated form *CohFrictMat::normalCohesion* (C_n) and *CohFrictMat::shearlCohesion* (C_s). For particles of size R_1,R_2 the adhesion will be $a_i = C_i \min(R_1, R_2)^2$, $i = n\,s$.

Twist and rolling stiffnesses are proportional to the shear stiffness through dimensionless factors alphaKtw and alphaKr, such that the rotational stiffnesses are defined by $k_s \alpha_i R_1 R_2$, $i = tw\,r$

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

setCohesionNow(=*false*)
If true, assign cohesion to all existing contacts in current time-step. The flag is turned false automatically, so that assignment is done in the current timestep only.

setCohesionOnNewContacts(=*false*)
If true, assign cohesion at all new contacts. If false, only existing contacts can be cohesive (also see *Ip2_CohFrictMat_CohFrictMat_CohFrictPhys::setCohesionNow*), and new contacts are only frictional.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.Ip2_CpmMat_CpmMat_CpmPhys(*inherits IPhysFunctor → Functor → Serializable*)

Convert 2 *CpmMat* instances to *CpmPhys* with corresponding parameters. Uses simple (arithmetic) averages if material are different. Simple copy of parameters is performed if the *material* is shared between both particles. See *cpm-model* for detals.

bases
Ordered list of types (as strings) this functor accepts.

cohesiveThresholdIter(*=10*)
Should new contacts be cohesive? They will before this iter#, they will not be afterwards. If 0, they will never be. If negative, they will always be created as cohesive (10 by default).

dict() → dict
Return dictionary of attributes.

label(*=uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_ElastMat_ElastMat_NormPhys**(*inherits IPhysFunctor → Functor → Serializable*)
Create a *NormPhys* from two *ElastMats*. TODO. EXPERIMENTAL

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(*=uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_ElastMat_ElastMat_NormShearPhys**(*inherits IPhysFunctor → Functor → Serializable*)
Create a *NormShearPhys* from two *ElastMats*. TODO. EXPERIMENTAL

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(*=uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_FrictMat_CpmMat_FrictPhys**(*inherits IPhysFunctor → Functor → Serializable*)
Convert *CpmMat* instance and *FrictMat* instance to *FrictPhys* with corresponding parameters (young, poisson, frictionAngle). Uses simple (arithmetic) averages if material parameters are different.

bases
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ip2_FrictMat_FrictMat_CapillaryPhys`(*inherits IPhysFunctor → Functor → Serializable*)
RelationShips to use with *Law2_ScGeom_CapillaryPhys_Capillarity*.

In these RelationShips all the interaction attributes are computed.

Warning: as in the others *Ip2 functors*, most of the attributes are computed only once, when the interaction is new.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ip2_FrictMat_FrictMat_FrictPhys`(*inherits IPhysFunctor → Functor → Serializable*)
Create a *FrictPhys* from two *FrictMats*. The compliance of one sphere under point load is defined here as $1/(E.D)$, with E the stiffness of the sphere and D its diameter. The compliance of the contact itself will be the sum of compliances from each sphere, i.e. $1/(E_1.D_1) + 1/(E_2.D_2)$ in the general case, or $2/(E.D)$ in the special case of equal sizes and equal stiffness. Note that summing compliances corresponds to an harmonic average of stiffnesss (as in e.g. *[Scholtes2009a]*), which is how kn is actually computed in the *Ip2_FrictMat_FrictMat_FrictPhys* functor:

$k_n = \frac{E_1 D_1 * E_2 D_2}{E_1 D_1 + E_2 D_2} = \frac{k_1 * k_2}{k_1 + k_2}$, with $k_i = E_i D_i$.

The shear stiffness ks of one sphere is defined via the material parameter *ElastMat::poisson*, as ks=poisson*kn, and the resulting shear stiffness of the interaction will be also an harmonic average. In the case of a contact between a *ViscElMat* and a *FrictMat*, be sure to set *FrictMat::young* and *FrictMat::poisson*, otherwise the default value will be used.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`frictAngle`(=*uninitalized*)
Instance of *MatchMaker* determining how to compute interaction's friction angle. If `None`, minimum value is used.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_FrictMat_FrictMat_MindlinCapillaryPhys**(*inherits IPhysFunctor → Functor → Serializable*)
RelationShips to use with Law2_ScGeom_CapillaryPhys_Capillarity

In these RelationShips all the interaction attributes are computed.

Warning: as in the others *Ip2 functors*, most of the attributes are computed only once, when the interaction is new.

bases
Ordered list of types (as strings) this functor accepts.

betan(=*uninitalized*)
Normal viscous damping ratio β_n.

betas(=*uninitalized*)
Shear viscous damping ratio β_s.

dict() → dict
Return dictionary of attributes.

en(=*uninitalized*)
Normal coefficient of restitution e_n.

es(=*uninitalized*)
Shear coefficient of restitution e_s.

eta(=*0.0*)
Coefficient to determine the plastic bending moment

gamma(=*0.0*)
Surface energy parameter [J/m^2] per each unit contact surface, to derive DMT formulation from HM

krot(=*0.0*)
Rotational stiffness for moment contact law

ktwist(=*0.0*)
Torsional stiffness for moment contact law

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_FrictMat_FrictMat_MindlinPhys**(*inherits IPhysFunctor → Functor → Serializable*)
Calculate some physical parameters needed to obtain the normal and shear stiffnesses according to the Hertz-Mindlin formulation (as implemented in PFC).

Viscous parameters can be specified either using coefficients of restitution (e_n, e_s) or viscous damping ratio (β_n, β_s). The following rules apply: #. If the β_n (β_s) ratio is given, it is assigned to

MindlinPhys.betan (*MindlinPhys.betas*) directly. #. If e_n is given, *MindlinPhys.betan* is computed using $\beta_n = -(\log e_n)/\sqrt{\pi^2 + (\log e_n)^2}$. The same applies to e_s, *MindlinPhys.betas*. #. It is an error (exception) to specify both e_n and β_n (e_s and β_s). #. If neither e_n nor β_n is given, zero value for *MindlinPhys.betan* is used; there will be no viscous effects. #.If neither e_s nor β_s is given, the value of *MindlinPhys.betan* is used for *MindlinPhys.betas* as well.

The e_n, β_n, e_s, β_s are *MatchMaker* objects; they can be constructed from float values to always return constant value.

See scripts/test/shots.py for an example of specifying e_n based on combination of parameters.

bases
Ordered list of types (as strings) this functor accepts.

betan(=*uninitalized*)
Normal viscous damping ratio β_n.

betas(=*uninitalized*)
Shear viscous damping ratio β_s.

dict() → dict
Return dictionary of attributes.

en(=*uninitalized*)
Normal coefficient of restitution e_n.

es(=*uninitalized*)
Shear coefficient of restitution e_s.

eta(=*0.0*)
Coefficient to determine the plastic bending moment

frictAngle(=*uninitalized*)
Instance of *MatchMaker* determining how to compute the friction angle of an interaction. If `None`, minimum value is used.

gamma(=*0.0*)
Surface energy parameter [J/m^2] per each unit contact surface, to derive DMT formulation from HM

krot(=*0.0*)
Rotational stiffness for moment contact law

ktwist(=*0.0*)
Torsional stiffness for moment contact law

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_FrictMat_FrictMat_ViscoFrictPhys**(*inherits Ip2_FrictMat_FrictMat_FrictPhys → IPhysFunctor → Functor → Serializable*)
Create a *FrictPhys* from two *FrictMats*. The compliance of one sphere under symetric point loads is defined here as 1/(E.r), with E the stiffness of the sphere and r its radius, and corresponds to a compliance 1/(2.E.r)=1/(E.D) from each contact point. The compliance of the contact itself will be the sum of compliances from each sphere, i.e. 1/(E.D1)+1/(E.D2) in the general case, or 1/(E.r) in the special case of equal sizes. Note that summing compliances corresponds to an harmonic average of stiffnesss, which is how kn is actually computed in the *Ip2_FrictMat_FrictMat_FrictPhys* functor.

The shear stiffness ks of one sphere is defined via the material parameter *ElastMat::poisson*, as ks=poisson*kn, and the resulting shear stiffness of the interaction will be also an harmonic average.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

frictAngle(=*uninitalized*)
Instance of *MatchMaker* determining how to compute interaction's friction angle. If `None`, minimum value is used.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_FrictMat_FrictViscoMat_FrictViscoPhys**(*inherits IPhysFunctor → Functor → Serializable*)
Converts a *FrictMat* and *FrictViscoMat* instance to *FrictViscoPhys* with corresponding parameters. Basically this functor corresponds to *Ip2_FrictMat_FrictMat_FrictPhys* with the only difference that damping in normal direction can be considered.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

frictAngle(=*uninitalized*)
Instance of *MatchMaker* determining how to compute interaction's friction angle. If `None`, minimum value is used.

kRatio(=*uninitalized*)
Instance of *MatchMaker* determining how to compute interaction's shear contact stiffnesses. If this value is not given the elastic properties (i.e. poisson) of the two colliding materials are used to calculate the stiffness.

kn(=*uninitalized*)
Instance of *MatchMaker* determining how to compute interaction's normal contact stiffnesses. If this value is not given the elastic properties (i.e. young) of the two colliding materials are used to calculate the stiffness.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_FrictMat_PolyhedraMat_FrictPhys**(*inherits IPhysFunctor → Functor → Serializable*)

bases
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ip2_FrictViscoMat_FrictViscoMat_FrictViscoPhys`(*inherits IPhysFunctor → Functor → Serializable*)

Converts 2 *FrictViscoMat* instances to *FrictViscoPhys* with corresponding parameters. Basically this functor corresponds to *Ip2_FrictMat_FrictMat_FrictPhys* with the only difference that damping in normal direction can be considered.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`frictAngle`(=*uninitalized*)
Instance of *MatchMaker* determining how to compute interaction's friction angle. If `None`, minimum value is used.

`kRatio`(=*uninitalized*)
Instance of *MatchMaker* determining how to compute interaction's shear contact stiffnesses. If this value is not given the elastic properties (i.e. poisson) of the two colliding materials are used to calculate the stiffness.

`kn`(=*uninitalized*)
Instance of *MatchMaker* determining how to compute interaction's normal contact stiffnesses. If this value is not given the elastic properties (i.e. young) of the two colliding materials are used to calculate the stiffness.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Ip2_JCFpmMat_JCFpmMat_JCFpmPhys`(*inherits IPhysFunctor → Functor → Serializable*)

Converts 2 *JCFpmMat* instances to one *JCFpmPhys* instance, with corresponding parameters.

`bases`
Ordered list of types (as strings) this functor accepts.

`cohesiveTresholdIteration`(=*1*)
should new contacts be cohesive? If strictly negativ, they will in any case. If positiv, they will before this iter, they won't afterward.

`dict()` → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_LudingMat_LudingMat_LudingPhys**(*inherits IPhysFunctor → Functor → Serializable*)
Convert 2 instances of *LudingMat* to *LudingPhys* using the rule of consecutive connection.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_PolyhedraMat_PolyhedraMat_PolyhedraPhys**(*inherits IPhysFunctor → Functor → Serializable*)

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_ViscElCapMat_ViscElCapMat_ViscElCapPhys**(*inherits Ip2_ViscElMat_ViscElMat_ViscElPhys → IPhysFunctor → Functor → Serializable*)
Convert 2 instances of *ViscElCapMat* to *ViscElCapPhys* using the rule of consecutive connection.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

en(=*uninitalized*)
Instance of *MatchMaker* determining restitution coefficient in normal direction

et(=*uninitalized*)
Instance of *MatchMaker* determining restitution coefficient in tangential direction

frictAngle(=*uninitalized*)
Instance of *MatchMaker* determining how to compute interaction's friction angle. If `None`, minimum value is used.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

tc(=*uninitalized*)
Instance of *MatchMaker* determining contact time

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_ViscElMat_ViscElMat_ViscElPhys**(*inherits IPhysFunctor → Functor → Serializable*)
Convert 2 instances of *ViscElMat* to *ViscElPhys* using the rule of consecutive connection.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

en(=*uninitalized*)
Instance of *MatchMaker* determining restitution coefficient in normal direction

et(=*uninitalized*)
Instance of *MatchMaker* determining restitution coefficient in tangential direction

frictAngle(=*uninitalized*)
Instance of *MatchMaker* determining how to compute interaction's friction angle. If `None`, minimum value is used.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

tc(=*uninitalized*)
Instance of *MatchMaker* determining contact time

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Ip2_WireMat_WireMat_WirePhys**(*inherits IPhysFunctor → Functor → Serializable*)
Converts 2 *WireMat* instances to *WirePhys* with corresponding parameters.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`linkThresholdIteration`(=*1*)
Iteration to create the link.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

8.7.2 IPhysDispatcher

class `yade.wrapper.IPhysDispatcher`(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling *functors* based on received argument type(s).

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`dispFunctor`(*(Material)arg2*, *(Material)arg3*) → IPhysFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

`dispMatrix`(*[(bool)names=True]*) → dict
Return dictionary with contents of the dispatch matrix.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`functors`
Functors associated with this dispatcher.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

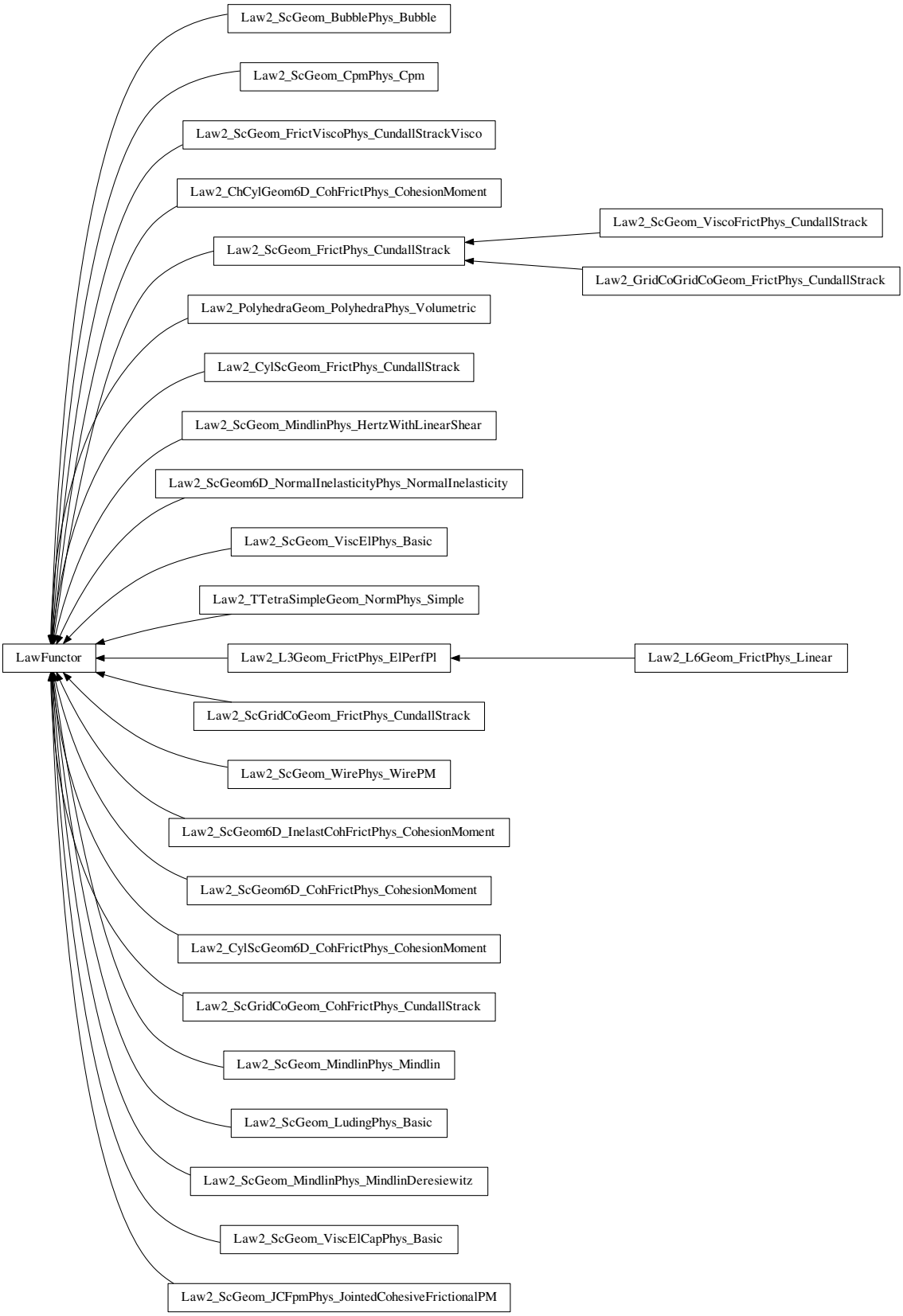
`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

8.8 Constitutive laws

8.8.1 LawFunctor



class `yade.wrapper.LawFunctor`(*inherits Functor → Serializable*)

Functor for applying constitutive laws on *interactions*.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ChCylGeom6D_CohFrictPhys_CohesionMoment**(*inherits LawFunctor → Functor → Serializable*)

Law for linear compression, and Mohr-Coulomb plasticity surface without cohesion. This law implements the classical linear elastic-plastic law from *[CundallStrack1979]* (see also *[Pfc3dManual30]*). The normal force is (with the convention of positive tensile forces) $F_n = \min(k_n u_n, 0)$. The shear force is $F_s = k_s u_s$, the plasticity condition defines the maximum value of the shear force : $F_s^{max} = F_n \tan(\varphi)$, with φ the friction angle.

Note: This law is well tested in the context of triaxial simulation, and has been used for a number of published results (see e.g. *[Scholtes2009b]* and other papers from the same authors). It is generalised by *Law2_ScGeom6D_CohFrictPhys_CohesionMoment*, which adds cohesion and moments at contact.

---

always_use_moment_law(=*false*)
If true, use bending/twisting moments at all contacts. If false, compute moments only for cohesive contacts.

bases
Ordered list of types (as strings) this functor accepts.

creep_viscosity(=*1*)
creep viscosity [Pa.s/m]. probably should be moved to Ip2_CohFrictMat_CohFrictMat_CohFrictPhys...

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

neverErase(=*false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. *Law2_ScGeom_CapillaryPhys_Capillarity*)

shear_creep(=*false*)
activate creep on the shear force, using *CohesiveFrictionalContactLaw::creep_viscosity*.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

twist_creep(=*false*)
activate creep on the twisting moment, using *CohesiveFrictionalContactLaw::creep_viscosity*.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

useIncrementalForm(=*false*)
use the incremental formulation to compute bending and twisting moments. Creep on the twisting moment is not included in such a case.

class yade.wrapper.**Law2_CylScGeom6D_CohFrictPhys_CohesionMoment**(*inherits LawFunctor → Functor → Serializable*)
This law generalises *Law2_CylScGeom_FrictPhys_CundallStrack* by adding cohesion and moments at contact.

always_use_moment_law(=*false*)
If true, use bending/twisting moments at all contacts. If false, compute moments only for cohesive contacts.

bases
Ordered list of types (as strings) this functor accepts.

creep_viscosity(=*1*)
creep viscosity [Pa.s/m]. probably should be moved to Ip2_CohFrictMat_CohFrictMat_-CohFrictPhys...

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

neverErase(=*false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. *Law2_ScGeom_CapillaryPhys_Capillarity*)

shear_creep(=*false*)
activate creep on the shear force, using *CohesiveFrictionalContactLaw::creep_viscosity*.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

twist_creep(=*false*)
activate creep on the twisting moment, using *CohesiveFrictionalContactLaw::creep_viscosity*.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

useIncrementalForm(=*false*)
use the incremental formulation to compute bending and twisting moments. Creep on the twisting moment is not included in such a case.

class yade.wrapper.**Law2_CylScGeom_FrictPhys_CundallStrack**(*inherits LawFunctor → Functor → Serializable*)
Law for linear compression, and Mohr-Coulomb plasticity surface without cohesion. This law implements the classical linear elastic-plastic law from *[CundallStrack1979]* (see also *[Pfc3dManual30]*). The normal force is (with the convention of positive tensile forces) $F_n = \min(k_n u_n, 0)$. The shear force is $F_s = k_s u_s$, the plasticity condition defines the maximum value of the shear force : $F_s^{max} = F_n \tan(\varphi)$, with φ the friction angle.

Note: This law uses *ScGeom*.

Note: This law is well tested in the context of triaxial simulation, and has been used for a number of published results (see e.g. *[Scholtes2009b]* and other papers from the same authors). It is generalised by *Law2_ScGeom6D_CohFrictPhys_CohesionMoment*, which adds cohesion and moments at contact.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

neverErase(=*false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. *Law2_ScGeom_CapillaryPhys_Capillarity*)

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_GridCoGridCoGeom_FrictPhys_CundallStrack**(*inherits Law2_ScGeom_FrictPhys_CundallStrack → LawFunctor → Functor → Serializable*)
Frictional elastic contact law between two *gridConnection* . See *Law2_ScGeom_FrictPhys_CundallStrack* for more details.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

elasticEnergy() → float
Compute and return the total elastic energy in all "FrictPhys" contacts

initPlasticDissipation(*(float)arg2*) → None
Initialize cummulated plastic dissipation to a value (0 by default).

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

neverErase(=*false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. *Law2_ScGeom_CapillaryPhys_Capillarity*)

plasticDissipation() → float
Total energy dissipated in plastic slips at all FrictPhys contacts. Computed only if *Law2_ScGeom_FrictPhys_CundallStrack::traceEnergy* is true.

sphericalBodies(=*true*)
If true, compute branch vectors from radii (faster), else use contactPoint-position. Turning this flag true is safe for sphere-sphere contacts and a few other specific cases. It will give wrong values of torques on facets or boxes.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

traceEnergy(=*false*)
Define the total energy dissipated in plastic slips at all contacts. This will trace only plastic energy in this law, see O.trackEnergy for a more complete energies tracing

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_L3Geom_FrictPhys_ElPerfPl**(*inherits LawFunctor → Functor → Serializable*)
Basic law for testing *L3Geom*; it bears no cohesion (unless *noBreak* is `True`), and plastic slip obeys the Mohr-Coulomb criterion (unless *noSlip* is `True`).

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(*=uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

noBreak(*=false*)
Do not break contacts when particles separate.

noSlip(*=false*)
No plastic slipping.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_L6Geom_FrictPhys_Linear**(*inherits Law2_L3Geom_FrictPhys_ElPerfPl → LawFunctor → Functor → Serializable*)
Basic law for testing *L6Geom* – linear in both normal and shear sense, without slip or breakage.

bases
Ordered list of types (as strings) this functor accepts.

charLen(*=1*)
Characteristic length with the meaning of the stiffness ratios bending/shear and torsion/normal.

dict() → dict
Return dictionary of attributes.

label(*=uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

noBreak(*=false*)
Do not break contacts when particles separate.

noSlip(*=false*)
No plastic slipping.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_PolyhedraGeom_PolyhedraPhys_Volumetric**(*inherits LawFunctor → Functor → Serializable*)
Calculate physical response of 2 *vector* in interaction, based on penetration configuration given by *PolyhedraGeom*. Normal force is proportional to the volume of intersection

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

elasticEnergy() → float
Compute and return the total elastic energy in all "FrictPhys" contacts

initPlasticDissipation(*(float)arg2*) → None
Initialize cummulated plastic dissipation to a value (0 by default).

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

plasticDissipation() → float
Total energy dissipated in plastic slips at all FrictPhys contacts. Computed only if *Law2_PolyhedraGeom_PolyhedraPhys_Volumetric::traceEnergy* is true.

shearForce(=*Vector3r::Zero()*)
Shear force from last step

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

traceEnergy(=*false*)
Define the total energy dissipated in plastic slips at all contacts. This will trace only plastic energy in this law, see O.trackEnergy for a more complete energies tracing

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

volumePower(=*1.*)
Power of volume used in evaluation of normal force. Default is 1.0 - normal force is linearly proportional to volume. 1.0/3.0 would mean that normal force is proportional to the cube root of volume, approximation of penetration depth.

class yade.wrapper.**Law2_ScGeom6D_CohFrictPhys_CohesionMoment**(*inherits LawFunctor → Functor → Serializable*)

Law for linear traction-compression-bending-twisting, with cohesion+friction and Mohr-Coulomb plasticity surface. This law adds adhesion and moments to *Law2_ScGeom_FrictPhys_CundallStrack*.

The normal force is (with the convention of positive tensile forces) $F_n = \min(k_n * u_n, a_n)$, with a_n the normal adhesion. The shear force is $F_s = k_s * u_s$, the plasticity condition defines the maximum value of the shear force, by default $F_s^{max} = F_n * \tan(\varphi) + a_s$, with φ the friction angle and a_s the shear adhesion. If *CohFrictPhys::cohesionDisableFriction* is True, friction is ignored as long as adhesion is active, and the maximum shear force is only $F_s^{max} = a_s$.

If the maximum tensile or maximum shear force is reached and *CohFrictPhys::fragile* =True (default), the cohesive link is broken, and a_n, a_s are set back to zero. If a tensile force is present, the contact is lost, else the shear strength is $F_s^{max} = F_n * \tan(\varphi)$. If *CohFrictPhys::fragile* =False, the behaviour is perfectly plastic, and the shear strength is kept constant.

If *Law2_ScGeom6D_CohFrictPhys_CohesionMoment::momentRotationLaw* =True, bending and twisting moments are computed using a linear law with moduli respectively k_t and k_r, so that the moments are : $M_b = k_b * \Theta_b$ and $M_t = k_t * \Theta_t$, with $\Theta_{b,t}$ the relative rotations between interacting bodies (details can be found in *[Bourrier2013]*). The maximum value of moments can be defined and takes the form of rolling friction. Cohesive -type moment may also be included in the future.

Creep at contact is implemented in this law, as defined in *[Hassan2010]*. If activated, there is a viscous behaviour of the shear and twisting components, and the evolution of the elastic parts of shear displacement and relative twist is given by $du_{s,e}/dt = -F_s/\nu_s$ and $d\Theta_{t,e}/dt = -M_t/\nu_t$.

always_use_moment_law(=*false*)
If true, use bending/twisting moments at all contacts. If false, compute moments only for cohesive contacts.

bases
Ordered list of types (as strings) this functor accepts.

bendingElastEnergy() → float
Compute bending elastic energy.

creep_viscosity(=*1*)
creep viscosity [Pa.s/m]. probably should be moved to Ip2_CohFrictMat_CohFrictMat_-CohFrictPhys.

dict() → dict
Return dictionary of attributes.

elasticEnergy() → float
Compute total elastic energy.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

neverErase(=*false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. *Law2_ScGeom_CapillaryPhys_Capillarity*)

normElastEnergy() → float
Compute normal elastic energy.

shearElastEnergy() → float
Compute shear elastic energy.

shear_creep(=*false*)
activate creep on the shear force, using *CohesiveFrictionalContactLaw::creep_viscosity*.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

traceEnergy(=*false*)
Define the total energy dissipated in plastic slips at all contacts. This will trace only plastic energy in this law, see O.trackEnergy for a more complete energies tracing

twistElastEnergy() → float
Compute twist elastic energy.

twist_creep(=*false*)
activate creep on the twisting moment, using *CohesiveFrictionalContactLaw::creep_viscosity*.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

useIncrementalForm(=*false*)
use the incremental formulation to compute bending and twisting moments. Creep on the twisting moment is not included in such a case.

class yade.wrapper.**Law2_ScGeom6D_InelastCohFrictPhys_CohesionMoment**(*inherits LawFunctor → Functor → Serializable*)
This law is currently under developpement. Final version and documentation will come before the end of 2014.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

normElastEnergy() → float
Compute normal elastic energy.

shearElastEnergy() → float
Compute shear elastic energy.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity**(*inherits LawFunctor → Functor → Serializable*)
Contact law used to simulate granular filler in rock joints *[Duriez2009a]*, *[Duriez2011]*. It includes possibility of cohesion, moment transfer and inelastic compression behaviour (to reproduce the normal inelasticity observed for rock joints, for the latter).

The moment transfer relation corresponds to the adaptation of the work of Plassiard & Belheine (see in *[DeghmReport2006]* for example), which was realized by J. Kozicki, and is now coded in *ScGeom6D*.

As others *LawFunctor*, it uses pre-computed data of the interactions (rigidities, friction angles -with their tan()-, orientations of the interactions); this work is done here in *Ip2_2xNormalInelasticMat_NormalInelasticityPhys*.

To use this you should also use *NormalInelasticMat* as material type of the bodies.

The effects of this law are illustrated in examples/normalInelasticity-test.py

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

momentAlwaysElastic(=*false*)
boolean, true=> the part of the contact torque (caused by relative rotations, which is computed only if momentRotationLaw..) is not limited by a plastic threshold

momentRotationLaw(=*true*)
boolean, true=> computation of a torque (against relative rotation) exchanged between particles

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.Law2_ScGeom_BubblePhys_Bubble(*inherits LawFunctor → Functor → Serializable*)

Constitutive law for Bubble model.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(*=uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

pctMaxForce(*=0.1*)
Chan[2011] states the contact law is valid only for small interferences; therefore an exponential force-displacement curve models the contact stiffness outside that regime (large penetration). This artificial stiffening ensures that bubbles will not pass through eachother or completely overlap during the simulation. The maximum force is Fmax = (2*pi*surfaceTension*rAvg). pctMaxForce is the percentage of the maximum force dictates the separation threshold, Dmax, for each contact. Penetrations less than Dmax calculate the reaction force from the derived contact law, while penetrations equal to or greater than Dmax calculate the reaction force from the artificial exponential curve.

surfaceTension(*=0.07197*)
The surface tension in the liquid surrounding the bubbles. The default value is that of water at 25 degrees Celcius.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.Law2_ScGeom_CpmPhys_Cpm(*inherits LawFunctor → Functor → Serializable*)

Constitutive law for the *cpm-model*.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

elasticEnergy() → float
Compute and return the total elastic energy in all "CpmPhys" contacts

epsSoft(*=-3e-3, approximates confinement -20MPa precisely, -100MPa a little over, -200 and -400 are OK (secant)*)
Strain at which softening in compression starts (non-negative to deactivate)

label(*=uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

omegaThreshold(*=1., >=1. to deactivate, i.e. never delete any contacts*)
damage after which the contact disappears (<1), since omega reaches 1 only for strain →+∞

relKnSoft(*=.3*)
Relative rigidity of the softening branch in compression (0=perfect elastic-plastic, <0 softening, >0 hardening)

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

`yieldEllipseShift`(=*NaN*)
: horizontal scaling of the ellipse (shifts on the +x axis as interactions with +y are given)

`yieldLogSpeed`(=*.1*)
: scaling in the logarithmic yield surface (should be <1 for realistic results; >=0 for meaningful results)

`yieldSigmaTMagnitude`(*(float)sigmaN*, *(float)omega*, *(float)undamagedCohesion*, *(float)tanFrictionAngle*) → float
: Return radius of yield surface for given material and state parameters; uses attributes of the current instance (*yieldSurfType* etc), change them before calling if you need that.

`yieldSurfType`(=*2*)
: yield function: 0: mohr-coulomb (original); 1: parabolic; 2: logarithmic, 3: log+lin_tension, 4: elliptic, 5: elliptic+log

class `yade.wrapper.Law2_ScGeom_FrictPhys_CundallStrack`(*inherits* *LawFunctor* → *Functor* → *Serializable*)

Law for linear compression, and Mohr-Coulomb plasticity surface without cohesion. This law implements the classical linear elastic-plastic law from *[CundallStrack1979]* (see also *[Pfc3dManual30]*). The normal force is (with the convention of positive tensile forces) $F_n = \min(k_n u_n, 0)$. The shear force is $F_s = k_s u_s$, the plasticity condition defines the maximum value of the shear force : $F_s^{max} = F_n \tan(\varphi)$, with φ the friction angle.

This law is well tested in the context of triaxial simulation, and has been used for a number of published results (see e.g. *[Scholtes2009b]* and other papers from the same authors). It is generalised by *Law2_ScGeom6D_CohFrictPhys_CohesionMoment*, which adds cohesion and moments at contact.

`bases`
: Ordered list of types (as strings) this functor accepts.

`dict`() → dict
: Return dictionary of attributes.

`elasticEnergy`() → float
: Compute and return the total elastic energy in all "FrictPhys" contacts

`initPlasticDissipation`(*(float)arg2*) → None
: Initialize cummulated plastic dissipation to a value (0 by default).

`label`(=*uninitalized*)
: Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`neverErase`(=*false*)
: Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. *Law2_ScGeom_CapillaryPhys_Capillarity*)

`plasticDissipation`() → float
: Total energy dissipated in plastic slips at all FrictPhys contacts. Computed only if *Law2_ScGeom_FrictPhys_CundallStrack::traceEnergy* is true.

`sphericalBodies`(=*true*)
: If true, compute branch vectors from radii (faster), else use contactPoint-position. Turning this flag true is safe for sphere-sphere contacts and a few other specific cases. It will give wrong values of torques on facets or boxes.

`timingDeltas`
: Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`traceEnergy`(=*false*)
: Define the total energy dissipated in plastic slips at all contacts. This will trace only plastic energy in this law, see O.trackEnergy for a more complete energies tracing

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom_FrictViscoPhys_CundallStrackVisco**(*inherits LawFunctor → Functor → Serializable*)

Constitutive law for the FrictViscoPM. Corresponds to *Law2_ScGeom_FrictPhys_CundallStrack* with the only difference that viscous damping in normal direction can be considered.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

elasticEnergy() → float
Compute and return the total elastic energy in all "FrictViscoPhys" contacts

initPlasticDissipation(*(float)arg2*) → None
Initialize cummulated plastic dissipation to a value (0 by default).

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

neverErase(=*false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. *Law2_ScGeom_CapillaryPhys_Capillarity*)

plasticDissipation() → float
Total energy dissipated in plastic slips at all FrictPhys contacts. Computed only if :yref:Law2_ScGeom_FrictViscoPhys_CundallStrackVisco::traceEnergy‘ is true.

sphericalBodies(=*true*)
If true, compute branch vectors from radii (faster), else use contactPoint-position. Turning this flag true is safe for sphere-sphere contacts and a few other specific cases. It will give wrong values of torques on facets or boxes.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

traceEnergy(=*false*)
Define the total energy dissipated in plastic slips at all contacts. This will trace only plastic energy in this law, see O.trackEnergy for a more complete energies tracing

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom_JCFpmPhys_JointedCohesiveFrictionalPM**(*inherits LawFunctor → Functor → Serializable*)

Interaction law for cohesive frictional material, e.g. rock, possibly presenting joint surfaces, that can be mechanically described with a smooth contact logic *[Ivars2011]* (implemented in Yade in *[Scholtes2012]*). See examples/jointedCohesiveFrictionalPM for script examples. Joint surface definitions (through stl meshes or direct definition with gts module) are illustrated there.

Key(=*""*)
string specifying the name of saved file 'cracks_____.txt', when *recordCracks* is true.

bases
Ordered list of types (as strings) this functor accepts.

cracksFileExist(=*false*)
if true (and if *recordCracks*), data are appended to an existing 'cracksKey' text file; otherwise its content is reset.

dict() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`neverErase`(=*false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene

`recordCracks`(=*false*)
if true, data about interactions that lose their cohesive feature are stored in a text file cracksKey.txt (see *Key* and *cracksFileExist*). It contains 9 columns: the break iteration, the 3 coordinates of the contact point, the type (1 means shear break, while 0 corresponds to tensile break), the ''cross section" (mean radius of the 2 spheres) and the 3 coordinates of the contact normal.

`smoothJoint`(=*false*)
if true, interactions of particles belonging to joint surface (*JCFpmPhys.isOnJoint*) are handled according to a smooth contact logic *[Ivars2011]*, *[Scholtes2012]*.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Law2_ScGeom_LudingPhys_Basic`(*inherits LawFunctor → Functor → Serializable*)
Linear viscoelastic model operating on *ScGeom* and *LudingPhys*.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Law2_ScGeom_MindlinPhys_HertzWithLinearShear`(*inherits LawFunctor → Functor → Serializable*)
Constitutive law for the Hertz formulation (using *MindlinPhys.kno*) and linear beahvior in shear (using *MindlinPhys.kso* for stiffness and *FrictPhys.tangensOfFrictionAngle*).

Note: No viscosity or damping. If you need those, look at *Law2_ScGeom_MindlinPhys_Mindlin*, which also includes non-linear Mindlin shear.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

neverErase(=*false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. *Law2_ScGeom_CapillaryPhys_Capillarity*)

nonLin(=*0*)
Shear force nonlinearity (the value determines how many features of the non-linearity are taken in account). 1: ks as in HM 2: shearElastic increment computed as in HM 3. granular ratcheting disabled.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom_MindlinPhys_Mindlin**(*inherits LawFunctor → Functor → Serializable*)
Constitutive law for the Hertz-Mindlin formulation. It includes non linear elasticity in the normal direction as predicted by Hertz for two non-conforming elastic contact bodies. In the shear direction, instead, it reseambles the simplified case without slip discussed in Mindlin's paper, where a linear relationship between shear force and tangential displacement is provided. Finally, the Mohr-Coulomb criterion is employed to established the maximum friction force which can be developed at the contact. Moreover, it is also possible to include the effect of linear viscous damping through the definition of the parameters β_n and β_s.

bases
Ordered list of types (as strings) this functor accepts.

calcEnergy(=*false*)
bool to calculate energy terms (shear potential energy, dissipation of energy due to friction and dissipation of energy due to normal and tangential damping)

contactsAdhesive() → float
Compute total number of adhesive contacts.

dict() → dict
Return dictionary of attributes.

frictionDissipation(=*uninitalized*)
Energy dissipation due to sliding

includeAdhesion(=*false*)
bool to include the adhesion force following the DMT formulation. If true, also the normal elastic energy takes into account the adhesion effect.

includeMoment(=*false*)
bool to consider rolling resistance (if *Ip2_FrictMat_FrictMat_MindlinPhys::eta* is 0.0, no plastic condition is applied.)

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

neverErase(=*false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. *Law2_ScGeom_CapillaryPhys_Capillarity*)

normDampDissip(=*uninitalized*)
Energy dissipated by normal damping

normElastEnergy() → float
Compute normal elastic potential energy. It handles the DMT formulation if *Law2_ScGeom_MindlinPhys_Mindlin::includeAdhesion* is set to true.

preventGranularRatcheting(=*true*)
bool to avoid granular ratcheting

`ratioSlidingContacts()` → float
Return the ratio between the number of contacts sliding to the total number at a given time.

`shearDampDissip(=uninitalized)`
Energy dissipated by tangential damping

`shearEnergy(=uninitalized)`
Shear elastic potential energy

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs(dict)arg2)` → None
Update object attributes from given dictionary

class `yade.wrapper.Law2_ScGeom_MindlinPhys_MindlinDeresiewitz(inherits LawFunctor → Functor → Serializable)`
Hertz-Mindlin contact law with partial slip solution, as described in *[Thornton1991]*.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label(=uninitalized)`
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`neverErase(=false)`
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. *Law2_ScGeom_CapillaryPhys_Capillarity*)

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs(dict)arg2)` → None
Update object attributes from given dictionary

class `yade.wrapper.Law2_ScGeom_ViscElCapPhys_Basic(inherits LawFunctor → Functor → Serializable)`
Extended version of Linear viscoelastic model with capillary parameters.

`NLiqBridg(=uninitalized)`
The total number of liquid bridges

`VLiqBridg(=uninitalized)`
The total volume of liquid bridges

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label(=uninitalized)`
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs(dict)arg2)` → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom_ViscElPhys_Basic**(*inherits LawFunctor → Functor → Serializable*)

Linear viscoelastic model operating on ScGeom and ViscElPhys. The contact law is visco-elastic in the normal direction, and visco-elastic frictional in the tangential direction. The normal contact is modelled as a spring of equivalent stiffness k_n, placed in parallel with a viscous damper of equivalent viscosity c_n. As for the tangential contact, it is made of a spring-dashpot system (in parallel with equivalent stiffness k_s and viscosity c_s) in serie with a slider of friction coefficient $\mu = \tan\varphi$.

The friction coefficient $\mu = \tan\varphi$ is always evaluated as $\tan(\min(\varphi_1, \varphi_2))$, where φ_1 and φ_2 are respectively the friction angle of particle 1 and 2. For the other parameters, depending on the material input, the equivalent parameters of the contact (K_n,C_n,K_s,C_s,φ) are evaluated differently. In the following, the quantities in parenthesis are the material constant which are precised for each particle. They are then associated to particle 1 and 2 (e.g. kn_1,kn_2,cn_1...), and should not be confused with the equivalent parameters of the contact (K_n,C_n,K_s,C_s,φ).

- If contact time (tc), normal and tangential restitution coefficient (en,et) are precised, the equivalent parameters are evaluated following the formulation of Pournin *[Pournin2001]*.
- If normal and tangential stiffnesses (kn, ks) and damping constant (cn,cs) of each particle are precised, the equivalent stiffnesses and damping constants of each contact made of two particles 1 and 2 is made $A = 2\frac{a_1 a_2}{a_1+a_2}$, where A is K_n, K_s, C_n and C_s, and 1 and 2 refer to the value associated to particle 1 and 2.
- Alternatively it is possible to precise the Young modulus (young) and poisson's ratio (poisson) instead of the normal and spring constant (kn and ks). In this case, the equivalent parameters are evaluated the same way as the previous case with $kn_x = E_x d_x$, $ks_x = \nu_x kn_x$, where E_x, ν_x and d_x are Young modulus, poisson's ratio and diameter of particle x.
- If Yound modulus (young), poisson's ratio (poisson), normal and tangential restitution coefficient (en,et) are precised, the equivalent stiffnesses are evaluated as previously: $K_n = 2\frac{kn_1 kn_2}{kn_1+kn_2}$, $kn_x = E_x d_x$, $K_s = 2(ks_1 ks_2)/(ks_1 + ks_2)$, $ks_x = \nu kn_x$. The damping constant is computed at each contact in order to fulfill the normal restitution coefficient $e_n = (en_1 en_2)/(en_1 + en_2)$. This is achieved resolving numerically equation 21 of *[Schwager2007]* (There is in fact a mistake in the article from equation 18 to 19, so that there is a change in sign). Be careful in this configuration the tangential restitution coefficient is set to 1 (no tangential damping). This formulation imposes directly the normal restitution coefficient of the collisions instead of the damping constant.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGeom_ViscoFrictPhys_CundallStrack**(*inherits Law2_ScGeom_FrictPhys_CundallStrack → LawFunctor → Functor → Serializable*)

Law similar to *Law2_ScGeom_FrictPhys_CundallStrack* with the addition of shear creep at contacts.

bases
Ordered list of types (as strings) this functor accepts.

creepStiffness(=*1*)

dict() → dict
Return dictionary of attributes.

elasticEnergy() → float
Compute and return the total elastic energy in all "FrictPhys" contacts

initPlasticDissipation(*(float)arg2*) → None
Initialize cummulated plastic dissipation to a value (0 by default).

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

neverErase(=*false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. *Law2_ScGeom_CapillaryPhys_Capillarity*)

plasticDissipation() → float
Total energy dissipated in plastic slips at all FrictPhys contacts. Computed only if *Law2_ScGeom_FrictPhys_CundallStrack::traceEnergy* is true.

shearCreep(=*false*)

sphericalBodies(=*true*)
If true, compute branch vectors from radii (faster), else use contactPoint-position. Turning this flag true is safe for sphere-sphere contacts and a few other specific cases. It will give wrong values of torques on facets or boxes.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

traceEnergy(=*false*)
Define the total energy dissipated in plastic slips at all contacts. This will trace only plastic energy in this law, see O.trackEnergy for a more complete energies tracing

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

viscosity(=*1*)

class yade.wrapper.**Law2_ScGeom_WirePhys_WirePM**(*inherits LawFunctor → Functor → Serializable*)
Constitutive law for the wire model.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

linkThresholdIteration(=*1*)
Iteration to create the link.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGridCoGeom_CohFrictPhys_CundallStrack**(*inherits LawFunctor → Functor → Serializable*)

Law between a cohesive frictional *GridConnection* and a cohesive frictional *Sphere*. Almost the same than *Law2_ScGeom6D_CohFrictPhys_CohesionMoment*, but THE ROTATIONAL MOMENTS ARE NOT COMPUTED.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

neverErase(=*false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. *Law2_ScGeom_CapillaryPhys_Capillarity*)

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_ScGridCoGeom_FrictPhys_CundallStrack**(*inherits LawFunctor → Functor → Serializable*)

Law between a frictional *GridConnection* and a frictional *Sphere*. Almost the same than *Law2_ScGeom_FrictPhys_CundallStrack*, but the force is divided and applied on the two *GridNodes* only.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

neverErase(=*false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. *Law2_ScGeom_CapillaryPhys_Capillarity*)

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Law2_TTetraSimpleGeom_NormPhys_Simple**(*inherits LawFunctor → Functor → Serializable*)

EXPERIMENTAL. TODO

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

8.8.2 LawDispatcher

class yade.wrapper.**LawDispatcher**(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling *functors* based on received argument type(s).

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

dispFunctor(*(IGeom)arg2*, *(IPhys)arg3*) → LawFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix([*(bool)names=True*]) → dict
Return dictionary with contents of the dispatch matrix.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==True).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==True).

functors
Functors associated with this dispatcher.

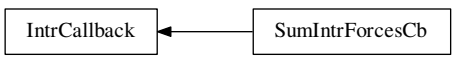
label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

8.9 Callbacks



class `yade.wrapper.IntrCallback`(*inherits Serializable*)
Abstract callback object which will be called for every (real) *Interaction* after the interaction has been processed by *InteractionLoop*.

At the beginning of the interaction loop, `stepInit` is called, initializing the object; it returns either `NULL` (to deactivate the callback during this time step) or pointer to function, which will then be passed (1) pointer to the callback object itself and (2) pointer to *Interaction*.

Note: (NOT YET DONE) This functionality is accessible from python by passing 4th argument to *InteractionLoop* constructor, or by appending the callback object to *InteractionLoop::callbacks*.

`dict()` → dict
Return dictionary of attributes.

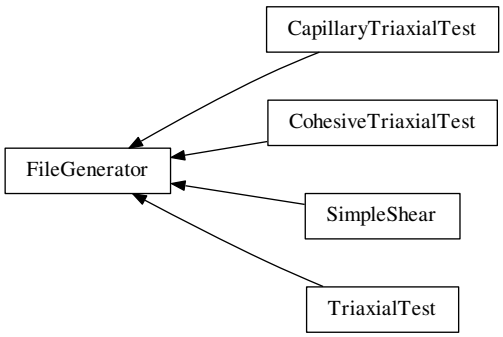
`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.SumIntrForcesCb`(*inherits IntrCallback → Serializable*)
Callback summing magnitudes of forces over all interactions. *IPhys* of interactions must derive from *NormShearPhys* (responsability fo the user).

`dict()` → dict
Return dictionary of attributes.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

8.10 Preprocessors



class `yade.wrapper.FileGenerator`(*inherits Serializable*)
Base class for scene generators, preprocessors.

`dict()` → dict
Return dictionary of attributes.

`generate`(*(str)out*) → None
Generate scene, save to given file

`load()` → None
Generate scene, save to temporary file and load immediately

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.CapillaryTriaxialTest`(*inherits FileGenerator → Serializable*)
This preprocessor is a variant of TriaxialTest, including the model of capillary forces developed as part of the PhD of Luc Scholtès. See the documentation of Law2_ScGeom_CapillaryPhys_-Capillarity or the main page https://yade-dem.org/wiki/CapillaryTriaxialTest, for more details.

Results obtained with this preprocessor were reported for instance in 'Scholtes et al. Micromechanics of granular materials with capillary effects. International Journal of Engineering Science 2009,(47)1, 64-75.'

`Key(=""")`
A code that is added to output filenames.

`Rdispersion(=0.3)`
Normalized standard deviation of generated sizes.

`StabilityCriterion(=0.01)`
Value of unbalanced force for which the system is considered stable. Used in conditionals to switch between loading stages.

`WallStressRecordFile(="./WallStressesWater"+Key)`

`autoCompressionActivation(=true)`
Do we just want to generate a stable packing under isotropic pressure (false) or do we want the triaxial loading to start automatically right after compaction stage (true)?

`autoStopSimulation(=false)`
freeze the simulation when conditions are reached (don't activate this if you want to be able to run/stop from Qt GUI)

`autoUnload(=true)`
auto adjust the isotropic stress state from *TriaxialTest::sigmaIsoCompaction* to *TriaxialTest::sigmaLateralConfinement* if they have different values. See docs for *TriaxialCompressionEngine::autoUnload*

`biaxial2dTest(=false)`
FIXME : what is that?

`binaryFusion(=true)`
Defines how overlapping bridges affect the capillary forces (see *CapillaryTriaxialTest::fusionDetection*). If binary=true, the force is null as soon as there is an overlap detected, if not, the force is divided by the number of overlaps.

`boxFrictionDeg(=0.0)`
Friction angle [°] of boundaries contacts.

`boxKsDivKn(=0.5)`
Ratio of shear vs. normal contact stiffness for boxes.

`boxWalls(=true)`
Use boxes for boundaries (recommended).

`boxYoungModulus(=15000000.0)`
Stiffness of boxes.

`capillaryPressure(=0)`
Define succion in the packing [Pa]. This is the value used in the capillary model.

`capillaryStressRecordFile(="./capStresses"+Key)`

`compactionFrictionDeg(=sphereFrictionDeg)`
Friction angle [°] of spheres during compaction (different values result in different porosities)]. This value is overridden by *TriaxialTest::sphereFrictionDeg* before triaxial testing.

`contactStressRecordFile(="./contStresses"+Key)`

`dampingForce(=0.2)`
Coefficient of Cundal-Non-Viscous damping (applied on on the 3 components of forces)

`dampingMomentum(=0.2)`
Coefficient of Cundal-Non-Viscous damping (applied on on the 3 components of torques)

`defaultDt(=0.0001)`
Max time-step. Used as initial value if defined. Latter adjusted by the time stepper.

density(=*2600*)
density of spheres

dict() → dict
Return dictionary of attributes.

facetWalls(=*false*)
Use facets for boundaries (not tested)

finalMaxMultiplier(=*1.001*)
max multiplier of diameters during internal compaction (secondary precise adjustment)

fixedBoxDims(=*""*)
string that contains some subset (max. 2) of {'x','y','z'} ; contains axes will have box dimension hardcoded, even if box is scaled as mean_radius is prescribed: scaling will be applied on the rest.

fixedPoroCompaction(=*false*)
flag to choose an isotropic compaction until a fixed porosity choosing a same translation speed for the six walls

fixedPorosity(=*1*)
FIXME : what is that?

fusionDetection(=*false*)
test overlaps between liquid bridges on modify forces if overlaps exist

generate(*(str)out*) → None
Generate scene, save to given file

importFilename(=*""*)
File with positions and sizes of spheres.

internalCompaction(=*false*)
flag for choosing between moving boundaries or increasing particles sizes during the compaction stage.

load() → None
Generate scene, save to temporary file and load immediately

lowerCorner(=*Vector3r(0, 0, 0)*)
Lower corner of the box.

maxMultiplier(=*1.01*)
max multiplier of diameters during internal compaction (initial fast increase)

maxWallVelocity(=*10*)
max velocity of boundaries. Usually useless, but can help stabilizing the system in some cases.

noFiles(=*false*)
Do not create any files during run (.xml, .spheres, wall stress records)

numberOfGrains(=*400*)
Number of generated spheres.

radiusControlInterval(=*10*)
interval between size changes when growing spheres.

radiusMean(=*-1*)
Mean radius. If negative (default), autocomputed to as a function of box size and *TriaxialTest::numberOfGrains*

recordIntervalIter(=*20*)
interval between file outputs

sigmaIsoCompaction(=*-50000*)
Confining stress during isotropic compaction (< 0 for real - compressive - compaction).

sigmaLateralConfinement(=*-50000*)
Lateral stress during triaxial loading (< 0 for classical compressive cases). An isotropic unloading is performed if the value is not equal to *CapillaryTriaxialTest::SigmaIsoCompaction*.

sphereFrictionDeg(=*18.0*)
Friction angle [°] of spheres assigned just before triaxial testing.

sphereKsDivKn(=*0.5*)
Ratio of shear vs. normal contact stiffness for spheres.

sphereYoungModulus(=*15000000.0*)
Stiffness of spheres.

strainRate(=*1*)
Strain rate in triaxial loading.

thickness(=*0.001*)
thickness of boundaries. It is arbitrary and should have no effect

timeStepOutputInterval(=*50*)
interval for outputing general information on the simulation (stress,unbalanced force,...)

timeStepUpdateInterval(=*50*)
interval for *GlobalStiffnessTimeStepper*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

upperCorner(=*Vector3r(1, 1, 1)*)
Upper corner of the box.

wallOversizeFactor(=*1.3*)
Make boundaries larger than the packing to make sure spheres don't go out during deformation.

wallStiffnessUpdateInterval(=*10*)
interval for updating the stiffness of sample/boundaries contacts

wallWalls(=*false*)
Use walls for boundaries (not tested)

water(=*true*)
activate capillary model

class yade.wrapper.**CohesiveTriaxialTest**(*inherits FileGenerator → Serializable*)
This preprocessor is a variant of TriaxialTest using the cohesive-frictional contact law with moments. It sets up a scene for cohesive triaxial tests. See full documentation at http://yade-dem.org/wiki/TriaxialTest.

Cohesion is initially 0 by default. The suggested usage is to define cohesion values in a second step, after isotropic compaction : define shear and normal cohesions in *Ip2_CohFrictMat_CohFrictMat_CohFrictPhys*, then turn *Ip2_CohFrictMat_CohFrictMat_CohFrictPhys*::setCohesionNow true to assign them at each contact at next iteration.

Key(=*""*)
A code that is added to output filenames.

StabilityCriterion(=*0.01*)
Value of unbalanced force for which the system is considered stable. Used in conditionals to switch between loading stages.

WallStressRecordFile(=*"./CohesiveWallStresses"+Key*)

autoCompressionActivation(=*true*)
Do we just want to generate a stable packing under isotropic pressure (false) or do we want the triaxial loading to start automatically right after compaction stage (true)?

autoStopSimulation(=*false*)
freeze the simulation when conditions are reached (don't activate this if you want to be able to run/stop from Qt GUI)

autoUnload(=*true*)
auto adjust the isotropic stress state from *TriaxialTest*::sigmaIsoCompaction to *TriaxialTest::sigmaLateralConfinement* if they have different values. See docs for *TriaxialCompressionEngine::autoUnload*

biaxial2dTest(=*false*)
FIXME : what is that?

boxFrictionDeg(=*0.0*)
Friction angle [°] of boundaries contacts.

boxKsDivKn(=*0.5*)
Ratio of shear vs. normal contact stiffness for boxes.

boxWalls(=*true*)
Use boxes for boundaries (recommended).

boxYoungModulus(=*15000000.0*)
Stiffness of boxes.

compactionFrictionDeg(=*sphereFrictionDeg*)
Friction angle [°] of spheres during compaction (different values result in different porosities)]. This value is overridden by *TriaxialTest::sphereFrictionDeg* before triaxial testing.

dampingForce(=*0.2*)
Coefficient of Cundal-Non-Viscous damping (applied on on the 3 components of forces)

dampingMomentum(=*0.2*)
Coefficient of Cundal-Non-Viscous damping (applied on on the 3 components of torques)

defaultDt(=*0.001*)
Max time-step. Used as initial value if defined. Latter adjusted by the time stepper.

density(=*2600*)
density of spheres

dict() → dict
Return dictionary of attributes.

facetWalls(=*false*)
Use facets for boundaries (not tested)

finalMaxMultiplier(=*1.001*)
max multiplier of diameters during internal compaction (secondary precise adjustment)

fixedBoxDims(=*""*)
string that contains some subset (max. 2) of {'x','y','z'} ; contains axes will have box dimension hardcoded, even if box is scaled as mean_radius is prescribed: scaling will be applied on the rest.

fixedPoroCompaction(=*false*)
flag to choose an isotropic compaction until a fixed porosity choosing a same translation speed for the six walls

fixedPorosity(=*1*)
FIXME : what is that?

generate(*(str)out*) → None
Generate scene, save to given file

importFilename(=*""*)
File with positions and sizes of spheres.

internalCompaction(=*false*)
flag for choosing between moving boundaries or increasing particles sizes during the compaction stage.

load() → None
Generate scene, save to temporary file and load immediately

lowerCorner(=*Vector3r(0, 0, 0)*)
Lower corner of the box.

maxMultiplier(=*1.01*)
max multiplier of diameters during internal compaction (initial fast increase)

maxWallVelocity(=*10*)
max velocity of boundaries. Usually useless, but can help stabilizing the system in some cases.

noFiles(=*false*)
Do not create any files during run (.xml, .spheres, wall stress records)

normalCohesion(=*0*)
Material parameter used to define contact strength in tension.

numberOfGrains(=*400*)
Number of generated spheres.

radiusControlInterval(=*10*)
interval between size changes when growing spheres.

radiusDeviation(=*0.3*)
Normalized standard deviation of generated sizes.

radiusMean(=*-1*)
Mean radius. If negative (default), autocomputed to as a function of box size and *TriaxialTest::numberOfGrains*

recordIntervalIter(=*20*)
interval between file outputs

setCohesionOnNewContacts(=*false*)
create cohesionless (False) or cohesive (True) interactions for new contacts.

shearCohesion(=*0*)
Material parameter used to define shear strength of contacts.

sigmaIsoCompaction(=*-50000*)
Confining stress during isotropic compaction (< 0 for real - compressive - compaction).

sigmaLateralConfinement(=*-50000*)
Lateral stress during triaxial loading (< 0 for classical compressive cases). An isotropic unloading is performed if the value is not equal to *TriaxialTest::sigmaIsoCompaction*.

sphereFrictionDeg(=*18.0*)
Friction angle [°] of spheres assigned just before triaxial testing.

sphereKsDivKn(=*0.5*)
Ratio of shear vs. normal contact stiffness for spheres.

sphereYoungModulus(=*15000000.0*)
Stiffness of spheres.

strainRate(=*0.1*)
Strain rate in triaxial loading.

thickness(=*0.001*)
thickness of boundaries. It is arbitrary and should have no effect

timeStepUpdateInterval(=*50*)
interval for *GlobalStiffnessTimeStepper*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

upperCorner(=*Vector3r(1, 1, 1)*)
Upper corner of the box.

wallOversizeFactor(=*1.3*)
Make boundaries larger than the packing to make sure spheres don't go out during deformation.

wallStiffnessUpdateInterval(*=10*)
interval for updating the stiffness of sample/boundaries contacts

wallWalls(*=false*)
Use walls for boundaries (not tested)

class yade.wrapper.**SimpleShear**(*inherits FileGenerator → Serializable*)
Preprocessor for creating a numerical model of a simple shear box.

- Boxes (6) constitute the different sides of the box itself
- Spheres are contained in the box. The sample is generated by default via the same method used in TriaxialTest Preprocesor (=> see in source function GenerateCloud). But import of a list of spheres from a text file can be also performed after few changes in the source code.

  Launching this preprocessor will carry out an oedometric compression, until a value of normal stress equal to 2 MPa (and stable). But with others Engines *KinemCNDEngine*, *KinemCNSEngine* and *KinemCNLEngine*, respectively constant normal displacement, constant normal rigidity and constant normal stress paths can be carried out for such simple shear boxes.

NB about micro-parameters : their default values correspond to those used in *[Duriez2009a]* and *[Duriez2011]* to simulate infilled rock joints.

boxPoissonRatio(*=0.04*)
value of *ElastMat::poisson* for the spheres [-]

boxYoungModulus(*=4.0e9*)
value of *ElastMat::young* for the boxes [Pa]

density(*=2600*)
density of the spheres [kg/m^3]

dict() → dict
Return dictionary of attributes.

generate(*(str)out*) → None
Generate scene, save to given file

gravApplied(*=false*)
depending on this, *GravityEngine* is added or not to the scene to take into account the weight of particles

gravity(*=Vector3r(0, -9.81, 0)*)
vector corresponding to used gravity (if *gravApplied*) [m/s^2]

height(*=0.02*)
initial height (along y-axis) of the shear box [m]

length(*=0.1*)
initial length (along x-axis) of the shear box [m]

load() → None
Generate scene, save to temporary file and load immediately

sphereFrictionDeg(*=37*)
value of *ElastMat::poisson* for the spheres [°] (the necessary conversion in rad is done automatically)

spherePoissonRatio(*=0.04*)
value of *ElastMat::poisson* for the spheres [-]

sphereYoungModulus(*=4.0e9*)
value of *ElastMat::young* for the spheres [Pa]

thickness(*=0.001*)
thickness of the boxes constituting the shear box [m]

`timeStepUpdateInterval`(*=50*)
value of *TimeStepper::timeStepUpdateInterval* for the *TimeStepper* used here

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`width`(*=0.04*)
initial width (along z-axis) of the shear box [m]

class `yade.wrapper.TriaxialTest`(*inherits FileGenerator → Serializable*)
Create a scene for triaxal test.

Introduction Yade includes tools to simulate triaxial tests on particles assemblies. This preprocessor (and variants like e.g. *CapillaryTriaxialTest*) illustrate how to use them. It generates a scene which will - by default - go through the following steps :

- generate random loose packings in a parallelepiped.
- compress the packing isotropicaly, either squeezing the packing between moving rigid boxes or expanding the particles while boxes are fixed (depending on flag *internalCompaction*). The confining pressure in this stage is defined via *sigmaIsoCompaction*.
- when the packing is dense and stable, simulate a loading path and get the mechanical response as a result.

The default loading path corresponds to a constant lateral stress (*sigmaLateralConfinement*) in 2 directions and constant strain rate on the third direction. This default loading path is performed when the flag *autoCompressionActivation* it `True`, otherwise the simulation stops after isotropic compression.

Different loading paths might be performed. In order to define them, the user can modify the flags found in engine *TriaxialStressController* at any point in the simulation (in c++). If `TriaxialStressController.wall_X_activated` is `true` boundary X is moved automatically to maintain the defined stress level *sigmaN* (see axis conventions below). If `false` the boundary is not controlled by the engine at all. In that case the user is free to prescribe fixed position, constant velocity, or more complex conditions.

Note: *Axis conventions.* Boundaries perpendicular to the x axis are called "left" and "right", y corresponds to "top" and "bottom", and axis z to "front" and "back". In the default loading path, strain rate is assigned along y, and constant stresses are assigned on x and z.

Essential engines

1. The *TriaxialCompressionEngine* is used for controlling the state of the sample and simulating loading paths. *TriaxialCompressionEngine* inherits from *TriaxialStressController*, which computes stress- and strain-like quantities in the packing and maintain a constant level of stress at each boundary. *TriaxialCompressionEngine* has few more members in order to impose constant strain rate and control the transition between isotropic compression and triaxial test. Transitions are defined by changing some flags of the *TriaxialStressController*, switching from/to imposed strain rate to/from imposed stress.
2. The class *TriaxialStateRecorder* is used to write to a file the history of stresses and strains.
3. *TriaxialTest* is using *GlobalStiffnessTimeStepper* to compute an appropriate Δt for the numerical scheme.

Note: `TriaxialStressController::ComputeUnbalancedForce` returns a value that can be useful for evaluating the stability of the packing. It is defined as (mean force on particles)/(mean contact force), so that it tends to 0 in a stable packing. This parameter is checked by *TriaxialCompressionEngine* to switch from one stage of the simulation to the next one (e.g. stop isotropic confinment and start axial loading)

Frequently Asked Questions

1. **How is generated the packing? How to change particles sizes distribution? Why do I have a m**
   The initial positioning of spheres is done by generating random (x,y,z) in a box and checking if a sphere of radius R (R also randomly generated with respect to a uniform distribution between mean*(1-std_dev) and mean*(1+std_dev) can be inserted at this location without overlaping with others.

   If the sphere overlaps, new (x,y,z)'s are generated until a free position for the new sphere is found. This explains the message you have: after 3000 trial-and-error, the sphere couldn't be placed, and the algorithm stops.

   You get the message above if you try to generate an initialy dense packing, which is not possible with this algorithm. It can only generate clouds. You should keep the default value of porosity (n~0.7), or even increase if it is still to low in some cases. The dense state will be obtained in the second step (compaction, see below).

2. **How is the compaction done, what are the parameters *maxWallVelocity* and *finalMaxMultiplie***

   Compaction is done

   (a) by moving rigid boxes or

   (b) by increasing the sizes of the particles (decided using the option *internalCompaction* size increase).

   Both algorithm needs numerical parameters to prevent instabilities. For instance, with the method (1) *maxWallVelocity* is the maximum wall velocity, with method (2) *finalMaxMultiplier* is the max value of the multiplier applied on sizes at each iteration (always something like 1.00001).

3. **During the simulation of triaxial compression test, the wall in one direction moves with an incr**
   The control of stress on a boundary is based on the total stiffness K of all contacts between the packing and this boundary. In short, at each step, displacement=stress_error/K. This algorithm is implemented in *TriaxialStressController*, and the control itself is in `TriaxialStressController::ControlExternalStress`. The control can be turned off independently for each boundary, using the flags `wall_XXX_activated`, with *XXX* {*top*, *bottom*, *left*, *right*, *back*, *front*}. The imposed sress is a unique value (*sigma_iso*) for all directions if *TriaxialStressController.isAxisymetric*, or 3 independent values *sigma1*, *sigma2*, *sigma3*.

4. **Which value of friction angle do you use during the compaction phase of the Triaxial Test?**
   The friction during the compaction (whether you are using the expansion method or the compression one for the specimen generation) can be anything between 0 and the final value used during the Triaxial phase. Note that higher friction than the final one would result in volumetric collapse at the beginning of the test. The purpose of using a different value of friction during this phase is related to the fact that the final porosity you get at the end of the sample generation essentially depends on it as well as on the assumed Particle Size Distribution. Changing the initial value of friction will get to a different value of the final porosity.

5. **Which is the aim of the `bool isRadiusControlIteration`?** This internal variable (updated automatically) is true each N timesteps (with N=*radiusControlInterval*). For other timesteps, there is no expansion. Cycling without expanding is just a way to speed up the simulation, based on the idea that 1% increase each 10 iterations needs less operations than 0.1% at each iteration, but will give similar results.

6. **How comes the unbalanced force reaches a low value only after many timesteps in the compact**
   The value of unbalanced force (dimensionless) is expected to reach low value (i.e. identifying a static-equilibrium condition for the specimen) only at the end of the compaction phase. The code is not aiming at simulating a quasistatic isotropic compaction process, it is only giving a stable packing at the end of it.

`Key(="")`
A code that is added to output filenames.

StabilityCriterion(=*0.01*)
Value of unbalanced force for which the system is considered stable. Used in conditionals to switch between loading stages.

WallStressRecordFile(=*"./WallStresses"+Key*)

autoCompressionActivation(=*true*)
Do we just want to generate a stable packing under isotropic pressure (false) or do we want the triaxial loading to start automatically right after compaction stage (true)?

autoStopSimulation(=*false*)
freeze the simulation when conditions are reached (don't activate this if you want to be able to run/stop from Qt GUI)

autoUnload(=*true*)
auto adjust the isotropic stress state from *TriaxialTest::sigmaIsoCompaction* to *TriaxialTest::sigmaLateralConfinement* if they have different values. See docs for *TriaxialCompressionEngine::autoUnload*

biaxial2dTest(=*false*)
FIXME : what is that?

boxFrictionDeg(=*0.0*)
Friction angle [°] of boundaries contacts.

boxKsDivKn(=*0.5*)
Ratio of shear vs. normal contact stiffness for boxes.

boxYoungModulus(=*15000000.0*)
Stiffness of boxes.

compactionFrictionDeg(=*sphereFrictionDeg*)
Friction angle [°] of spheres during compaction (different values result in different porosities)]. This value is overridden by *TriaxialTest::sphereFrictionDeg* before triaxial testing.

dampingForce(=*0.2*)
Coefficient of Cundal-Non-Viscous damping (applied on on the 3 components of forces)

dampingMomentum(=*0.2*)
Coefficient of Cundal-Non-Viscous damping (applied on on the 3 components of torques)

defaultDt(=*-1*)
Max time-step. Used as initial value if defined. Latter adjusted by the time stepper.

density(=*2600*)
density of spheres

dict() → dict
Return dictionary of attributes.

facetWalls(=*false*)
Use facets for boundaries (not tested)

finalMaxMultiplier(=*1.001*)
max multiplier of diameters during internal compaction (secondary precise adjustment)

fixedBoxDims(=*""*)
string that contains some subset (max. 2) of {'x','y','z'} ; contains axes will have box dimension hardcoded, even if box is scaled as mean_radius is prescribed: scaling will be applied on the rest.

generate(*(str)out*) → None
Generate scene, save to given file

importFilename(=*""*)
File with positions and sizes of spheres.

internalCompaction(=*false*)
flag for choosing between moving boundaries or increasing particles sizes during the compaction stage.

load() → None
Generate scene, save to temporary file and load immediately

lowerCorner(=*Vector3r(0, 0, 0)*)
Lower corner of the box.

maxMultiplier(=*1.01*)
max multiplier of diameters during internal compaction (initial fast increase)

maxWallVelocity(=*10*)
max velocity of boundaries. Usually useless, but can help stabilizing the system in some cases.

noFiles(=*false*)
Do not create any files during run (.xml, .spheres, wall stress records)

numberOfGrains(=*400*)
Number of generated spheres.

radiusControlInterval(=*10*)
interval between size changes when growing spheres.

radiusMean(=*-1*)
Mean radius. If negative (default), autocomputed to as a function of box size and *TriaxialTest::numberOfGrains*

radiusStdDev(=*0.3*)
Normalized standard deviation of generated sizes.

recordIntervalIter(=*20*)
interval between file outputs

sigmaIsoCompaction(=*-50000*)
Confining stress during isotropic compaction (< 0 for real - compressive - compaction).

sigmaLateralConfinement(=*-50000*)
Lateral stress during triaxial loading (< 0 for classical compressive cases). An isotropic unloading is performed if the value is not equal to *TriaxialTest::sigmaIsoCompaction*.

sphereFrictionDeg(=*18.0*)
Friction angle [°] of spheres assigned just before triaxial testing.

sphereKsDivKn(=*0.5*)
Ratio of shear vs. normal contact stiffness for spheres.

sphereYoungModulus(=*15000000.0*)
Stiffness of spheres.

strainRate(=*0.1*)
Strain rate in triaxial loading.

thickness(=*0.001*)
thickness of boundaries. It is arbitrary and should have no effect

timeStepUpdateInterval(=*50*)
interval for *GlobalStiffnessTimeStepper*

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

upperCorner(=*Vector3r(1, 1, 1)*)
Upper corner of the box.

wallOversizeFactor(=*1.3*)
Make boundaries larger than the packing to make sure spheres don't go out during deformation.

wallStiffnessUpdateInterval(=*10*)
interval for updating the stiffness of sample/boundaries contacts

wallWalls(=*false*)
Use walls for boundaries (not tested)

8.11 Rendering

8.11.1 OpenGLRenderer

class `yade.wrapper.OpenGLRenderer`(*inherits Serializable*)
Class responsible for rendering scene on OpenGL devices.

`bgColor`(=*Vector3r(.2, .2, .2)*)
Color of the background canvas (RGB)

`bound`(=*false*)
Render body *Bound*

`cellColor`(=*Vector3r(1, 1, 0)*)
Color of the periodic cell (RGB).

`clipPlaneActive`(=*vector<bool>(numClipPlanes, false)*)
Activate/deactivate respective clipping planes

`clipPlaneSe3`(=*vector<Se3r>(numClipPlanes, Se3r(Vector3r::Zero(), Quaternionr::Identity()))*)
Position and orientation of clipping planes

`dict`() → dict
Return dictionary of attributes.

`dispScale`(=*Vector3r::Ones()*, *disable scaling*)
Artificially enlarge (scale) dispalcements from bodies' *reference positions* by this relative amount, so that they become better visible (independently in 3 dimensions). Disbled if (1,1,1).

`dof`(=*false*)
Show which degrees of freedom are blocked for each body

`extraDrawers`(=*uninitalized*)
Additional rendering components (*GlExtraDrawer*).

`ghosts`(=*true*)
Render objects crossing periodic cell edges by cloning them in multiple places (periodic simulations only).

`hideBody`(*(int)id*) → None
Hide body from id (see *OpenGLRenderer::showBody*)

`id`(=*false*)
Show body id's

`intrAllWire`(=*false*)
Draw wire for all interactions, blue for potential and green for real ones (mostly for debugging)

`intrGeom`(=*false*)
Render *Interaction::geom* objects.

`intrPhys`(=*false*)
Render *Interaction::phys* objects

`intrWire`(=*false*)
If rendering interactions, use only wires to represent them.

`light1`(=*true*)
Turn light 1 on.

`light2`(=*true*)
Turn light 2 on.

`light2Color`(=*Vector3r(0.5, 0.5, 0.1)*)
Per-color intensity of secondary light (RGB).

`light2Pos`(=*Vector3r(-130, 75, 30)*)
Position of secondary OpenGL light source in the scene.

lightColor(=*Vector3r(0.6, 0.6, 0.6)*)
Per-color intensity of primary light (RGB).

lightPos(=*Vector3r(75, 130, 0)*)
Position of OpenGL light source in the scene.

mask(=*~0*, *draw everything*)
Bitmask for showing only bodies where ((mask & *Body::mask*)!=0)

render() → None
Render the scene in the current OpenGL context.

rotScale(=*1.*, *disable scaling*)
Artificially enlarge (scale) rotations of bodies relative to their *reference orientation*, so the they are better visible.

selId(=*Body::ID_NONE*)
Id of particle that was selected by the user.

setRefSe3() → None
Make current positions and orientation reference for scaleDisplacements and scaleRotations.

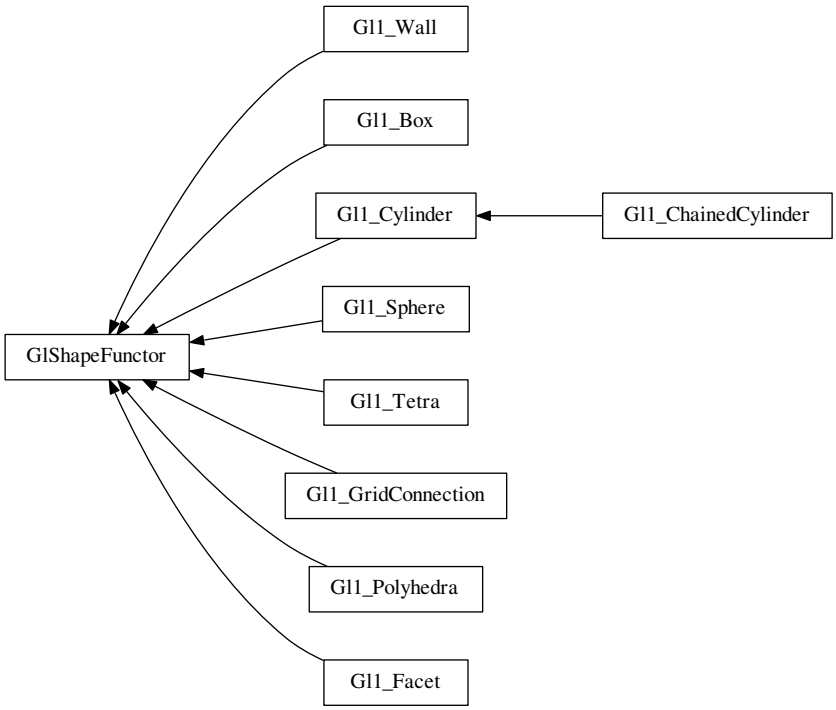
shape(=*true*)
Render body *Shape*

showBody(*(int)id*) → None
Make body visible (see *OpenGLRenderer::hideBody*)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

wire(=*false*)
Render all bodies with wire only (faster)

8.11.2 GlShapeFunctor



class `yade.wrapper.GlShapeFunctor`(*inherits* *Functor* → *Serializable*)
Abstract functor for rendering *Shape* objects.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Gl1_Box`(*inherits* *GlShapeFunctor* → *Functor* → *Serializable*)
Renders *Box* object

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Gl1_ChainedCylinder`(*inherits* *Gl1_Cylinder* → *GlShapeFunctor* → *Functor* → *Serializable*)
Renders *ChainedCylinder* object including a shift for compensating flexion.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`glutNormalize` = **True**

`glutSlices` = **8**

`glutStacks` = **4**

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wire` = **False**

class `yade.wrapper.Gl1_Cylinder`(*inherits* *GlShapeFunctor* → *Functor* → *Serializable*)
Renders *Cylinder* object

`wire`(=*false*) [**static**]
Only show wireframe (controlled by `glutSlices` and `glutStacks`.

`glutNormalize`(=*true*) [**static**]
Fix normals for non-wire rendering

`glutSlices`(=*8*) [**static**]
Number of sphere slices.

`glutStacks`(=*4*) [**static**]
Number of sphere stacks.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`glutNormalize` = **True**

`glutSlices` = **8**

`glutStacks` = **4**

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wire` = **False**

class `yade.wrapper.Gl1_Facet`(*inherits GlShapeFunctor → Functor → Serializable*)
Renders *Facet* object

`normals`(=*false*) [**static**]
In wire mode, render normals of facets and edges; facet's *colors* are disregarded in that case.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`normals` = **False**

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Gl1_GridConnection`(*inherits GlShapeFunctor → Functor → Serializable*)
Renders *Cylinder* object

`wire`(=*false*) [**static**]
Only show wireframe (controlled by `glutSlices` and `glutStacks`.

`glutNormalize`(=*true*) [**static**]
Fix normals for non-wire rendering

`glutSlices`(=*8*) [**static**]
Number of cylinder slices.

`glutStacks`(=*4*) [**static**]
Number of cylinder stacks.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`glutNormalize` = **True**

`glutSlices` = **8**

`glutStacks` = **4**

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wire` = **False**

class `yade.wrapper.Gl1_Polyhedra`(*inherits GlShapeFunctor → Functor → Serializable*)
Renders *Polyhedra* object

`wire`(=*false*) [**static**]
Only show wireframe

`bases`
Ordered list of types (as strings) this functor accepts.

`dict`() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wire` = **False**

class `yade.wrapper.Gl1_Sphere`(*inherits GlShapeFunctor → Functor → Serializable*)
Renders *Sphere* object

`quality`(=*1.0*) [**static**]
Change discretization level of spheres. quality>1 for better image quality, at the price of more cpu/gpu usage, 0<quality<1 for faster rendering. If mono-color spheres are displayed (*Gl1_Sphere::stripes* = False), quality mutiplies *Gl1_Sphere::glutSlices* and *Gl1_Sphere::glutStacks*. If striped spheres are displayed (*Gl1_Sphere::stripes* = True), only integer increments are meaningfull : quality=1 and quality=1.9 will give the same result, quality=2 will give finer result.

`wire`(=*false*) [**static**]
Only show wireframe (controlled by `glutSlices` and `glutStacks`.

`stripes`(=*false*) [**static**]
In non-wire rendering, show stripes clearly showing particle rotation.

`localSpecView`(=*true*) [**static**]
Compute specular light in local eye coordinate system.

`glutSlices`(=*12*) [**static**]
Base number of sphere slices, multiplied by *Gl1_Sphere::quality* before use); not used with `stripes` (see glut{Solid,Wire}Sphere reference)

`glutStacks`(=*6*) [**static**]
Base number of sphere stacks, multiplied by *Gl1_Sphere::quality* before use; not used with `stripes` (see glut{Solid,Wire}Sphere reference)

`circleView`(=*false*) [**static**]
For 2D simulations : display tori instead of spheres, so they will appear like circles if the viewer is looking in the right direction. In this case, remember to disable perspective by pressing "t"-key in the viewer.

`circleRelThickness`(=*0.2*) [**static**]
If *Gl1_Sphere::circleView* is enabled, this is the torus diameter relative to the sphere radius (i.e. the circle relative thickness).

`circleAllowedRotationAxis`(='z') [**static**]
If *Gl1_Sphere::circleView* is enabled, this is the only axis ('x', 'y' or 'z') along which rotation is allowed for the 2D simulation. It allows right orientation of the tori to appear like circles in the viewer. For example, if circleAllowedRotationAxis='x' is set, blockedDOFs="YZ" should also be set for all your particles.

`bases`
Ordered list of types (as strings) this functor accepts.

`circleAllowedRotationAxis` = **'z'**

`circleRelThickness` = **0.2**

`circleView` = **False**

`dict`() → dict
Return dictionary of attributes.

`glutSlices` = **12**

`glutStacks` = **6**

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`localSpecView` = **True**

`quality` = **1.0**

`stripes` = **False**

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`wire` = **False**

class `yade.wrapper.Gl1_Tetra`(*inherits GlShapeFunctor → Functor → Serializable*)
Renders *Tetra* object

`wire`(=*true*) [**static**]
TODO

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

wire = True

class yade.wrapper.**Gl1_Wall**(*inherits GlShapeFunctor → Functor → Serializable*)
Renders *Wall* object

div(=*20*) [**static**]
Number of divisions of the wall inside visible scene part.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

div = 20

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

8.11.3 GlStateFunctor

class yade.wrapper.**GlStateFunctor**(*inherits Functor → Serializable*)
Abstract functor for rendering *State* objects.

bases
Ordered list of types (as strings) this functor accepts.

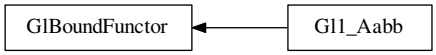
dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

8.11.4 GlBoundFunctor



class `yade.wrapper.GlBoundFunctor`(*inherits Functor → Serializable*)
Abstract functor for rendering *Bound* objects.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Gl1_Aabb`(*inherits GlBoundFunctor → Functor → Serializable*)
Render Axis-aligned bounding box (*Aabb*).

`bases`
Ordered list of types (as strings) this functor accepts.

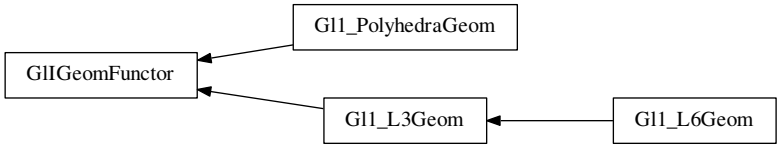
`dict()` → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

8.11.5 GlIGeomFunctor



class `yade.wrapper.GlIGeomFunctor`(*inherits Functor → Serializable*)
Abstract functor for rendering *IGeom* objects.

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label(=uninitalized)`
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs((dict)arg2)` → None
Update object attributes from given dictionary

class `yade.wrapper.Gl1_L3Geom(`*inherits GlIGeomFunctor → Functor → Serializable*`)`
Render *L3Geom* geometry.

`axesLabels(=false)` **[static]**
Whether to display labels for local axes (x,y,z)

`axesScale(=1.)` **[static]**
Scale local axes, their reference length being half of the minimum radius.

`axesWd(=1.)` **[static]**
Width of axes lines, in pixels; not drawn if non-positive

`uPhiWd(=2.)` **[static]**
Width of lines for drawing displacements (and rotations for *L6Geom*); not drawn if non-positive.

`uScale(=1.)` **[static]**
Scale local displacements (*u* - *u0*); 1 means the true scale, 0 disables drawing local displacements; negative values are permissible.

`axesLabels` = **False**

`axesScale` = **1.0**

`axesWd` = **1.0**

`bases`
Ordered list of types (as strings) this functor accepts.

`dict()` → dict
Return dictionary of attributes.

`label(=uninitalized)`
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`uPhiWd` = **2.0**

`uScale` = **1.0**

`updateAttrs((dict)arg2)` → None
Update object attributes from given dictionary

class `yade.wrapper.Gl1_L6Geom(`*inherits Gl1_L3Geom → GlIGeomFunctor → Functor → Serializable*`)`
Render *L6Geom* geometry.

`phiScale(=1.)` **[static]**
Scale local rotations (*phi* - *phi0*). The default scale is to draw π rotation with length equal to minimum radius.

`axesLabels` = **False**

`axesScale` = **1.0**

axesWd = 1.0

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

phiScale = 1.0

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

uPhiWd = 2.0

uScale = 1.0

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Gl1_PolyhedraGeom**(*inherits GlIGeomFunctor → Functor → Serializable*)
Render *PolyhedraGeom* geometry.

bases
Ordered list of types (as strings) this functor accepts.

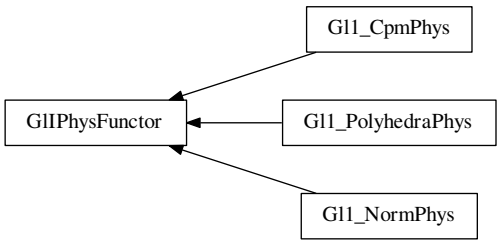
dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

8.11.6 GlIPhysFunctor



class yade.wrapper.**GlIPhysFunctor**(*inherits Functor → Serializable*)
Abstract functor for rendering *IPhys* objects.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Gl1_CpmPhys`(*inherits GlIPhysFunctor → Functor → Serializable*)
Render *CpmPhys* objects of interactions.

`contactLine`(=*true*) [**static**]
Show contact line

`dmgLabel`(=*true*) [**static**]
Numerically show contact damage parameter

`dmgPlane`(=*false*) [**static**]
[what is this?]

`epsT`(=*false*) [**static**]
Show shear strain

`epsTAxes`(=*false*) [**static**]
Show axes of shear plane

`normal`(=*false*) [**static**]
Show contact normal

`colorStrainRatio`(=*-1*) [**static**]
If positive, set the interaction (wire) color based on ε_N normalized by $\varepsilon_0 \times$ *colorStrainRatio* (ε_0 = *CpmPhys.epsCrackOnset*). Otherwise, color based on the residual strength.

`epsNLabel`(=*false*) [**static**]
Numerically show normal strain

`bases`
Ordered list of types (as strings) this functor accepts.

`colorStrainRatio` = **-1.0**

`contactLine` = **True**

`dict`() → dict
Return dictionary of attributes.

`dmgLabel` = **True**

`dmgPlane` = **False**

`epsNLabel` = **False**

`epsT` = **False**

`epsTAxes` = **False**

`label`(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`normal` = **False**

`timingDeltas`
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Gl1_NormPhys`(*inherits GlIPhysFunctor → Functor → Serializable*)

Renders *NormPhys* objects as cylinders of which diameter and color depends on *NormPhys.normalForce* magnitude.

`maxFn`(*=0*) [**static**]
: Value of *NormPhys.normalForce* corresponding to *maxRadius*. This value will be increased (but *not decreased*) automatically.

`signFilter`(*=0*) [**static**]
: If non-zero, only display contacts with negative (-1) or positive (+1) normal forces; if zero, all contacts will be displayed.

`refRadius`(*=std::numeric_limits<Real>::infinity()*) [**static**]
: Reference (minimum) particle radius; used only if *maxRadius* is negative. This value will be decreased (but *not increased*) automatically. *(auto-updated)*

`maxRadius`(*=-1*) [**static**]
: Cylinder radius corresponding to the maximum normal force. If negative, auto-updated *refRadius* will be used instead.

`slices`(*=6*) [**static**]
: Number of sphere slices; (see glutCylinder reference)

`stacks`(*=1*) [**static**]
: Number of sphere stacks; (see glutCylinder reference)

`maxWeakFn`(*=NaN*) [**static**]
: Value that divides contacts by their normal force into the 'weak fabric' and 'strong fabric'. This value is set as side-effect by *utils.fabricTensor*.

`weakFilter`(*=0*) [**static**]
: If non-zero, only display contacts belonging to the 'weak' (-1) or 'strong' (+1) fabric.

`weakScale`(*=1.*) [**static**]
: If *maxWeakFn* is set, scale radius of the weak fabric by this amount (usually smaller than 1). If zero, 1 pixel line is displayed. Colors are not affected by this value.

`bases`
: Ordered list of types (as strings) this functor accepts.

`dict`() → dict
: Return dictionary of attributes.

`label`(*=uninitalized*)
: Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`maxFn` = **0.0**

`maxRadius` = **-1.0**

`maxWeakFn` = **nan**

`refRadius` = **inf**

`signFilter` = **0**

`slices` = **6**

`stacks` = **1**

`timingDeltas`
: Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

`weakFilter` = **0**

`weakScale` = **1.0**

class `yade.wrapper.Gl1_PolyhedraPhys`(*inherits GlIPhysFunctor → Functor → Serializable*)

Renders *PolyhedraPhys* objects as cylinders of which diameter and color depends on *PolyhedraPhys::normForce* magnitude.

`maxFn`(*=0*) [**static**]
: Value of *NormPhys.normalForce* corresponding to *maxDiameter*. This value will be increased (but *not decreased*) automatically.

`refRadius`(*=std::numeric_limits<Real>::infinity()*) [**static**]
: Reference (minimum) particle radius

`signFilter`(*=0*) [**static**]
: If non-zero, only display contacts with negative (-1) or positive (+1) normal forces; if zero, all contacts will be displayed.

`maxRadius`(*=-1*) [**static**]
: Cylinder radius corresponding to the maximum normal force.

`slices`(*=6*) [**static**]
: Number of sphere slices; (see glutCylinder reference)

`stacks`(*=1*) [**static**]
: Number of sphere stacks; (see glutCylinder reference)

`bases`
: Ordered list of types (as strings) this functor accepts.

`dict`() → dict
: Return dictionary of attributes.

`label`(*=uninitalized*)
: Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

`maxFn` = **0.0**

`maxRadius` = **-1.0**

`refRadius` = **inf**

`signFilter` = **0**

`slices` = **6**

`stacks` = **1**

`timingDeltas`
: Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

`updateAttrs`(*(dict)arg2*) → None
: Update object attributes from given dictionary

8.12 Simulation data

8.12.1 Omega

class `yade.wrapper.Omega`

`addScene`() → int
: Add new scene to Omega, returns its number

`bodies`
: Bodies in the current simulation (container supporting index access by id and iteration)

`cell`
: Periodic cell of the current scene (None if the scene is aperiodic).

childClassesNonrecursive(*(str)arg2*) → list
Return list of all classes deriving from given class, as registered in the class factory

disableGdb() → None
Revert SEGV and ABRT handlers to system defaults.

dt
Current timestep (Δt) value.

dynDt
Whether a *TimeStepper* is used for dynamic Δt control. See *dt* on how to enable/disable *TimeStepper*.

dynDtAvailable
Whether a *TimeStepper* is amongst *O.engines*, activated or not.

energy
EnergyTracker of the current simulation. (meaningful only with *O.trackEnergy*)

engines
List of engines in the simulation (corresponds to Scene::engines in C++ source code).

exitNoBacktrace([*(int)status=0*]) → None
Disable SEGV handler and exit, optionally with given status number.

filename
Filename under which the current simulation was saved (None if never saved).

forceSyncCount
Counter for number of syncs in ForceContainer, for profiling purposes.

forces
ForceContainer (forces, torques, displacements) in the current simulation.

interactions
Access to *interactions* of simulation, by using

1.id's of both *Bodies* of the interactions, e.g. `O.interactions[23,65]`

2.iteraction over the whole container:

```
for i in O.interactions: print i.id1,i.id2
```

Note: Iteration silently skips interactions that are not *real*.

isChildClassOf(*(str)arg2*, *(str)arg3*) → bool
Tells whether the first class derives from the second one (both given as strings).

iter
Get current step number

labeledEngine(*(str)arg2*) → object
Return instance of engine/functor with the given label. This function shouldn't be called by the user directly; every ehange in O.engines will assign respective global python variables according to labels.

For example:

O.engines=[InsertionSortCollider(label='collider')]

collider.nBins=5 # collider has become a variable after assignment to O.engines automatically

load(*(str)file*[, *(bool)quiet=False*]) → None
Load simulation from file. The file should be *saved* in the same version of Yade, otherwise compatibility is not guaranteed.

loadTmp([*(str)mark=''*[, *(bool)quiet=False*]]) → None
Load simulation previously stored in memory by saveTmp. *mark* optionally distinguishes multiple saved simulations

lsTmp() → list
Return list of all memory-saved simulations.

materials
Shared materials; they can be accessed by id or by label

miscParams
MiscParams in the simulation (Scene::mistParams), usually used to save serializables that don't fit anywhere else, like GL functors

numThreads
Get maximum number of threads openMP can use.

pause() → None
Stop simulation execution. (May be called from within the loop, and it will stop after the current step).

periodic
Get/set whether the scene is periodic or not (True/False).

plugins() → list
Return list of all plugins registered in the class factory.

realtime
Return clock (human world) time the simulation has been running.

reload([*(bool)quiet=False*]) → None
Reload current simulation

reset() → None
Reset simulations completely (including another scenes!).

resetAllScenes() → None
Reset all scenes.

resetCurrentScene() → None
Reset current scene.

resetThisScene() → None
Reset current scene.

resetTime() → None
Reset simulation time: step number, virtual and real time. (Doesn't touch anything else, including timings).

run([*(int)nSteps=-1*[, *(bool)wait=False*]]) → None
Run the simulation. *nSteps* how many steps to run, then stop (if positive); *wait* will cause not returning to python until simulation will have stopped.

runEngine(*(Engine)arg2*) → None
Run given engine exactly once; simulation time, step number etc. will not be incremented (use only if you know what you do).

running
Whether background thread is currently running a simulation.

save(*(str)file*[, *(bool)quiet=False*]) → None
Save current simulation to file (should be .xml or .xml.bz2 or .yade or .yade.gz). .xml files are bigger than .yade, but can be more or less easily (due to their size) opened and edited, e.g. with text editors. .bz2 and .gz correspond both to compressed versions. All saved files should be *loaded* in the same version of Yade, otherwise compatibility is not guaranteed.

saveTmp([*(str)mark=''*[, *(bool)quiet=False*]]) → None
Save simulation to memory (disappears at shutdown), can be loaded later with loadTmp. *mark* optionally distinguishes different memory-saved simulations.

sceneToString() → str
Return the entire scene as a string. Equivalent to using O.save(...) except that the scene goes to a string instead of a file. (see also stringToScene())

speed
Return current calculation speed [iter/sec].

step() → None
Advance the simulation by one step. Returns after the step will have finished.

stopAtIter
Get/set number of iteration after which the simulation will stop.

stopAtTime
Get/set time after which the simulation will stop.

stringToScene(*(str)arg2*[, *(str)mark=''*]) → None
Load simulation from a string passed as argument (see also sceneToString).

subStep
Get the current subStep number (only meaningful if O.subStepping==True); -1 when outside the loop, otherwise either 0 (O.subStepping==False) or number of engine to be run (O.subStepping==True)

subStepping
Get/set whether subStepping is active.

switchScene() → None
Switch to alternative simulation (while keeping the old one). Calling the function again switches back to the first one. Note that most variables from the first simulation will still refer to the first simulation even after the switch (e.g. b=O.bodies[4]; O.switchScene(); [b still refers to the body in the first simulation here])

switchToScene(*(int)arg2*) → None
Switch to defined scene. Default scene has number 0, other scenes have to be created by addScene method.

tags
Tags (string=string dictionary) of the current simulation (container supporting string-index access/assignment)

thisScene
Return current scene's id.

time
Return virtual (model world) time of the simulation.

timingEnabled
Globally enable/disable timing services (see documentation of the *timing module*).

tmpFilename() → str
Return unique name of file in temporary directory which will be deleted when yade exits.

tmpToFile(*(str)fileName*[, *(str)mark=''*]) → None
Save XML of *saveTmp*'d simulation into *fileName*.

tmpToString([*(str)mark=''*]) → str
Return XML of *saveTmp*'d simulation as string.

trackEnergy
When energy tracking is enabled or disabled in this simulation.

wait() → None
Don't return until the simulation will have been paused. (Returns immediately if not running).

8.12.2 BodyContainer

class `yade.wrapper.BodyContainer`

`__init__`(*(BodyContainer)arg2*) → None

`addToClump`(*(object)arg2*, *(int)arg3*[, *(int)discretization=0*]) → None
Add body b (or a list of bodies) to an existing clump c. c must be clump and b may not be a clump member of c. Clump masses and inertia are adapted automatically (for details see *clump()*).

See examples/clumps/addToClump-example.py for an example script.

Note: If b is a clump itself, then all members will be added to c and b will be deleted. If b is a clump member of clump d, then all members from d will be added to c and d will be deleted. If you need to add just clump member b, *release* this member from d first.

`append`(*(Body)arg2*) → int

Append one Body instance, return its id.

append((BodyContainer)arg1, (object)arg2) → object : Append list of Body instance, return list of ids

`appendClumped`(*(object)arg2*[, *(int)discretization=0*]) → tuple
Append given list of bodies as a clump (rigid aggregate); returns a tuple of `(clumpId,[memberId1,memberId2,...])`. Clump masses and inertia are adapted automatically (for details see *clump()*).

`clear`() → None
Remove all bodies (interactions not checked)

`clump`(*(object)arg2*[, *(int)discretization=0*]) → int
Clump given bodies together (creating a rigid aggregate); returns `clumpId`. Clump masses and inertia are adapted automatically when discretization>0. If clump members are overlapping this is done by integration/summation over mass points using a regular grid of cells (grid cells length is defined as $\mathtt{R}_{\mathrm{min}}/\mathtt{discretization}$, where $\mathtt{R}_{\mathrm{min}}$ is minimum clump member radius). For non-overlapping members inertia of the clump is the sum of inertias from members. If discretization<=0 sum of inertias from members is used (faster, but inaccurate).

`erase`(*(int)arg2*[, *(bool)eraseClumpMembers=0*]) → bool
Erase body with the given id; all interaction will be deleted by InteractionLoop in the next step. If a clump is erased use *O.bodies.erase(clumpId,True)* to erase the clump AND its members.

`getRoundness`([*(list)excludeList=[]*]) → float
Returns roundness coefficient RC = R2/R1. R1 is the equivalent sphere radius of a clump. R2 is the minimum radius of a sphere, that imbeds the clump. If just spheres are present RC = 1. If clumps are present 0 < RC < 1. Bodies can be excluded from the calculation by giving a list of ids: *O.bodies.getRoundness([ids])*.

See examples/clumps/replaceByClumps-example.py for an example script.

`releaseFromClump`(*(int)arg2*, *(int)arg3*[, *(int)discretization=0*]) → None
Release body b from clump c. b must be a clump member of c. Clump masses and inertia are adapted automatically (for details see *clump()*).

See examples/clumps/releaseFromClump-example.py for an example script.

Note: If c contains only 2 members b will not be released and a warning will appear. In this case clump c should be *erased*.

`replace`(*(object)arg2*) → object

replaceByClumps(*(list)arg2*, *(object)arg3*[, *(int)discretization=0*]) → list
Replace spheres by clumps using a list of clump templates and a list of amounts; returns a list of tuples: `[(clumpId1,[memberId1,memberId2,...]),(clumpId2,[memberId1,memberId2,...]),...]`. A new clump will have the same volume as the sphere, that was replaced. Clump masses and inertia are adapted automatically (for details see *clump()*).

O.bodies.replaceByClumps([utils.clumpTemplate([1,1],[.5,.5])] , [.9]) #will replace 90 % of all standalone spheres by 'dyads'

See examples/clumps/replaceByClumps-example.py for an example script.

updateClumpProperties([*(list)excludeList=[]*[, *(int)discretization=5*]]) → None
Manually force Yade to update clump properties mass, volume and inertia (for details of 'discretization' value see *clump()*). Can be used, when clumps are modified or erased during a simulation. Clumps can be excluded from the calculation by giving a list of ids: *O.bodies.updateProperties([ids])*.

8.12.3 InteractionContainer

class yade.wrapper.InteractionContainer
Access to *interactions* of simulation, by using

1.id's of both *Bodies* of the interactions, e.g. `O.interactions[23,65]`

2.iteration over the whole container:

```
for i in O.interactions: print i.id1,i.id2
```

Note: Iteration silently skips interactions that are not *real*.

__init__(*(InteractionContainer)arg2*) → None

all([*(bool)onlyReal=False*]) → list
Return list of all interactions. Virtual interaction are filtered out if onlyReal=True, else (default) it dumps the full content.

clear() → None
Remove all interactions, and invalidate persistent collider data (if the collider supports it).

countReal() → int
Return number of interactions that are "real", i.e. they have phys and geom.

erase(*(int)arg2*, *(int)arg3*) → None
Erase one interaction, given by id1, id2 (internally, `requestErase` is called – the interaction might still exist as potential, if the *Collider* decides so).

eraseNonReal() → None
Erase all interactions that are not *real* .

has(*(int)arg2*, *(int)arg3*) → bool
Tell if a pair of ids corresponds to an existing interaction (real or not)

nth(*(int)arg2*) → Interaction
Return n-th interaction from the container (usable for picking random interaction).

serializeSorted

withBody(*(int)arg2*) → list
Return list of real interactions of given body.

withBodyAll(*(int)arg2*) → list
Return list of all (real as well as non-real) interactions of given body.

8.12.4 ForceContainer

class yade.wrapper.**ForceContainer**

__init__(*(ForceContainer)arg2*) → None

addF(*(int)id*, *(Vector3)f*[, *(bool)permanent=False*]) → None
Apply force on body (accumulates).

If permanent=false (default), the force applies for one iteration, then it is reset by ForceResetter. # If permanent=true, it persists over iterations, until it is overwritten by another call to addF(id,f,True) or removed by reset(resetAll=True). The permanent force on a body can be checked with permF(id).

addMove(*(int)id*, *(Vector3)m*) → None
Apply displacement on body (accumulates).

addRot(*(int)id*, *(Vector3)r*) → None
Apply rotation on body (accumulates).

addT(*(int)id*, *(Vector3)t*[, *(bool)permanent=False*]) → None
Apply torque on body (accumulates).

If permanent=false (default), the torque applies for one iteration, then it is reset by ForceResetter. # If permanent=true, it persists over iterations, until it is overwritten by another call to addT(id,f,True) or removed by reset(resetAll=True). The permanent torque on a body can be checked with permT(id).

f(*(int)id*[, *(bool)sync=False*]) → Vector3
Force applied on body. For clumps in openMP, synchronize the force container with sync=True, else the value will be wrong.

getPermForceUsed() → bool
Check wether permanent forces are present.

m(*(int)id*[, *(bool)sync=False*]) → Vector3
Deprecated alias for t (torque).

move(*(int)id*) → Vector3
Displacement applied on body.

permF(*(int)id*) → Vector3
read the value of permanent force on body (set with setPermF()).

permT(*(int)id*) → Vector3
read the value of permanent torque on body (set with setPermT()).

reset([*(bool)resetAll=True*]) → None
Reset the force container, including user defined permanent forces/torques. resetAll=False will keep permanent forces/torques unchanged.

rot(*(int)id*) → Vector3
Rotation applied on body.

syncCount
Number of synchronizations of ForceContainer (cummulative); if significantly higher than number of steps, there might be unnecessary syncs hurting performance.

t(*(int)id*[, *(bool)sync=False*]) → Vector3
Torque applied on body. For clumps in openMP, synchronize the force container with sync=True, else the value will be wrong.

8.12.5 MaterialContainer

class `yade.wrapper.MaterialContainer`

Container for *Materials*. A material can be accessed using

1. numerical index in range(0,len(cont)), like cont[2];
2. textual label that was given to the material, like cont['steel']. This etails traversing all materials and should not be used frequently.

`__init__`(*(MaterialContainer)arg2*) → None

`append`(*(Material)arg2*) → int

Add new shared *Material*; changes its id and return it.

append((MaterialContainer)arg1, (object)arg2) → object : Append list of *Material* instances, return list of ids.

`index`(*(str)arg2*) → int

Return id of material, given its label.

8.12.6 Scene

class `yade.wrapper.Scene`(*inherits Serializable*)

Object comprising the whole simulation.

`compressionNegative`

Whether the convention is that compression has negative sign (set by *IGeomFunctor*).

`dict`() → dict

Return dictionary of attributes.

`doSort`(*=false*)

Used, when new body is added to the scene.

`dt`(*=1e-8*)

Current timestep for integration.

`flags`(*=0*)

Various flags of the scene; 1 (Scene::LOCAL_COORDS): use local coordinate system rather than global one for per-interaction quantities (set automatically from the functor).

`isPeriodic`(*=false*)

Whether periodic boundary conditions are active.

`iter`(*=0*)

Current iteration (computational step) number

`localCoords`

Whether local coordianate system is used on interactions (set by *IGeomFunctor*).

`selectedBody`(*=-1*)

Id of body that is selected by the user

`speed`(*=0*)

Current calculation speed [iter/s]

`stopAtIter`(*=0*)

Iteration after which to stop the simulation.

`stopAtTime`(*=0*)

Time after which to stop the simulation

`subStep`(*=-1*)

Number of sub-step; not to be changed directly. -1 means to run loop prologue (cell integration), 0…n-1 runs respective engines (n is number of engines), n runs epilogue (increment step number and time.

`subStepping`(=*false*)
Whether we currently advance by one engine in every step (rather than by single run through all engines).

`tags`(=*uninitalized*)
Arbitrary key=value associations (tags like mp3 tags: author, date, version, description etc.)

`time`(=*0*)
Simulation time (virtual time) [s]

`trackEnergy`(=*false*)
Whether energies are being traced.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

8.12.7 Cell

class `yade.wrapper.Cell`(*inherits Serializable*)
Parameters of periodic boundary conditions. Only applies if O.isPeriodic==True.

`dict`() → dict
Return dictionary of attributes.

`getDefGrad`() → Matrix3
Returns deformation gradient tensor $\mathbf{F}$ of the cell deformation (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getEulerianAlmansiStrain`() → Matrix3
Returns Eulerian-Almansi strain tensor $\boldsymbol{e} = \frac{1}{2}(\mathbf{I} - \mathbf{b}^{-1}) = \frac{1}{2}(\mathbf{I} - (\mathbf{F}\mathbf{F}^{\mathsf{T}})^{-1})$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getLCauchyGreenDef`() → Matrix3
Returns left Cauchy-Green deformation tensor $\mathbf{b} = \mathbf{F}\mathbf{F}^{\mathsf{T}}$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getLagrangianStrain`() → Matrix3
Returns Lagrangian strain tensor $\mathbf{E} = \frac{1}{2}(\mathbf{C} - \mathbf{I}) = \frac{1}{2}(\mathbf{F}^{\mathsf{T}}\mathbf{F} - \mathbf{I}) = \frac{1}{2}(\mathbf{U}^2 - \mathbf{I})$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getLeftStretch`() → Matrix3
Returns left (spatial) stretch tensor of the cell (matrix $\mathbf{U}$ from polar decomposition $\mathbf{F} = \mathbf{R}\mathbf{U}$)

`getPolarDecOfDefGrad`() → tuple
Returns orthogonal matrix $\mathbf{R}$ and symmetric positive semi-definite matrix $\mathbf{U}$ as polar decomposition of deformation gradient $\mathbf{F}$ of the cell ($\mathbf{F} = \mathbf{R}\mathbf{U}$)

`getRCauchyGreenDef`() → Matrix3
Returns right Cauchy-Green deformation tensor $\mathbf{C} = \mathbf{F}^{\mathsf{T}}\mathbf{F}$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getRightStretch`() → Matrix3
Returns right (material) stretch tensor of the cell (matrix $\mathbf{V}$ from polar decomposition $\mathbf{F} = \mathbf{R}\mathbf{U} = \mathbf{V}\mathbf{R} \rightarrow \mathbf{V} = \mathbf{F}\mathbf{R}^{\mathsf{T}}$)

`getRotation`() → Matrix3
Returns rotation of the cell (orthogonal matrix $\mathbf{R}$ from polar decomposition $\mathbf{F} = \mathbf{R}\mathbf{U}$)

`getSmallStrain`() → Matrix3
Returns small strain tensor $\boldsymbol{\varepsilon} = \frac{1}{2}(\mathbf{F}+\mathbf{F}^{\mathsf{T}})-\mathbf{I}$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getSpin`() → Vector3
Returns the spin defined by the skew symmetric part of *velGrad*

hSize
Base cell vectors (columns of the matrix), updated at every step from *velGrad* (*trsf* accumulates applied *velGrad* transformations). Setting *hSize* during a simulation is not supported by most contact laws, it is only meant to be used at iteration 0 before any interactions have been created.

hSize0
Value of untransformed hSize, with respect to current *trsf* (computed as *trsf* ^{1} × *hSize*.

homoDeform(=*2*)
If >0, deform (*velGrad*) the cell homothetically by adjusting positions and velocities of bodies. The velocity change is obtained by deriving the expression v= v.x, where v is the macroscopic velocity gradient, giving in an incremental form: Δv=Δ v x + v Δx. As a result, velocities are modified as soon as **velGrad** changes, according to the first term: Δv(t)=Δ v x(t), while the 2nd term reflects a convective term: Δv'= v v(t-dt/2). The second term is neglected if homoDeform=1. All terms are included if homoDeform=2 (default)

nextVelGrad(=*Matrix3r::Zero()*)
see *Cell.velGrad*.

prevHSize(=*Matrix3r::Identity()*)
hSize from the previous step, used in the definition of relative velocity across periods.

prevVelGrad(=*Matrix3r::Zero()*)
Velocity gradient in the previous step.

refHSize(=*Matrix3r::Identity()*)
Reference cell configuration, only used with *OpenGLRenderer.dispScale*. Updated automatically when *hSize* or *trsf* is assigned directly; also modified by *utils.setRefSe3* (called e.g. by the **Reference** button in the UI).

refSize
Reference size of the cell (lengths of initial cell vectors, i.e. column norms of *hSize*).

Note: Modifying this value is deprecated, use *setBox* instead.

---

setBox(*(Vector3)arg2*) → None

Set *Cell* shape to be rectangular, with dimensions along axes specified by given argument. Shorthand for assigning diagonal matrix with respective entries to *hSize*.

setBox((Cell)arg1, (float)arg2, (float)arg3, (float)arg4) → None : Set *Cell* shape to be rectangular, with dimensions along x, y, z specified by arguments. Shorthand for assigning diagonal matrix with the respective entries to *hSize*.

shearPt(*(Vector3)arg2*) → Vector3
Apply shear (cell skew+rot) on the point

shearTrsf
Current skew+rot transformation (no resize)

size
Current size of the cell, i.e. lengths of the 3 cell lateral vectors contained in *Cell.hSize* columns. Updated automatically at every step.

trsf
Current transformation matrix of the cell, obtained from time integration of *Cell.velGrad*.

unshearPt(*(Vector3)arg2*) → Vector3
Apply inverse shear on the point (removes skew+rot of the cell)

unshearTrsf
Inverse of the current skew+rot transformation (no resize)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

`velGrad`
Velocity gradient of the transformation; used in *NewtonIntegrator*. Values of *velGrad* accumulate in *trsf* at every step.

NOTE: changing velGrad at the beginning of a simulation loop would lead to inaccurate integration for one step, as it should normaly be changed after the contact laws (but before Newton). To avoid this problem, assignment is deferred automatically. The target value typed in terminal is actually stored in *Cell.nextVelGrad* and will be applied right in time by Newton integrator.

Note: Assigning individual components of velGrad is not possible (it will not return any error but it will have no effect). Instead, you can assign to *Cell.nextVelGrad*, as in O.cell.nextVelGrad[1,2]=1.

`velGradChanged(=false)`
true when velGrad has been changed manually (see also *Cell.nextVelGrad*)

`volume`
Current volume of the cell.

`wrap(`*(Vector3)arg2*`)` → Vector3
Transform an arbitrary point into a point in the reference cell

`wrapPt(`*(Vector3)arg2*`)` → Vector3
Wrap point inside the reference cell, assuming the cell has no skew+rot.

8.13 Other classes

class `yade.wrapper.TimingDeltas`

`data`
Get timing data as list of tuples (label, execTime[nsec], execCount) (one tuple per checkpoint)

`reset()` → None
Reset timing information

class `yade.wrapper.GlShapeDispatcher(`*inherits Dispatcher → Engine → Serializable*`)`
Dispatcher calling *functors* based on received argument type(s).

`dead(=false)`
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict()` → dict
Return dictionary of attributes.

`dispFunctor(`*(Shape)arg2*`)` → GlShapeFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

`dispMatrix(`[*(bool)names=True*]`)` → dict
Return dictionary with contents of the dispatch matrix.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`functors`
Functors associated with this dispatcher.

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.LBMlink`(*inherits Serializable*)
Link class for Lattice Boltzmann Method

`DistMid`(=*Vector3r::Zero()*)
Distance between middle of the link and mass center of body

`PointingOutside`(=*false*)
True if it is a link pointing outside to the system (from a fluid or solid node)

`VbMid`(=*Vector3r::Zero()*)
Velocity of boundary at midpoint

`ct`(=*0.*)
Coupling term in modified bounce back rule

`dict`() → dict
Return dictionary of attributes.

`fid`(=*-1*)
Fluid node identifier

`i`(=*-1*)
direction index of the link

`idx_sigma_i`(=*-1*)
sigma_i direction index (Fluid->Solid)

`isBd`(=*false*)
True if it is a boundary link

`nid1`(=*-1*)
fixed node identifier

`nid2`(=*-1*)
fixed node identifier or -1 if node points outside

`sid`(=*-1*)
Solid node identifier

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.GlExtra_LawTester`(*inherits GlExtraDrawer → Serializable*)
Find an instance of *LawTester* and show visually its data.

`dead`(=*false*)
Deactivate the object (on error/exception).

`dict`() → dict
Return dictionary of attributes.

`tester`(=*uninitalized*)
Associated *LawTester* object.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.MatchMaker`(*inherits Serializable*)
Class matching pair of ids to return pre-defined (for a pair of ids defined in *matches*) or derived value (computed using *algo*) of a scalar parameter. It can be called (`id1`, `id2`, `val1=NaN`, `val2=NaN`) in both python and c++.

Note: There is a *converter* from python number defined for this class, which creates a new *MatchMaker* returning the value of that number; instead of giving the object instance therefore, you can only pass the number value and it will be converted automatically.

`algo`
Alogorithm used to compute value when no match for ids is found. Possible values are

- 'avg' (arithmetic average)
- 'min' (minimum value)
- 'max' (maximum value)
- 'harmAvg' (harmonic average)

The following algo algorithms do *not* require meaningful input values in order to work:

- 'val' (return value specified by *val*)
- 'zero' (always return 0.)

`computeFallback`(*(float)val1*, *(float)val2*) → float
Compute algo value for *val1* and *val2*, using algorithm specified by *algo*.

`dict`() → dict
Return dictionary of attributes.

`matches`(=*uninitalized*)
Array of `(id1,id2,value)` items; queries matching `id1` + `id2` or `id2` + `id1` will return `value`

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

`val`(=*NaN*)
Constant value returned if there is no match and *algo* is `val`

class `yade.wrapper.GlBoundDispatcher`(*inherits Dispatcher* → *Engine* → *Serializable*)
Dispatcher calling *functors* based on received argument type(s).

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`dispFunctor`(*(Bound)arg2*) → GlBoundFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

`dispMatrix`(*[(bool)names=True]*) → dict
Return dictionary with contents of the dispatch matrix.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

functors
Functors associated with this dispatcher.

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**EnergyTracker**(*inherits Serializable*)
Storage for tracing energies. Only to be used if *O.trackEnergy* is True.

clear() → None
Clear all stored values.

dict() → dict
Return dictionary of attributes.

energies(=*uninitalized*)
Energy values, in linear array

items() → list
Return contents as list of (name,value) tuples.

keys() → list
Return defined energies.

total() → float
Return sum of all energies.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Engine**(*inherits Serializable*)
Basic execution unit of simulation, called from the simulation loop (O.engines)

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**LBMnode**(*inherits Serializable*)
Node class for Lattice Boltzmann Method

dict() → dict
Return dictionary of attributes.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**GlIGeomDispatcher**(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling *functors* based on received argument type(s).

dead(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

dispFunctor(*(IGeom)arg2*) → GlIGeomFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix([*(bool)names=True*]) → dict
Return dictionary with contents of the dispatch matrix.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

functors
Functors associated with this dispatcher.

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(*=-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.ParallelEngine`(*inherits Engine → Serializable*)
Engine for running other Engine in parallel.

`__init__`() → None
object ___init___(tuple args, dict kwds)

___init___((list)arg2) → object : Construct from (possibly nested) list of slaves.

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`slaves`
List of lists of Engines; each top-level group will be run in parallel with other groups, while Engines inside each group will be run sequentially, in given order.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.LBMbody`(*inherits Serializable*)
Body class for Lattice Boltzmann Method

`AVel`(=*Vector3r::Zero()*)
Angular velocity of body

`Fh`(=*Vector3r::Zero()*)
Hydrodynamical force on body

`Mh`(=*Vector3r::Zero()*)
Hydrodynamical momentum on body

`dict`() → dict
Return dictionary of attributes.

`fm`(=*Vector3r::Zero()*)
Hydrodynamic force (LB unit) at t-0.5dt

`force`(=*Vector3r::Zero()*)
Hydrodynamic force, need to be reinitialized (LB unit)

`fp`(=*Vector3r::Zero()*)
Hydrodynamic force (LB unit) at t+0.5dt

`isEroded`(=*false*)
Hydrodynamical force on body

mm(=*Vector3r::Zero()*)
Hydrodynamic momentum (LB unit) at t-0.5dt

momentum(=*Vector3r::Zero()*)
Hydrodynamic momentum,need to be reinitialized (LB unit)

mp(=*Vector3r::Zero()*)
Hydrodynamic momentum (LB unit) at t+0.5dt

pos(=*Vector3r::Zero()*)
Position of body

radius(=*-1000.*)
Radius of body (for sphere)

saveProperties(=*false*)
To save properties of the body

type(=*-1*)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

vel(=*Vector3r::Zero()*)
Velocity of body

class yade.wrapper.**Functor**(*inherits Serializable*)
Function-like object that is called by Dispatcher, if types of arguments match those the Functor declares to accept.

bases
Ordered list of types (as strings) this functor accepts.

dict() → dict
Return dictionary of attributes.

label(=*uninitalized*)
Textual label for this object; must be a valid python identifier, you can refer to it directly from python.

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**Serializable**

dict() → dict
Return dictionary of attributes.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**GlStateDispatcher**(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling *functors* based on received argument type(s).

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

dispFunctor(*(State)arg2*) → GlStateFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix([*(bool)names=True*]) → dict
Return dictionary with contents of the dispatch matrix.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

functors
Functors associated with this dispatcher.

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

timingDeltas
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**GlIPhysDispatcher**(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling *functors* based on received argument type(s).

dead(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

dict() → dict
Return dictionary of attributes.

dispFunctor(*(IPhys)arg2*) → GlIPhysFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix([*(bool)names=True*]) → dict
Return dictionary with contents of the dispatch matrix.

execCount
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

execTime
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

functors
Functors associated with this dispatcher.

label(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

ompThreads(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.GlExtra_OctreeCubes`(*inherits GlExtraDrawer → Serializable*)
Render boxed read from file

`boxesFile`(=*uninitalized*)
File to read boxes from; ascii files with `x0 y0 z0 x1 y1 z1 c` records, where `c` is an integer specifying fill (0 for wire, 1 for filled).

`dead`(=*false*)
Deactivate the object (on error/exception).

`dict`() → dict
Return dictionary of attributes.

`fillRangeDraw`(=*Vector2i(-2, 2)*)
Range of fill indices that will be rendered.

`fillRangeFill`(=*Vector2i(2, 2)*)
Range of fill indices that will be filled.

`levelRangeDraw`(=*Vector2i(-2, 2)*)
Range of levels that will be rendered.

`noFillZero`(=*true*)
Do not fill 0-fill boxed (those that are further subdivided)

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Dispatcher`(*inherits Engine → Serializable*)
Engine dispatching control to its associated functors, based on types of argument it receives. This abstract base class provides no functionality in itself.

`dead`(=*false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`dict`() → dict
Return dictionary of attributes.

`execCount`
Cummulative count this engine was run (only used if *O.timingEnabled*==`True`).

`execTime`
Cummulative time this Engine took to run (only used if *O.timingEnabled*==`True`).

`label`(=*uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`ompThreads`(=*-1*)
Number of threads to be used in the engine. If ompThreads<0 (default), the number will be typically OMP_NUM_THREADS or the number N defined by 'yade -jN' (this behavior can depend on the engine though). This attribute will only affect engines whose code includes openMP parallel regions (e.g. *InteractionLoop*). This attribute is mostly useful for experiments or when combining *ParallelEngine* with engines that run parallel regions, resulting in nested OMP loops with different number of threads at each level.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and *O.timingEnabled*==`True`.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

class `yade.wrapper.Cell`(*inherits Serializable*)
Parameters of periodic boundary conditions. Only applies if O.isPeriodic==True.

`dict`() → dict
Return dictionary of attributes.

`getDefGrad`() → Matrix3
Returns deformation gradient tensor $\mathbf{F}$ of the cell deformation (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getEulerianAlmansiStrain`() → Matrix3
Returns Eulerian-Almansi strain tensor $\boldsymbol{e} = \frac{1}{2}(\mathbf{I} - \mathbf{b}^{-1}) = \frac{1}{2}(\mathbf{I} - (\mathbf{F}\mathbf{F}^{\mathsf{T}})^{-1})$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getLCauchyGreenDef`() → Matrix3
Returns left Cauchy-Green deformation tensor $\mathbf{b} = \mathbf{F}\mathbf{F}^{\mathsf{T}}$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getLagrangianStrain`() → Matrix3
Returns Lagrangian strain tensor $\mathbf{E} = \frac{1}{2}(\mathbf{C} - \mathbf{I}) = \frac{1}{2}(\mathbf{F}^{\mathsf{T}}\mathbf{F} - \mathbf{I}) = \frac{1}{2}(\mathbf{U}^2 - \mathbf{I})$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getLeftStretch`() → Matrix3
Returns left (spatial) stretch tensor of the cell (matrix $\mathbf{U}$ from polar decomposition $\mathbf{F} = \mathbf{R}\mathbf{U}$)

`getPolarDecOfDefGrad`() → tuple
Returns orthogonal matrix $\mathbf{R}$ and symmetric positive semi-definite matrix $\mathbf{U}$ as polar decomposition of deformation gradient $\mathbf{F}$ of the cell ($\mathbf{F} = \mathbf{R}\mathbf{U}$)

`getRCauchyGreenDef`() → Matrix3
Returns right Cauchy-Green deformation tensor $\mathbf{C} = \mathbf{F}^{\mathsf{T}}\mathbf{F}$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getRightStretch`() → Matrix3
Returns right (material) stretch tensor of the cell (matrix $\mathbf{V}$ from polar decomposition $\mathbf{F} = \mathbf{R}\mathbf{U} = \mathbf{V}\mathbf{R} \rightarrow \mathbf{V} = \mathbf{F}\mathbf{R}^{\mathsf{T}}$)

`getRotation`() → Matrix3
Returns rotation of the cell (orthogonal matrix $\mathbf{R}$ from polar decomposition $\mathbf{F} = \mathbf{R}\mathbf{U}$)

`getSmallStrain`() → Matrix3
Returns small strain tensor $\boldsymbol{\varepsilon} = \frac{1}{2}(\mathbf{F}+\mathbf{F}^{\mathsf{T}})-\mathbf{I}$ of the cell (http://en.wikipedia.org/wiki/Finite_strain_theory)

`getSpin`() → Vector3
Returns the spin defined by the skew symmetric part of *velGrad*

`hSize`
Base cell vectors (columns of the matrix), updated at every step from *velGrad* (*trsf* accumulates applied *velGrad* transformations). Setting *hSize* during a simulation is not supported by most contact laws, it is only meant to be used at iteration 0 before any interactions have been created.

`hSize0`
Value of untransformed hSize, with respect to current *trsf* (computed as $trsf^{-1} \times hSize$.

`homoDeform`(*=2*)
If >0, deform (*velGrad*) the cell homothetically by adjusting positions and velocities of bodies. The velocity change is obtained by deriving the expression v= v.x, where v is the macroscopic velocity gradient, giving in an incremental form: Δv=Δ v x + v Δx. As a result, velocities are modified as soon as `velGrad` changes, according to the first term: Δv(t)=Δ v x(t), while the 2nd term reflects a convective term: Δv'= v v(t-dt/2). The second term is neglected if homoDeform=1. All terms are included if homoDeform=2 (default)

nextVelGrad(=*Matrix3r::Zero()*)
see *Cell.velGrad*.

prevHSize(=*Matrix3r::Identity()*)
hSize from the previous step, used in the definition of relative velocity across periods.

prevVelGrad(=*Matrix3r::Zero()*)
Velocity gradient in the previous step.

refHSize(=*Matrix3r::Identity()*)
Reference cell configuration, only used with *OpenGLRenderer.dispScale*. Updated automatically when *hSize* or *trsf* is assigned directly; also modified by *utils.setRefSe3* (called e.g. by the `Reference` button in the UI).

refSize
Reference size of the cell (lengths of initial cell vectors, i.e. column norms of *hSize*).

Note: Modifying this value is deprecated, use *setBox* instead.

---

setBox(*(Vector3)arg2*) → None

Set *Cell* shape to be rectangular, with dimensions along axes specified by given argument. Shorthand for assigning diagonal matrix with respective entries to *hSize*.

setBox((Cell)arg1, (float)arg2, (float)arg3, (float)arg4) → None : Set *Cell* shape to be rectangular, with dimensions along `x`, `y`, `z` specified by arguments. Shorthand for assigning diagonal matrix with the respective entries to *hSize*.

shearPt(*(Vector3)arg2*) → Vector3
Apply shear (cell skew+rot) on the point

shearTrsf
Current skew+rot transformation (no resize)

size
Current size of the cell, i.e. lengths of the 3 cell lateral vectors contained in *Cell.hSize* columns. Updated automatically at every step.

trsf
Current transformation matrix of the cell, obtained from time integration of *Cell.velGrad*.

unshearPt(*(Vector3)arg2*) → Vector3
Apply inverse shear on the point (removes skew+rot of the cell)

unshearTrsf
Inverse of the current skew+rot transformation (no resize)

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

velGrad
Velocity gradient of the transformation; used in *NewtonIntegrator*. Values of *velGrad* accumulate in *trsf* at every step.

NOTE: changing velGrad at the beginning of a simulation loop would lead to inacurate integration for one step, as it should normaly be changed after the contact laws (but before Newton). To avoid this problem, assignment is deferred automatically. The target value typed in terminal is actually stored in *Cell.nextVelGrad* and will be applied right in time by Newton integrator.

Note: Assigning individual components of velGrad is not possible (it will not return any error but it will have no effect). Instead, you can assign to *Cell.nextVelGrad*, as in O.cell.nextVelGrad[1,2]=1.

---

velGradChanged(=*false*)
true when velGrad has been changed manually (see also *Cell.nextVelGrad*)

`volume`
Current volume of the cell.

`wrap`(*(Vector3)arg2*) → Vector3
Transform an arbitrary point into a point in the reference cell

`wrapPt`(*(Vector3)arg2*) → Vector3
Wrap point inside the reference cell, assuming the cell has no skew+rot.

class `yade.wrapper.GlExtraDrawer`(*inherits Serializable*)
Performing arbitrary OpenGL drawing commands; called from *OpenGLRenderer* (see *OpenGLRenderer.extraDrawers*) once regular rendering routines will have finished.

This class itself does not render anything, derived classes should override the *render* method.

`dead`(=*false*)
Deactivate the object (on error/exception).

`dict`() → dict
Return dictionary of attributes.

`updateAttrs`(*(dict)arg2*) → None
Update object attributes from given dictionary

Chapter 9

Yade modules

9.1 yade.bodiesHandling module

Miscellaneous functions, which are useful for handling bodies.

`yade.bodiesHandling.facetsDimensions`(*idFacets=[]*, *mask=-1*)
The function accepts the list of facet id's or list of facets and calculates max and min dimensions, geometrical center.

Parameters

- `idFacets` (*list*) – list of spheres
- `mask` (*int*) – *Body.mask* for the checked bodies

Returns dictionary with keys `min` (minimal dimension, Vector3), `max` (maximal dimension, Vector3), `minId` (minimal dimension facet Id, Vector3), `maxId` (maximal dimension facet Id, Vector3), `center` (central point of bounding box, Vector3), `extends` (sizes of bounding box, Vector3), `number` (number of facets, int),

`yade.bodiesHandling.sphereDuplicate`(*idSphere*)
The functions makes a copy of sphere

`yade.bodiesHandling.spheresModify`(*idSpheres=[]*, *mask=-1*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, *orientation=Quaternion((1, 0, 0), 0)*, *copy=False*)
The function accepts the list of spheres id's or list of bodies and modifies them: rotating, scaling, shifting. if copy=True copies bodies and modifies them. Also the mask can be given. If idSpheres not empty, the function affects only bodies, where the mask passes. If idSpheres is empty, the function search for bodies, where the mask passes.

Parameters

- `shift` (*Vector3*) – Vector3(X,Y,Z) parameter moves spheres.
- `scale` (*float*) – factor scales given spheres.
- `orientation` (*Quaternion*) – orientation of spheres
- `mask` (*int*) – *Body.mask* for the checked bodies

Returns list of bodies if copy=True, and Boolean value if copy=False

`yade.bodiesHandling.spheresPackDimensions`(*idSpheres=[]*, *mask=-1*)
The function accepts the list of spheres id's or list of bodies and calculates max and min dimensions, geometrical center.

Parameters

- `idSpheres` (*list*) – list of spheres
- `mask` (*int*) – *Body.mask* for the checked bodies

Returns dictionary with keys `min` (minimal dimension, Vector3), `max` (maximal dimension, Vector3), `minId` (minimal dimension sphere Id, Vector3), `maxId` (maximal dimension sphere Id, Vector3), `center` (central point of bounding box, Vector3), `extends` (sizes of bounding box, Vector3), `volume` (volume of spheres, Real), `mass` (mass of spheres, Real), `number` (number of spheres, int),

9.2 yade.export module

Export (not only) geometry to various formats.

class `yade.export.VTKExporter`

Class for exporting data to VTK Simple Legacy File (for example if, for some reason, you are not able to use VTKRecorder). Export of spheres, facets, interactions and polyhedra is supported.

USAGE: create object vtkExporter = VTKExporter('baseFileName'), add to engines PyRunner with command='vtkExporter.exportSomething(params)' alternatively just use vtkExporter.exportSomething(...) at the end of the script for instance

Example: examples/test/vtk-exporter/vtkExporter.py, examples/test/unv-read/unvReadVTKExport.py.

Parameters

- **`baseName`** (*string*) – name of the exported files. The files would be named baseName-spheres-snapNb.vtk or baseName-facets-snapNb.vtk
- **`startSnap`** (*int*) – the numbering of files will start form startSnap

`exportContactPoints()`

exports constact points and defined properties.

:param [(int,int)] ids: see exportInteractions :param [tuple(2)] what: what to export. parameter is list of couple (name,command). Name is string under which it is save to vtk, command is string to evaluate. Note that the CPs are labeled as i in this function (sccording to their interaction). Scalar, vector and tensor variables are supported. For example, to export stiffness difference from certain value (1e9) (named as dStiff) you should write: ... what=[('dStiff','i.phys.kn-1e9'), ... :param {Interaction:Vector3} useRef: if not specified, current position used. Otherwise use position from dict using interactions as keys. Interactions not in dict are not exported :param string comment: comment to add to vtk file :param int numLabel: number of file (e.g. time step), if unspecified, the last used value + 1 will be used

`exportFacets()`

exports facets (positions) and defined properties. Facets are exported with multiplicated nodes

:param [int]|"all" ids: if "all", then export all facets, otherwise only facets from integer list :param [tuple(2)] what: see exportSpheres :param string comment: comment to add to vtk file :param int numLabel: number of file (e.g. time step), if unspecified, the last used value + 1 will be used

`exportFacetsAsMesh()`

exports facets (positions) and defined properties. Facets are exported as mesh (not with multiplicated nodes). Therefore additional parameters connectivityTable is needed

:param [int]|"all" ids: if "all", then export all facets, otherwise only facets from integer list :param [tuple(2)] what: see exportSpheres :param string comment: comment to add to vtk file :param int numLabel: number of file (e.g. time step), if unspecified, the last used value + 1 will be used :param [(float,float,float)|Vector3] nodes: list of coordinates of nodes :param [(int,int,int)] connectivityTable: list of node ids of individual elements (facets)

`exportInteractions()`

exports interactions and defined properties.

:param [(int,int)]|"all" ids: if "all", then export all interactions, otherwise only interactions from (int,int) list :param [tuple(2)] what: what to export. parameter is list of couple

(name,command). Name is string under which it is save to vtk, command is string to evaluate. Note that the interactions are labeled as i in this function. Scalar, vector and tensor variables are supported. For example, to export stiffness difference from certain value (1e9) (named as dStiff) you should write: ... what=[('dStiff','i.phys.kn-1e9'), ... :param [tuple(2|3)] verticesWhat: what to export on connected bodies. Bodies are labeled as 'b' (or 'b1' and 'b2' if you need treat both bodies differently) :param string comment: comment to add to vtk file :param int numLabel: number of file (e.g. time step), if unspecified, the last used value + 1 will be used

exportPeriodicCell()

exports spheres (positions and radius) and defined properties.

:param string comment: comment to add to vtk file :param int numLabel: number of file (e.g. time step), if unspecified, the last used value + 1 will be used

exportPolyhedra()

Exports polyhedrons and defined properties.

:param ids: if "all", then export all polyhedrons, otherwise only polyhedrons from integer list :type ids: [int] | "all" :param what: what other than then position to export. parameter is list of couple (name,command). Name is string under which it is save to vtk, command is string to evaluate. Note that the bodies are labeled as b in this function. Scalar, vector and tensor variables are supported. For example, to export velocity (with name particleVelocity) and the distance form point (0,0,0) (named as dist) you should write: ... what=[('particleVelocity','b.state.vel'),('dist','b.state.pos.norm()', ... :type what: [tuple(2)] :param string comment: comment to add to vtk file :param int numLabel: number of file (e.g. time step), if unspecified, the last used value + 1 will be used

exportSpheres()

exports spheres (positions and radius) and defined properties.

:param [int]|"all" ids: if "all", then export all spheres, otherwise only spheres from integer list :param [tuple(2)] what: what other than then position and radius export. parameter is list of couple (name,command). Name is string under which it is save to vtk, command is string to evaluate. Note that the bodies are labeled as b in this function. Scalar, vector and tensor variables are supported. For example, to export velocity (with name particleVelocity) and the distance form point (0,0,0) (named as dist) you should write: ... what=[('particleVelocity','b.state.vel'),('dist','b.state.pos.norm()', ... :param string comment: comment to add to vtk file :param int numLabel: number of file (e.g. time step), if unspecified, the last used value + 1 will be used :param bool useRef: if False (default), use current position of the spheres for export, use reference position otherwise

class yade.export.**VTKWriter**

USAGE: create object vtk_writer = VTKWriter('base_file_name'), add to engines PyRunner with command='vtk_writer.snapshot()'

snapshot()

yade.export.**gmshGeo**(*filename*, *comment=''*, *mask=-1*, *accuracy=-1*)

Save spheres in geo-file for the following using in GMSH (http://www.geuz.org/gmsh/doc/texinfo/) program. The spheres can be there meshed.

Parameters

- **filename** (*string*) – the name of the file, where sphere coordinates will be exported.
- **mask** (*int*) – export only spheres with the corresponding mask export only spheres with the corresponding mask
- **accuracy** (*float*) – the accuracy parameter, which will be set for the poinst in geo-file. By default: 1./10. of the minimal sphere diameter.

Returns number of spheres which were exported.

Return type int

yade.export.text(*filename*, *mask=-1*)
Save sphere coordinates into a text file; the format of the line is: x y z r. Non-spherical bodies are silently skipped. Example added to examples/regular-sphere-pack/regular-sphere-pack.py

Parameters

- **filename** (*string*) – the name of the file, where sphere coordinates will be exported.
- **mask** (*int*) – export only spheres with the corresponding mask

Returns number of spheres which were written.

Return type int

yade.export.text2vtk(*inFileName*, *outFileName*)
Converts text file (created by *export.textExt* function) into vtk file. See examples/test/paraview-spheres-solid-section/export_text.py example

Parameters

- **inFileName** (*str*) – name of input text file
- **outFileName** (*str*) – name of output vtk file

yade.export.text2vtkSection(*inFileName*, *outFileName*, *point*, *normal=(1, 0, 0)*)
Converts section through spheres from text file (created by *export.textExt* function) into vtk file. See examples/test/paraview-spheres-solid-section/export_text.py example

Parameters

- **inFileName** (*str*) – name of input text file
- **outFileName** (*str*) – name of output vtk file
- **point** (*Vector3|(float,float,float)*) – coordinates of a point lying on the section plane
- **normal** (*Vector3|(float,float,float)*) – normal vector of the section plane

yade.export.textClumps(*filename*, *format='x_y_z_r_clumpId'*, *comment=''*, *mask=-1*)
Save clumps-members into a text file. Non-clumps members are bodies are silently skipped.

Parameters

- **filename** (*string*) – the name of the file, where sphere coordinates will be exported.
- **comment** (*string*) – the text, which will be added as a comment at the top of file. If you want to create several lines of text, please use '\n#' for next lines.
- **mask** (*int*) – export only spheres with the corresponding mask export only spheres with the corresponding mask

Returns number of clumps, number of spheres which were written.

Return type int

yade.export.textExt(*filename*, *format='x_y_z_r'*, *comment=''*, *mask=-1*, *attrs=[]*)
Save sphere coordinates and other parameters into a text file in specific format. Non-spherical bodies are silently skipped. Users can add here their own specific format, giving meaningful names. The first file row will contain the format name. Be sure to add the same format specification in ymport.textExt.

Parameters

- **filename** (*string*) – the name of the file, where sphere coordinates will be exported.
- **format** (*string*) – the name of output format. Supported 'x_y_z_r'(default), 'x_y_z_r_matId', 'x_y_z_r_attrs' (use proper comment)

- **comment** (*string*) – the text, which will be added as a comment at the top of file. If you want to create several lines of text, please use '\n#' for next lines. With 'x_y_z_r_attrs' format, the last (or only) line should consist of column headers of quantities passed as attrs (1 comment word for scalars, 3 comment words for vectors and 9 comment words for matrices)
- **mask** (*int*) – export only spheres with the corresponding mask export only spheres with the corresponding mask
- **attrs** (*[str]*) – attributes to be exported with 'x_y_z_r_attrs' format. Each str in the list is evaluated for every body exported with body=b (i.e. 'b.state.pos.norm()' would stand for distance of body from coordinate system origin)

Returns number of spheres which were written.

Return type int

yade.export.**textPolyhedra**(*fileName*, *comment=''*, *mask=-1*, *explanationComment=True*, *attrs=[]*)

Save polyhedra into a text file. Non-polyhedra bodies are silently skipped.

Parameters

- **filename** (*string*) – the name of the output file
- **comment** (*string*) – the text, which will be added as a comment at the top of file. If you want to create several lines of text, please use '\n#' for next lines.
- **mask** (*int*) – export only polyhedra with the corresponding mask
- **explanationComment** (*str*) – inclde explanation of format to the beginning of file

Returns number of polyhedra which were written.

Return type int

9.3 yade.geom module

Creates geometry objects from facets.

yade.geom.**facetBox**(*center*, *extents*, *orientation=Quaternion((1, 0, 0), 0)*, *wallMask=63*, ***kw*)

Create arbitrarily-aligned box composed of facets, with given center, extents and orientation. If any of the box dimensions is zero, corresponding facets will not be created. The facets are oriented outwards from the box.

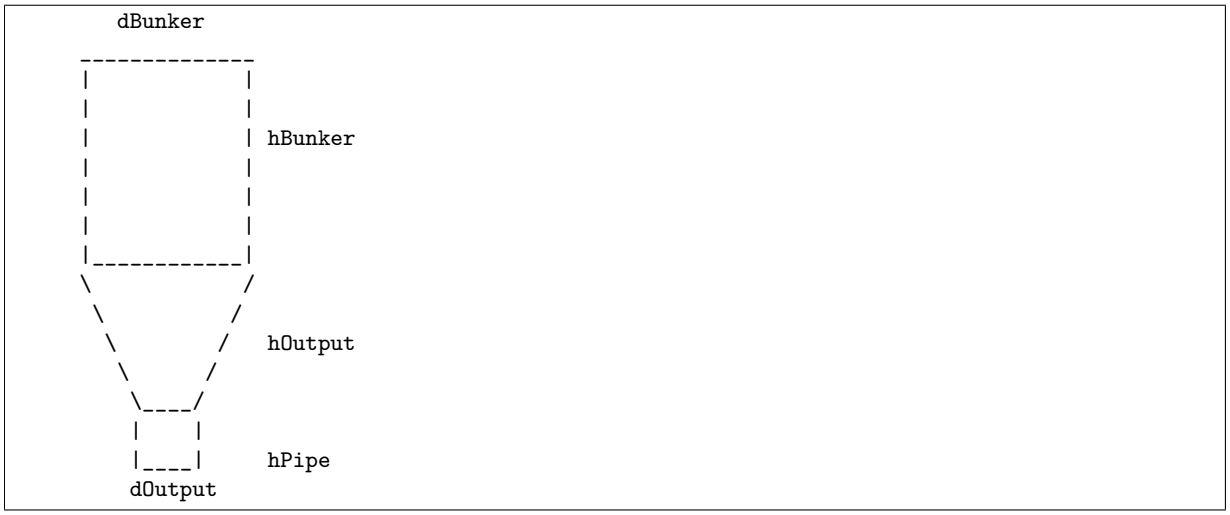
Parameters

- **center** (*Vector3*) – center of the box
- **extents** (*Vector3*) – lengths of the box sides
- **orientation** (*Quaternion*) – orientation of the box
- **wallMask** (*bitmask*) – determines which walls will be created, in the order -x (1), +x (2), -y (4), +y (8), -z (16), +z (32). The numbers are ANDed; the default 63 means to create all walls
- ****kw** – (unused keyword arguments) passed to *utils.facet*

Returns list of facets forming the box

yade.geom.**facetBunker**(*center*, *dBunker*, *dOutput*, *hBunker*, *hOutput*, *hPipe=0.0*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=4*, *angleRange=None*, *closeGap=False*, ***kw*)

Create arbitrarily-aligned bunker, composed of facets, with given center, radii, heights and orientation. Return List of facets forming the bunker;



Parameters

- **center** (*Vector3*) – center of the created bunker
- **dBunker** (*float*) – bunker diameter, top
- **dOutput** (*float*) – bunker output diameter
- **hBunker** (*float*) – bunker height
- **hOutput** (*float*) – bunker output height
- **hPipe** (*float*) – bunker pipe height
- **orientation** (*Quaternion*) – orientation of the bunker; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the bunker surface (>=5)
- **wallMask** (*bitmask*) – determines which walls will be created, in the order up (1), down (2), side (4). The numbers are ANDed; the default 7 means to create all walls
- **angleRange** (*(ϑmin,Θmax)*) – allows one to create only part of bunker by specifying range of angles; if `None`, (0,2*pi) is assumed.
- **closeGap** (*bool*) – close range skipped in angleRange with triangular facets at cylinder bases.
- ****kw** – (unused keyword arguments) passed to utils.facet;

yade.geom.**facetCone**(*center*, *radiusTop*, *radiusBottom*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=7*, *angleRange=None*, *closeGap=False*, *radiusTopInner=-1*, *radiusBottomInner=-1*, ***kw*)

Create arbitrarily-aligned cone composed of facets, with given center, radius, height and orientation. Return List of facets forming the cone;

Parameters

- **center** (*Vector3*) – center of the created cylinder
- **radiusTop** (*float*) – cone top radius
- **radiusBottom** (*float*) – cone bottom radius
- **radiusTopInner** (*float*) – inner radius of cones top, -1 by default
- **radiusBottomInner** (*float*) – inner radius of cones bottom, -1 by default
- **height** (*float*) – cone height
- **orientation** (*Quaternion*) – orientation of the cone; the reference orientation has axis along the +x axis.

- **segmentsNumber** (*int*) – number of edges on the cone surface (>=5)
- **wallMask** (*bitmask*) – determines which walls will be created, in the order up (1), down (2), side (4). The numbers are ANDed; the default 7 means to create all walls
- **angleRange** (*(ϑmin,Θmax)*) – allows one to create only part of cone by specifying range of angles; if None, (0,2*pi) is assumed.
- **closeGap** (*bool*) – close range skipped in angleRange with triangular facets at cylinder bases.
- ****kw** – (unused keyword arguments) passed to utils.facet;

yade.geom.**facetCylinder**(*center*, *radius*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=7*, *angleRange=None*, *closeGap=False*, *radiusTopInner=-1*, *radiusBottomInner=-1*, ***kw*)

Create arbitrarily-aligned cylinder composed of facets, with given center, radius, height and orientation. Return List of facets forming the cylinder;

Parameters

- **center** (*Vector3*) – center of the created cylinder
- **radius** (*float*) – cylinder radius
- **height** (*float*) – cylinder height
- **radiusTopInner** (*float*) – inner radius of cylinders top, -1 by default
- **radiusBottomInner** (*float*) – inner radius of cylinders bottom, -1 by default
- **orientation** (*Quaternion*) – orientation of the cylinder; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the cylinder surface (>=5)
- **wallMask** (*bitmask*) – determines which walls will be created, in the order up (1), down (2), side (4). The numbers are ANDed; the default 7 means to create all walls
- **angleRange** (*(ϑmin,Θmax)*) – allows one to create only part of bunker by specifying range of angles; if None, (0,2*pi) is assumed.
- **closeGap** (*bool*) – close range skipped in angleRange with triangular facets at cylinder bases.
- ****kw** – (unused keyword arguments) passed to utils.facet;

yade.geom.**facetCylinderConeGenerator**(*center*, *radiusTop*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=7*, *angleRange=None*, *closeGap=False*, *radiusBottom=-1*, *radiusTopInner=-1*, *radiusBottomInner=-1*, ***kw*)

Please, do not use this function directly! Use geom.facetCylinder and geom.facetCone instead. This is the base function for generating cylinders and cones from facets. :param float radiusTop: top radius :param float radiusBottom: bottom radius :param **kw: (unused keyword arguments) passed to utils.facet;

yade.geom.**facetHelix**(*center*, *radiusOuter*, *pitch*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *angleRange=None*, *radiusInner=0*, ***kw*)

Create arbitrarily-aligned helix composed of facets, with given center, radius (outer and inner), pitch and orientation. Return List of facets forming the helix;

Parameters

- **center** (*Vector3*) – center of the created cylinder
- **radiusOuter** (*float*) – outer radius
- **radiusInner** (*float*) – inner height (can be 0)

- **orientation** (*Quaternion*) – orientation of the helix; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the helix surface (>=3)
- **angleRange** (*(ϑmin,Θmax)*) – range of angles; if None, (0,2*pi) is assumed.
- ****kw** – (unused keyword arguments) passed to utils.facet;

yade.geom.**facetParallelepiped**(*center*, *extents*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *wallMask=63*, ***kw*)

Create arbitrarily-aligned Parallelepiped composed of facets, with given center, extents, height and orientation. If any of the parallelepiped dimensions is zero, corresponding facets will not be created. The facets are oriented outwards from the parallelepiped.

Parameters

- **center** (*Vector3*) – center of the parallelepiped
- **extents** (*Vector3*) – lengths of the parallelepiped sides
- **height** (*Real*) – height of the parallelepiped (along axis z)
- **orientation** (*Quaternion*) – orientation of the parallelepiped
- **wallMask** (*bitmask*) – determines which walls will be created, in the order -x (1), +x (2), -y (4), +y (8), -z (16), +z (32). The numbers are ANDed; the default 63 means to create all walls
- ****kw** – (unused keyword arguments) passed to *utils.facet*

Returns list of facets forming the parallelepiped

yade.geom.**facetPolygon**(*center*, *radiusOuter*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *angleRange=None*, *radiusInner=0*, ***kw*)

Create arbitrarily-aligned polygon composed of facets, with given center, radius (outer and inner) and orientation. Return List of facets forming the polygon;

Parameters

- **center** (*Vector3*) – center of the created cylinder
- **radiusOuter** (*float*) – outer radius
- **radiusInner** (*float*) – inner height (can be 0)
- **orientation** (*Quaternion*) – orientation of the polygon; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the polygon surface (>=3)
- **angleRange** (*(ϑmin,Θmax)*) – allows one to create only part of polygon by specifying range of angles; if None, (0,2*pi) is assumed.
- ****kw** – (unused keyword arguments) passed to utils.facet;

yade.geom.**facetPolygonHelixGenerator**(*center*, *radiusOuter*, *pitch=0*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *angleRange=None*, *radiusInner=0*, ***kw*)

Please, do not use this function directly! Use geom.facetPloygon and geom.facetHelix instead. This is the base function for generating polygons and helixes from facets.

yade.geom.**facetSphere**(*center*, *radius*, *thetaResolution=8*, *phiResolution=8*, *returnElementMap=False*, ***kw*)

Create arbitrarily-aligned sphere composed of facets, with given center, radius and orientation. Return List of facets forming the sphere. Parameters inspired by ParaView sphere glyph

Parameters

- **center** (*Vector3*) – center of the created sphere
- **radius** (*float*) – sphere radius
- **thetaResolution** (*int*) – number of facets around “equator”

- **phiResolution** (*int*) – number of facets between "poles" + 1
- **returnElementMap** (*bool*) – returns also tuple of nodes ((x1,y1,z1),(x2,y2,z2),...) and elements ((id01,id02,id03),(id11,id12,id13),...) if true, only facets otherwise
- ****kw** – (unused keyword arguments) passed to utils.facet;

9.4 yade.linterpolation module

Module for rudimentary support of manipulation with piecewise-linear functions (which are usually interpolations of higher-order functions, whence the module name). Interpolation is always given as two lists of the same length, where the x-list must be increasing.

Periodicity is supported by supposing that the interpolation can wrap from the last x-value to the first x-value (which should be 0 for meaningful results).

Non-periodic interpolation can be converted to periodic one by padding the interpolation with constant head and tail using the sanitizeInterpolation function.

There is a c++ template function for interpolating on such sequences in pkg/common/Engine/PartialEngine/LinearInterpolate.hpp (stateful, therefore fast for sequential reads).

TODO: Interpolating from within python is not (yet) supported.

yade.linterpolation.integral(*x*, *y*)
Return integral of piecewise-linear function given by points x0,x1,… and y0,y1,…

yade.linterpolation.revIntegrateLinear(*I*, *x0*, *y0*, *x1*, *y1*)
Helper function, returns value of integral variable x for linear function f passing through (x0,y0),(x1,y1) such that 1. x [x0,x1] 2. _x0^x f dx=I and raise exception if such number doesn't exist or the solution is not unique (possible?)

yade.linterpolation.sanitizeInterpolation(*x*, *y*, *x0*, *x1*)
Extends piecewise-linear function in such way that it spans at least the x0…x1 interval, by adding constant padding at the beginning (using y0) and/or at the end (using y1) or not at all.

yade.linterpolation.xFractionalFromIntegral(*integral*, *x*, *y*)
Return x within range x0…xn such that _x0^x f dx==integral. Raises error if the integral value is not reached within the x-range.

yade.linterpolation.xFromIntegral(*integralValue*, *x*, *y*)
Return x such that _x0^x f dx==integral. x wraps around at xn. For meaningful results, therefore, x0 should == 0

9.5 yade.pack module

Creating packings and filling volumes defined by boundary representation or constructive solid geometry.

For examples, see

- scripts/test/gts-operators.py
- scripts/test/gts-random-pack-obb.py
- scripts/test/gts-random-pack.py
- scripts/test/pack-cloud.py
- scripts/test/pack-predicates.py
- examples/packs/packs.py
- examples/gts-horse/gts-horse.py
- examples/WireMatPM/wirepackings.py

yade.pack.**SpherePack_toSimulation**(*self*, *rot=Matrix3(1, 0, 0, 0, 1, 0, 0, 0, 1)*, ***kw*)
Append spheres directly to the simulation. In addition calling *O.bodies.append*, this method also appropriately sets periodic cell information of the simulation.

```
>>> from yade import pack; from math import *
>>> sp=pack.SpherePack()
```

Create random periodic packing with 20 spheres:

```
>>> sp.makeCloud((0,0,0),(5,5,5),rMean=.5,rRelFuzz=.5,periodic=True,num=20)
20
```

Virgin simulation is aperiodic:

```
>>> O.reset()
>>> O.periodic
False
```

Add generated packing to the simulation, rotated by 45° along +z

```
>>> sp.toSimulation(rot=Quaternion((0,0,1),pi/4),color=(0,0,1))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Periodic properties are transferred to the simulation correctly, including rotation (this could be avoided by explicitly passing "hSize=O.cell.hSize" as an argument):

```
>>> O.periodic
True
>>> O.cell.refSize
Vector3(5,5,5)
```

yade.pack.**filterSpherePack**(*predicate*, *spherePack*, *returnSpherePack=None*, ***kw*)
Using given SpherePack instance, return spheres that satisfy predicate. It returns either a *pack.SpherePack* (if returnSpherePack) or a list. The packing will be recentered to match the predicate and warning is given if the predicate is larger than the packing.

yade.pack.**gtsSurface2Facets**(*surf*, ***kw*)
Construct facets from given GTS surface. **kw is passed to utils.facet.

yade.pack.**gtsSurfaceBestFitOBB**(*surf*)
Return (Vector3 center, Vector3 halfSize, Quaternion orientation) describing best-fit oriented bounding box (OBB) for the given surface. See cloudBestFitOBB for details.

yade.pack.**hexaNet**(*radius*, *cornerCoord=[0, 0, 0]*, *xLength=1.0*, *yLength=0.5*, *mos=0.08*, *a=0.04*, *b=0.04*, *startAtCorner=True*, *isSymmetric=False*, ***kw*)
Definition of the particles for a hexagonal wire net in the x-y-plane for the WireMatPM.

Parameters

- **radius** – radius of the particle
- **cornerCoord** – coordinates of the lower left corner of the net
- **xLenght** – net length in x-direction
- **yLenght** – net length in y-direction
- **mos** – mesh opening size (horizontal distance between the double twists)
- **a** – length of double-twist
- **b** – height of single wire section
- **startAtCorner** – if true the generation starts with a double-twist at the lower left corner
- **isSymmetric** – defines if the net is symmetric with respect to the y-axis

Returns set of spheres which defines the net (net) and exact dimensions of the net (lx,ly).

note:: This packing works for the WireMatPM only. The particles at the corner are always generated first. For examples on how to use this packing see examples/WireMatPM. In order to create the proper interactions for the net the interaction radius has to be adapted in the simulation.

class yade.pack.**inGtsSurface_py**(*inherits Predicate*)

This class was re-implemented in c++, but should stay here to serve as reference for implementing Predicates in pure python code. C++ allows us to play dirty tricks in GTS which are not accessible through pygts itself; the performance penalty of pygts comes from fact that if constructs and destructs bb tree for the surface at every invocation of gts.Point().is_inside(). That is cached in the c++ code, provided that the surface is not manipulated with during lifetime of the object (user's responsibility).

—

Predicate for GTS surfaces. Constructed using an already existing surfaces, which must be closed.

import gts surf=gts.read(open('horse.gts')) inGtsSurface(surf)

Note: Padding is optionally supported by testing 6 points along the axes in the pad distance. This must be enabled in the ctor by saying doSlowPad=True. If it is not enabled and pad is not zero, warning is issued.

---

aabb()

center() → Vector3

dim() → Vector3

class yade.pack.**inSpace**(*inherits Predicate*)

Predicate returning True for any points, with infinite bounding box.

aabb()

center()

dim()

yade.pack.**randomDensePack**(*predicate*, *radius*, *material=-1*, *dim=None*, *cropLayers=0*, *rRelFuzz=0.0*, *spheresInCell=0*, *memoizeDb=None*, *useOBB=False*, *memoDbg=False*, *color=None*, *returnSpherePack=None*)

Generator of random dense packing with given geometry properties, using TriaxialTest (aperiodic) or PeriIsoCompressor (periodic). The periodicity depens on whether the spheresInCell parameter is given.

O.switchScene() magic is used to have clean simulation for TriaxialTest without deleting the original simulation. This function therefore should never run in parallel with some code accessing your simulation.

Parameters

- **predicate** – solid-defining predicate for which we generate packing
- **spheresInCell** – if given, the packing will be periodic, with given number of spheres in the periodic cell.
- **radius** – mean radius of spheres
- **rRelFuzz** – relative fuzz of the radius – e.g. radius=10, rRelFuzz=.2, then spheres will have radii 10 ± (10*.2)), with an uniform distribution. 0 by default, meaning all spheres will have exactly the same radius.
- **cropLayers** – (aperiodic only) how many layers of spheres will be added to the computed dimension of the box so that there no (or not so much, at least) boundary effects at the boundaries of the predicate.
- **dim** – dimension of the packing, to override dimensions of the predicate (if it is infinite, for instance)
- **memoizeDb** – name of sqlite database (existent or nonexistent) to find an already generated packing or to store the packing that will be generated, if not found (the

technique of caching results of expensive computations is known as memoization). Fuzzy matching is used to select suitable candidate – packing will be scaled, rRelFuzz and dimensions compared. Packing that are too small are dictarded. From the remaining candidate, the one with the least number spheres will be loaded and returned.

- **useOBB** – effective only if a inGtsSurface predicate is given. If true (not default), oriented bounding box will be computed first; it can reduce substantially number of spheres for the triaxial compression (like 10× depending on how much asymmetric the body is), see examples/gts-horse/gts-random-pack-obb.py
- **memoDbg** – show packings that are considered and reasons why they are rejected/accepted
- **returnSpherePack** – see the corresponding argument in *pack.filterSpherePack*

Returns SpherePack object with spheres, filtered by the predicate.

yade.pack.**randomPeriPack**(*radius*, *initSize*, *rRelFuzz=0.0*, *memoizeDb=None*, *noPrint=False*)
Generate periodic dense packing.

A cell of initSize is stuffed with as many spheres as possible, then we run periodic compression with PeriIsoCompressor, just like with randomDensePack.

Parameters

- **radius** – mean sphere radius
- **rRelFuzz** – relative fuzz of sphere radius (equal distribution); see the same param for randomDensePack.
- **initSize** – initial size of the periodic cell.

Returns SpherePack object, which also contains periodicity information.

yade.pack.**regularHexa**(*predicate*, *radius*, *gap*, ***kw*)
Return set of spheres in regular hexagonal grid, clipped inside solid given by predicate. Created spheres will have given radius and will be separated by gap space.

yade.pack.**regularOrtho**(*predicate*, *radius*, *gap*, ***kw*)
Return set of spheres in regular orthogonal grid, clipped inside solid given by predicate. Created spheres will have given radius and will be separated by gap space.

yade.pack.**revolutionSurfaceMeridians**(*sects*, *angles*, *origin=Vector3(0, 0, 0)*, *orientation=Quaternion((1, 0, 0), 0)*)
Revolution surface given sequences of 2d points and sequence of corresponding angles, returning sequences of 3d points representing meridian sections of the revolution surface. The 2d sections are turned around z-axis, but they can be transformed using the origin and orientation arguments to give arbitrary orientation.

yade.pack.**sweptPolylines2gtsSurface**(*pts*, *threshold=0*, *capStart=False*, *capEnd=False*)
Create swept suface (as GTS triangulation) given same-length sequences of points (as 3-tuples).

If threshold is given (>0), then

- degenerate faces (with edges shorter than threshold) will not be created
- gts.Surface().cleanup(threshold) will be called before returning, which merges vertices mutually closer than threshold. In case your pts are closed (last point concident with the first one) this will the surface strip of triangles. If you additionally have capStart==True and capEnd==True, the surface will be closed.

Note: capStart and capEnd make the most naive polygon triangulation (diagonals) and will perhaps fail for non-convex sections.

Warning: the algorithm connects points sequentially; if two polylines are mutually rotated or have inverse sense, the algorithm will not detect it and connect them regardless in their given order.

Creation, manipulation, IO for generic sphere packings.

class yade._packSpheres.**SpherePack**

Set of spheres represented as centers and radii. This class is returned by *pack.randomDensePack*, *pack.randomPeriPack* and others. The object supports iteration over spheres, as in

```
>>> sp=SpherePack()
>>> for center,radius in sp: print center,radius
```

```
>>> for sphere in sp: print sphere[0],sphere[1]   ## same, but without unpacking the tuple automatically
```

```
>>> for i in range(0,len(sp)): print sp[i][0], sp[i][1]   ## same, but accessing spheres by index
```

Special constructors

Construct from list of `[(c1,r1),(c2,r2),…]`. To convert two same-length lists of `centers` and `radii`, construct with `zip(centers,radii)`.

`__init__`([*(list)list*]) → None

Empty constructor, optionally taking list [((cx,cy,cz),r), …] for initial data.

`aabb`() → tuple

Get axis-aligned bounding box coordinates, as 2 3-tuples.

`add`(*(Vector3)arg2*, *(float)arg3*) → None

Add single sphere to packing, given center as 3-tuple and radius

`appliedPsdScaling`

A factor between 0 and 1, uniformly applied on all sizes of of the PSD.

`cellFill`(*(Vector3)arg2*) → None

Repeat the packing (if periodic) so that the results has dim() >= given size. The packing retains periodicity, but changes cellSize. Raises exception for non-periodic packing.

`cellRepeat`(*(Vector3i)arg2*) → None

Repeat the packing given number of times in each dimension. Periodicity is retained, cellSize changes. Raises exception for non-periodic packing.

`cellSize`

Size of periodic cell; is Vector3(0,0,0) if not periodic. (Change this property only if you know what you're doing).

`center`() → Vector3

Return coordinates of the bounding box center.

`dim`() → Vector3

Return dimensions of the packing in terms of aabb(), as a 3-tuple.

`fromList`(*(list)arg2*) → None

Make packing from given list, same format as for constructor. Discards current data.

fromList((SpherePack)arg1, (object)centers, (object)radii) → None : Make packing from given list, same format as for constructor. Discards current data.

`fromSimulation`() → None

Make packing corresponding to the current simulation. Discards current data.

`getClumps`() → tuple

Return lists of sphere ids sorted by clumps they belong to. The return value is (standalones,[clump1,clump2,…]), where each item is list of id's of spheres.

`hasClumps`() → bool

Whether this object contains clumps.

`isPeriodic`

was the packing generated in periodic boundaries?

`load`(*(str)fileName*) → None
Load packing from external text file (current data will be discarded).

`makeCloud`([*(Vector3)minCorner=Vector3(0, 0, 0)*[, *(Vector3)maxCorner=Vector3(0, 0, 0)*[, *(float)rMean=-1*[, *(float)rRelFuzz=0*[, *(int)num=-1*[, *(bool)periodic=False*[, *(float)porosity=0.65*[, *(object)psdSizes=[]*[, *(object)psdCumm=[]*[, *(bool)distributeMass=False*[, *(int)seed=0*[, *(Matrix3)hSize=Matrix3(0, 0, 0, 0, 0, 0, 0, 0, 0)*]]]]]]]]]]]]) → int
Create random loose packing enclosed in a parallelepiped (also works in 2D if minCorner[k]=maxCorner[k] for one coordinate). Sphere radius distribution can be specified using one of the following ways:

1. *rMean*, *rRelFuzz* and *num* gives uniform radius distribution in $rMean \times (1 \pm rRelFuzz)$. Less than *num* spheres can be generated if it is too high.
2. *rRelFuzz*, *num* and (optional) *porosity*, which estimates mean radius so that *porosity* is attained at the end. *rMean* must be less than 0 (default). *porosity* is only an initial guess for the generation algorithm, which will retry with higher porosity until the prescibed *num* is obtained.
3. *psdSizes* and *psdCumm*, two arrays specifying points of the particle size distribution function. As many spheres as possible are generated.
4. *psdSizes*, *psdCumm*, *num*, and (optional) *porosity*, like above but if *num* is not obtained, *psdSizes* will be scaled down uniformly, until *num* is obtained (see *appliedPsdScaling*).

By default (with `distributeMass==False`), the distribution is applied to particle radii. The usual sense of "particle size distribution" is the distribution of *mass fraction* (rather than particle count); this can be achieved with `distributeMass=True`.

If *num* is defined, then sizes generation is deterministic, giving the best fit of target distribution. It enables spheres placement in descending size order, thus giving lower porosity than the random generation.

Parameters

- **`minCorner`** (*Vector3*) – lower corner of an axis-aligned box
- **`maxCorner`** (*Vector3*) – upper corner of an axis-aligned box
- **`hSize`** (*Matrix3*) – base vectors of a generalized box (arbitrary parallelepiped, typically *Cell::hSize*), superseeds minCorner and maxCorner if defined. For periodic boundaries only.
- **`rMean`** (*float*) – mean radius or spheres
- **`rRelFuzz`** (*float*) – dispersion of radius relative to rMean
- **`num`** (*int*) – number of spheres to be generated. If negavite (default), generate as many as possible with stochastic sizes, ending after a fixed number of tries to place the sphere in space, else generate exactly *num* spheres with deterministic size distribution.
- **`periodic`** (*bool*) – whether the packing to be generated should be periodic
- **`porosity`** (*float*) – initial guess for the iterative generation procedure (if *num*>1). The algorithm will be retrying until the number of generated spheres is *num*. The first iteration tries with the provided porosity, but next iterations increase it if necessary (hence an initialy high porosity can speed-up the algorithm). If *psdSizes* is not defined, *rRelFuzz* (z) and *num* (N) are used so that the porosity given (ρ) is approximately achieved at the end of generation, $r_m = \sqrt[3]{\frac{V(1-\rho)}{\frac{4}{3}\pi(1+z^2)N}}$. The default is ρ=0.5. The optimal value depends on *rRelFuzz* or *psdSizes*.
- **`psdSizes`** – sieve sizes (particle diameters) when particle size distribution (PSD) is specified

- **psdCumm** – cummulative fractions of particle sizes given by *psdSizes*; must be the same length as *psdSizes* and should be non-decreasing
- **distributeMass** (*bool*) – if `True`, given distribution will be used to distribute sphere's mass rather than radius of them.
- **seed** – number used to initialize the random number generator.

Returns number of created spheres, which can be lower than *num* depending on the method used.

makeClumpCloud(*(Vector3)minCorner*, *(Vector3)maxCorner*, *(object)clumps*[, *(bool)periodic=False*[, *(int)num=-1*[, *(int)seed=0*]]]) → int

Create random loose packing of clumps within box given by *minCorner* and *maxCorner*. Clumps are selected with equal probability. At most *num* clumps will be positioned if *num* is positive; otherwise, as many clumps as possible will be put in space, until maximum number of attempts to place a new clump randomly is attained. :param seed: number used to initialize the random number generator.

particleSD(*(Vector3)minCorner*, *(Vector3)maxCorner*, *(float)rMean*, *(bool)periodic=False*, *(str)name*, *(int)numSph*[, *(object)radii=[]*[, *(object)passing=[]*[, *(bool)passingIsNotPercentageButCount=False*[, *(int)seed=0*]]]]) → int

Not working. Use makeCloud instead.

particleSD2(*(object)radii*, *(object)passing*, *(int)numSph*[, *(bool)periodic=False*[, *(float)cloudPorosity=0.8*[, *(int)seed=0*]]]) → int

Not working. Use makeCloud instead.

particleSD_2d(*(Vector2)minCorner*, *(Vector2)maxCorner*, *(float)rMean*, *(bool)periodic=False*, *(str)name*, *(int)numSph*[, *(object)radii=[]*[, *(object)passing=[]*[, *(bool)passingIsNotPercentageButCount=False*[, *(int)seed=0*]]]]) → int

Not working. Use makeCloud instead.

psd([*(int)bins=50*[, *(bool)mass=True*]]) → tuple

Return particle size distribution of the packing. :param int bins: number of bins between minimum and maximum diameter :param mass: Compute relative mass rather than relative particle count for each bin. Corresponds to *distributeMass parameter for makeCloud*. :returns: tuple of `(cumm,edges)`, where `cumm` are cummulative fractions for respective diameters and `edges` are those diameter values. Dimension of both arrays is equal to `bins+1`.

relDensity() → float

Relative packing density, measured as sum of spheres' volumes / aabb volume. (Sphere overlaps are ignored.)

rotate(*(Vector3)axis*, *(float)angle*) → None

Rotate all spheres around packing center (in terms of aabb()), given axis and angle of the rotation.

save(*(str)fileName*) → None

Save packing to external text file (will be overwritten).

scale(*(float)arg2*) → None

Scale the packing around its center (in terms of aabb()) by given factor (may be negative).

toList() → list

Return packing data as python list.

toSimulation()

Append spheres directly to the simulation. In addition calling *O.bodies.append*, this method also appropriately sets periodic cell information of the simulation.

```
>>> from yade import pack; from math import *
>>> sp=pack.SpherePack()
```

Create random periodic packing with 20 spheres:

```
>>> sp.makeCloud((0,0,0),(5,5,5),rMean=.5,rRelFuzz=.5,periodic=True,num=20)
20
```

Virgin simulation is aperiodic:

```
>>> O.reset()
>>> O.periodic
False
```

Add generated packing to the simulation, rotated by 45° along +z

```
>>> sp.toSimulation(rot=Quaternion((0,0,1),pi/4),color=(0,0,1))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Periodic properties are transferred to the simulation correctly, including rotation (this could be avoided by explicitly passing "hSize=O.cell.hSize" as an argument):

```
>>> O.periodic
True
>>> O.cell.refSize
Vector3(5,5,5)
```

translate(*(Vector3)arg2*) → None
Translate all spheres by given vector.

class yade._packSpheres.**SpherePackIterator**

__init__(*(SpherePackIterator)arg2*) → None

next() → tuple

Spatial predicates for volumes (defined analytically or by triangulation).

class yade._packPredicates.**Predicate**

aabb() → tuple
aabb((Predicate)arg1) → None

center() → Vector3

dim() → Vector3

class yade._packPredicates.**PredicateBoolean**(*inherits Predicate*)
Boolean operation on 2 predicates (abstract class)

A

B

__init__()
Raises an exception This class cannot be instantiated from Python

aabb() → tuple
aabb((Predicate)arg1) → None

center() → Vector3

dim() → Vector3

class yade._packPredicates.**PredicateDifference**(*inherits PredicateBoolean → Predicate*)
Difference (conjunction with negative predicate) of 2 predicates. A point has to be inside the first and outside the second predicate. Can be constructed using the `-` operator on predicates: `pred1 - pred2`.

A

`B`

`__init__`(*(object)arg2*, *(object)arg3*) → None

`aabb`() → tuple
aabb((Predicate)arg1) → None

`center`() → Vector3

`dim`() → Vector3

class `yade._packPredicates.PredicateIntersection`(*inherits PredicateBoolean → Predicate*)
Intersection (conjunction) of 2 predicates. A point has to be inside both predicates. Can be constructed using the `&` operator on predicates: `pred1 & pred2`.

`A`

`B`

`__init__`(*(object)arg2*, *(object)arg3*) → None

`aabb`() → tuple
aabb((Predicate)arg1) → None

`center`() → Vector3

`dim`() → Vector3

class `yade._packPredicates.PredicateSymmetricDifference`(*inherits PredicateBoolean → Predicate*)
SymmetricDifference (exclusive disjunction) of 2 predicates. A point has to be in exactly one predicate of the two. Can be constructed using the `^` operator on predicates: `pred1 ^ pred2`.

`A`

`B`

`__init__`(*(object)arg2*, *(object)arg3*) → None

`aabb`() → tuple
aabb((Predicate)arg1) → None

`center`() → Vector3

`dim`() → Vector3

class `yade._packPredicates.PredicateUnion`(*inherits PredicateBoolean → Predicate*)
Union (non-exclusive disjunction) of 2 predicates. A point has to be inside any of the two predicates to be inside. Can be constructed using the `|` operator on predicates: `pred1 | pred2`.

`A`

`B`

`__init__`(*(object)arg2*, *(object)arg3*) → None

`aabb`() → tuple
aabb((Predicate)arg1) → None

`center`() → Vector3

`dim`() → Vector3

class `yade._packPredicates.inAlignedBox`(*inherits Predicate*)
Axis-aligned box predicate

`__init__`(*(Vector3)minAABB*, *(Vector3)maxAABB*) → None
Ctor taking minumum and maximum points of the box (as 3-tuples).

`aabb`() → tuple
aabb((Predicate)arg1) → None

`center`() → Vector3

`dim`() → Vector3

class yade._packPredicates.**inCylinder**(*inherits Predicate*)
Cylinder predicate

__init__(*(Vector3)centerBottom*, *(Vector3)centerTop*, *(float)radius*) → None
Ctor taking centers of the lateral walls (as 3-tuples) and radius.

aabb() → tuple
aabb((Predicate)arg1) → None

center() → Vector3

dim() → Vector3

class yade._packPredicates.**inEllipsoid**(*inherits Predicate*)
Ellipsoid predicate

__init__(*(Vector3)centerPoint*, *(Vector3)abc*) → None
Ctor taking center of the ellipsoid (3-tuple) and its 3 radii (3-tuple).

aabb() → tuple
aabb((Predicate)arg1) → None

center() → Vector3

dim() → Vector3

class yade._packPredicates.**inGtsSurface**(*inherits Predicate*)
GTS surface predicate

__init__(*(object)surface*[, *(bool)noPad*]) → None
Ctor taking a gts.Surface() instance, which must not be modified during instance lifetime. The optional noPad can disable padding (if set to True), which speeds up calls several times. Note: padding checks inclusion of 6 points along +- cardinal directions in the pad distance from given point, which is not exact.

aabb() → tuple
aabb((Predicate)arg1) → None

center() → Vector3

dim() → Vector3

surf
The associated gts.Surface object.

class yade._packPredicates.**inHyperboloid**(*inherits Predicate*)
Hyperboloid predicate

__init__(*(Vector3)centerBottom*, *(Vector3)centerTop*, *(float)radius*, *(float)skirt*) → None
Ctor taking centers of the lateral walls (as 3-tuples), radius at bases and skirt (middle radius).

aabb() → tuple
aabb((Predicate)arg1) → None

center() → Vector3

dim() → Vector3

class yade._packPredicates.**inParallelepiped**(*inherits Predicate*)
Parallelepiped predicate

__init__(*(Vector3)o*, *(Vector3)a*, *(Vector3)b*, *(Vector3)c*) → None
Ctor taking four points: `o` (for origin) and then `a`, `b`, `c` which define endpoints of 3 respective edges from `o`.

aabb() → tuple
aabb((Predicate)arg1) → None

center() → Vector3

dim() → Vector3

class yade._packPredicates.**inSphere**(*inherits Predicate*)
Sphere predicate.

__init__(*(Vector3)center*, *(float)radius*) → None
Ctor taking center (as a 3-tuple) and radius

aabb() → tuple
aabb((Predicate)arg1) → None

center() → Vector3

dim() → Vector3

class yade._packPredicates.**notInNotch**(*inherits Predicate*)
Outside of infinite, rectangle-shaped notch predicate

__init__(*(Vector3)centerPoint*, *(Vector3)edge*, *(Vector3)normal*, *(float)aperture*) → None
Ctor taking point in the symmetry plane, vector pointing along the edge, plane normal and aperture size. The side inside the notch is edge×normal. Normal is made perpendicular to the edge. All vectors are normalized at construction time.

aabb() → tuple
aabb((Predicate)arg1) → None

center() → Vector3

dim() → Vector3

Computation of oriented bounding box for cloud of points.

yade._packObb.**cloudBestFitOBB**(*(tuple)arg1*) → tuple
Return (Vector3 center, Vector3 halfSize, Quaternion orientation) of best-fit oriented bounding-box for given tuple of points (uses brute-force velome minimization, do not use for very large clouds).

9.6 yade.plot module

Module containing utility functions for plotting inside yade. See examples/simple-scene/simple-scene-plot.py or examples/concrete/uniax.py for example of usage.

yade.plot.**data** = **{'force': [nan, nan, 1000.0, nan, nan, 1000.0], 'sigma': [12, nan, nan, 12, nan, nan], 'eps**
Global dictionary containing all data values, common for all plots, in the form {'name':[value,...],...}. Data should be added using plot.addData function. All [value,...] columns have the same length, they are padded with NaN if unspecified.

yade.plot.**plots** = **{'i': ('t',), 'i ': ('z1', 'v1')}**
dictionary x-name -> (yspec,...), where yspec is either y-name or (y-name,'line-specification'). If `(yspec,...)` is `None`, then the plot has meaning of image, which will be taken from respective field of *plot.imgData*.

yade.plot.**labels** = **{}**
Dictionary converting names in data to human-readable names (TeX names, for instance); if a variable is not specified, it is left untranslated.

yade.plot.**live** = **False**
Enable/disable live plot updating. Disabled by default for now, since it has a few rough edges.

yade.plot.**liveInterval** = **1**
Interval for the live plot updating, in seconds.

yade.plot.**autozoom** = **True**
Enable/disable automatic plot rezooming after data update.

yade.plot.**plot**(*noShow=False*, *subPlots=True*)
Do the actual plot, which is either shown on screen (and nothing is returned: if *noShow* is `False` - note that your yade compilation should present qt4 feature so that figures can be displayed) or, if *noShow* is `True`, returned as matplotlib's Figure object or list of them.

You can use

```
>>> from yade import plot
>>> plot.resetData()
>>> plot.plots={'foo':('bar',)}
>>> plot.plot(noShow=True).savefig('someFile.pdf')
>>> import os
>>> os.path.exists('someFile.pdf')
True
>>> os.remove('someFile.pdf')
```

to save the figure to file automatically.

Note: For backwards compatibility reasons, *noShow* option will return list of figures for multiple figures but a single figure (rather than list with 1 element) if there is only 1 figure.

---

yade.plot.**reset**()
Reset all plot-related variables (data, plots, labels)

yade.plot.**resetData**()
Reset all plot data; keep plots and labels intact.

yade.plot.**splitData**()
Make all plots discontinuous at this point (adds nan's to all data fields)

yade.plot.**reverseData**()
Reverse yade.plot.data order.

Useful for tension-compression test, where the initial (zero) state is loaded and, to make data continuous, last part must *end* in the zero state.

yade.plot.**addData**(**d_in*, ***kw*)
Add data from arguments name1=value1,name2=value2 to yade.plot.data. (the old {'name1':value1,'name2':value2} is deprecated, but still supported)

New data will be padded with nan's, unspecified data will be nan (nan's don't appear in graphs). This way, equal length of all data is assured so that they can be plotted one against any other.

```
>>> from yade import plot
>>> from pprint import pprint
>>> plot.resetData()
>>> plot.addData(a=1)
>>> plot.addData(b=2)
>>> plot.addData(a=3,b=4)
>>> pprint(plot.data)
{'a': [1, nan, 3], 'b': [nan, 2, 4]}
```

Some sequence types can be given to addData; they will be saved in synthesized columns for individual components.

```
>>> plot.resetData()
>>> plot.addData(c=Vector3(5,6,7),d=Matrix3(8,9,10, 11,12,13, 14,15,16))
>>> pprint(plot.data)
{'c_x': [5.0],
 'c_y': [6.0],
 'c_z': [7.0],
 'd_xx': [8.0],
 'd_xy': [9.0],
 'd_xz': [10.0],
 'd_yy': [12.0],
 'd_yz': [11.0],
 'd_zx': [14.0],
 'd_zy': [15.0],
 'd_zz': [16.0]}
```

yade.plot.**addAutoData**()

Add data by evaluating contents of *plot.plots*. Expressions rasing exceptions will be handled gracefully, but warning is printed for each.

```
>>> from yade import plot
>>> from pprint import pprint
>>> O.reset()
>>> plot.resetData()
>>> plot.plots={'O.iter':('O.time',None,'numParticles=len(O.bodies)')}
>>> plot.addAutoData()
>>> pprint(plot.data)
{'O.iter': [0], 'O.time': [0.0], 'numParticles': [0]}
```

Note that each item in *plot.plots* can be

- an expression to be evaluated (using the `eval` builtin);
- `name=expression` string, where `name` will appear as label in plots, and expression will be evaluated each time;
- a dictionary-like object – current keys are labels of plots and current values are added to *plot.data*. The contents of the dictionary can change over time, in which case new lines will be created as necessary.

A simple simulation with plot can be written in the following way; note how the energy plot is specified.

```
>>> from yade import plot, utils
>>> plot.plots={'i=O.iter':(O.energy,None,'total energy=O.energy.total()')}
>>> # we create a simple simulation with one ball falling down
>>> plot.resetData()
>>> O.bodies.append(utils.sphere((0,0,0),1))
0
>>> O.dt=utils.PWaveTimeStep()
>>> O.engines=[
...    ForceResetter(),
...    GravityEngine(gravity=(0,0,-10),warnOnce=False),
...    NewtonIntegrator(damping=.4,kinSplit=True),
...    # get data required by plots at every step
...    PyRunner(command='yade.plot.addAutoData()',iterPeriod=1,initRun=True)
... ]
>>> O.trackEnergy=True
>>> O.run(2,True)
>>> pprint(plot.data)
{'gravWork': [0.0, -25.13274...],
 'i': [0, 1],
 'kinRot': [0.0, 0.0],
 'kinTrans': [0.0, 7.5398...],
 'nonviscDamp': [0.0, 10.0530...],
 'total energy': [0.0, -7.5398...]}
```

`yade.plot.saveGnuplot`(*baseName*, *term='wxt'*, *extension=None*, *timestamp=False*, *comment=None*, *title=None*, *varData=False*)

Save data added with *plot.addData* into (compressed) file and create .gnuplot file that attempts to mimick plots specified with *plot.plots*.

Parameters

- **baseName** – used for creating baseName.gnuplot (command file for gnuplot), associated `baseName.data.bz2` (data) and output files (if applicable) in the form `baseName.[plot number].extension`
- **term** – specify the gnuplot terminal; defaults to `x11`, in which case gnuplot will draw persistent windows to screen and terminate; other useful terminals are `png`, `cairopdf` and so on
- **extension** – extension for `baseName` defaults to terminal name; fine for png for example; if you use `cairopdf`, you should also say `extension='pdf'` however

- **timestamp** (*bool*) – append numeric time to the basename
- **varData** (*bool*) – whether file to plot will be declared as variable or be in-place in the plot expression
- **comment** – a user comment (may be multiline) that will be embedded in the control file

Returns name of the gnuplot file created.

yade.plot.**saveDataTxt**(*fileName*, *vars=None*)

Save plot data into a (optionally compressed) text file. The first line contains a comment (starting with #) giving variable name for each of the columns. This format is suitable for being loaded for further processing (outside yade) with `numpy.genfromtxt` function, which recognizes those variable names (creating numpy array with named entries) and handles decompression transparently.

```
>>> from yade import plot
>>> from pprint import pprint
>>> plot.reset()
>>> plot.addData(a=1,b=11,c=21,d=31)  # add some data here
>>> plot.addData(a=2,b=12,c=22,d=32)
>>> pprint(plot.data)
{'a': [1, 2], 'b': [11, 12], 'c': [21, 22], 'd': [31, 32]}
>>> plot.saveDataTxt('/tmp/dataFile.txt.bz2',vars=('a','b','c'))
>>> import numpy
>>> d=numpy.genfromtxt('/tmp/dataFile.txt.bz2',dtype=None,names=True)
>>> d['a']
array([1, 2])
>>> d['b']
array([11, 12])
```

Parameters

- **fileName** – file to save data to; if it ends with `.bz2` / `.gz`, the file will be compressed using bzip2 / gzip.
- **vars** – Sequence (tuple/list/set) of variable names to be saved. If `None` (default), all variables in *plot.plot* are saved.

yade.plot.**savePlotSequence**(*fileBase*, *stride=1*, *imgRatio=(5, 7)*, *title=None*, *titleFrames=20*, *lastFrames=30*)

Save sequence of plots, each plot corresponding to one line in history. It is especially meant to be used for *utils.makeVideo*.

Parameters

- **stride** – only consider every stride-th line of history (default creates one frame per each line)
- **title** – Create title frame, where lines of title are separated with newlines (`\n`) and optional subtitle is separated from title by double newline.
- **titleFrames** (*int*) – Create this number of frames with title (by repeating its filename), determines how long the title will stand in the movie.
- **lastFrames** (*int*) – Repeat the last frame this number of times, so that the movie does not end abruptly.

Returns List of filenames with consecutive frames.

9.7 yade.polyhedra_utils module

Auxiliary functions for polyhedra

yade.polyhedra_utils.**fillBox**(*mincoord, maxcoord, material, sizemin=[1, 1, 1], sizemax=[1, 1, 1], ratio=[0, 0, 0], seed=None, mask=1*)
fill box [mincoord, maxcoord] by non-overlaping polyhedrons with random geometry and sizes within the range (uniformly distributed) :param Vector3 mincoord: first corner :param Vector3 maxcoord: second corner :param Vector3 sizemin: minimal size of bodies :param Vector3 sizemax: maximal size of bodies :param Vector3 ratio: scaling ratio :param float seed: random seed

yade.polyhedra_utils.**fillBoxByBalls**(*mincoord, maxcoord, material, sizemin=[1, 1, 1], sizemax=[1, 1, 1], ratio=[0, 0, 0], seed=None, mask=1, numpoints=60*)

yade.polyhedra_utils.**polyhedra**(*material, size=Vector3(1,1,1), seed=None, v=[], mask=1, fixed=False, color=[-1, -1, -1]*)
create polyhedra, one can specify vertices directly, or leave it empty for random shape.

Parameters

- **material** (Material) – material of new body
- **size** (*Vector3*) – size of new body (see Polyhedra docs)
- **seed** (*float*) – seed for random operations
- **v** (*[Vector3]*) – list of body vertices (see Polyhedra docs)

yade.polyhedra_utils.**polyhedraSnubCube**(*radius, material, centre, mask=1*)

yade.polyhedra_utils.**polyhedraTruncIcosaHed**(*radius, material, centre, mask=1*)

yade.polyhedra_utils.**polyhedralBall**(*radius, N, material, center, mask=1*)
creates polyhedra having N vertices and resembling sphere

Parameters

- **radius** (*float*) – ball radius
- **N** (*int*) – number of vertices
- **material** (Material) – material of new body
- **center** (*Vector3*) – center of the new body

yade.polyhedra_utils.**randomColor**(*seed=None*)

yade._polyhedra_utils.**MaxCoord**(*(Shape)arg1, (State)arg2*) → Vector3
returns max coordinates

yade._polyhedra_utils.**MinCoord**(*(Shape)arg1, (State)arg2*) → Vector3
returns min coordinates

yade._polyhedra_utils.**PWaveTimeStep**() → float
Get timestep accoring to the velocity of P-Wave propagation; computed from sphere radii, rigidities and masses.

yade._polyhedra_utils.**PrintPolyhedra**(*(Shape)arg1*) → None
Print list of vertices sorted according to polyhedrons facets.

yade._polyhedra_utils.**PrintPolyhedraActualPos**(*(Shape)arg1, (State)arg2*) → None
Print list of vertices sorted according to polyhedrons facets.

yade._polyhedra_utils.**SieveCurve**() → None
save sieve curve coordinates into file

yade._polyhedra_utils.**SieveSize**(*(Shape)arg1*) → float
returns approximate sieve size of polyhedron

yade._polyhedra_utils.**SizeOfPolyhedra**(*(Shape)arg1*) → Vector3
returns max, middle an min size in perpendicular directions

yade._polyhedra_utils.**SizeRatio**() → None
save sizes of polyhedra into file

yade._polyhedra_utils.**Split**(*(Body)arg1, (Vector3)arg2, (Vector3)arg3*) → None
split polyhedron perpendicularly to given direction through given point

yade._polyhedra_utils.**convexHull**(*(object)arg1*) → bool

yade._polyhedra_utils.**do_Polyhedras_Intersect**(*(Shape)arg1*, *(Shape)arg2*, *(State)arg3*, *(State)arg4*) → bool

check polyhedras intersection

yade._polyhedra_utils.**fillBoxByBalls_cpp**(*(Vector3)arg1*, *(Vector3)arg2*, *(Vector3)arg3*, *(Vector3)arg4*, *(Vector3)arg5*, *(int)arg6*, *(Material)arg7*, *(int)arg8*) → object

Generate non-overlaping 'spherical' polyhedrons in box

yade._polyhedra_utils.**fillBox_cpp**(*(Vector3)arg1*, *(Vector3)arg2*, *(Vector3)arg3*, *(Vector3)arg4*, *(Vector3)arg5*, *(int)arg6*, *(Material)arg7*) → object

Generate non-overlaping polyhedrons in box

9.8 yade.post2d module

Module for 2d postprocessing, containing classes to project points from 3d to 2d in various ways, providing basic but flexible framework for extracting arbitrary scalar values from bodies/interactions and plotting the results. There are 2 basic components: flatteners and extractors.

The algorithms operate on bodies (default) or interactions, depending on the `intr` parameter of post2d.data.

9.8.1 Flatteners

Instance of classes that convert 3d (model) coordinates to 2d (plot) coordinates. Their interface is defined by the *post2d.Flatten* class (`__call__`, `planar`, `normal`).

9.8.2 Extractors

Callable objects returning scalar or vector value, given a body/interaction object. If a 3d vector is returned, Flattener.planar is called, which should return only in-plane components of the vector.

9.8.3 Example

This example can be found in examples/concrete/uniax-post.py

```
from yade import post2d
import pylab # the matlab-like interface of matplotlib

O.load('/tmp/uniax-tension.xml.bz2')

# flattener that project to the xz plane
flattener=post2d.AxisFlatten(useRef=False,axis=1)
# return scalar given a Body instance
extractDmg=lambda b: b.state.normDmg
# will call flattener.planar implicitly
# the same as: extractVelocity=lambda b: flattener.planar(b,b.state.vel)
extractVelocity=lambda b: b.state.vel

# create new figure
pylab.figure()
# plot raw damage
post2d.plot(post2d.data(extractDmg,flattener))

# plot smooth damage into new figure
pylab.figure(); ax,map=post2d.plot(post2d.data(extractDmg,flattener,stDev=2e-3))
```

```
# show color scale
pylab.colorbar(map,orientation='horizontal')

# raw velocity (vector field) plot
pylab.figure(); post2d.plot(post2d.data(extractVelocity,flattener))

# smooth velocity plot; data are sampled at regular grid
pylab.figure(); ax,map=post2d.plot(post2d.data(extractVelocity,flattener,stDev=1e-3))
# save last (current) figure to file
pylab.gcf().savefig('/tmp/foo.png')

# show the figures
pylab.show()
```

class yade.post2d.**AxisFlatten**(*inherits Flatten*)

__init__()
:param bool useRef: use reference positions rather than actual positions (only meaningful when operating on Bodies) :param {0,1,2} axis: axis normal to the plane; the return value will be simply position with this component dropped.

normal()

planar()

class yade.post2d.**CylinderFlatten**(*inherits Flatten*)
Class for converting 3d point to 2d based on projection onto plane from circle. The y-axis in the projection corresponds to the rotation axis; the x-axis is distance form the axis.

__init__()
:param useRef: (bool) use reference positions rather than actual positions :param axis: axis of the cylinder, {0,1,2}

normal()

planar()

class yade.post2d.**Flatten**
Abstract class for converting 3d point into 2d. Used by post2d.data2d.

normal()
Given position and vector value, return lenght of the vector normal to the flat plane.

planar()
Given position and vector value, project the vector value to the flat plane and return its 2 in-plane components.

class yade.post2d.**HelixFlatten**(*inherits Flatten*)
Class converting 3d point to 2d based on projection from helix. The y-axis in the projection corresponds to the rotation axis

__init__()
:param bool useRef: use reference positions rather than actual positions :param (ϑmin,ϑmax) thetaRange: bodies outside this range will be discarded :param float dH_dTheta: inclination of the spiral (per radian) :param {0,1,2} axis: axis of rotation of the spiral :param float periodStart: height of the spiral for zero angle

normal()

planar()

yade.post2d.**data**(*extractor*, *flattener*, *intr=False*, *onlyDynamic=True*, *stDev=None*, *relThreshold=3.0*, *perArea=0*, *div=(50, 50)*, *margin=(0, 0)*, *radius=1*)
Filter all bodies/interactions, project them to 2d and extract required scalar value; return either discrete array of positions and values, or smoothed data, depending on whether the stDev value is specified.

The `intr` parameter determines whether we operate on bodies or interactions; the extractor provided should expect to receive body/interaction.

Parameters

- **`extractor`** (*callable*) – receives *Body* (or *Interaction*, if `intr` is `True`) instance, should return scalar, a 2-tuple (vector fields) or None (to skip that body/interaction)
- **`flattener`** (*callable*) – *post2d.Flatten* instance, receiving body/interaction, returns its 2d coordinates or `None` (to skip that body/interaction)
- **`intr`** (*bool*) – operate on interactions rather than bodies
- **`onlyDynamic`** (*bool*) – skip all non-dynamic bodies
- **`stDev`** (*float/None*) – standard deviation for averaging, enables smoothing; `None` (default) means raw mode, where discrete points are returned
- **`relThreshold`** (*float*) – threshold for the gaussian weight function relative to stDev (smooth mode only)
- **`perArea`** (*int*) – if 1, compute weightedSum/weightedArea rather than weighted average (weightedSum/sumWeights); the first is useful to compute average stress; if 2, compute averages on subdivision elements, not using weight function
- **`div`** (*(int,int)*) – number of cells for the gaussian grid (smooth mode only)
- **`margin`** (*(float,float)*) – x,y margins around bounding box for data (smooth mode only)
- **`radius`** (*float/callable*) – Fallback value for radius (for raw plotting) for non-spherical bodies or interactions; if a callable, receives body/interaction and returns radius

Returns dictionary

Returned dictionary always containing keys 'type' (one of 'rawScalar','rawVector','smoothScalar','smoothVector', depending on value of smooth and on return value from extractor), 'x', 'y', 'bbox'.

Raw data further contains 'radii'.

Scalar fields contain 'val' (value from *extractor*), vector fields have 'valX' and 'valY' (2 components returned by the *extractor*).

`yade.post2d.plot(`*data*, *axes=None*, *alpha=0.5*, *clabel=True*, *cbar=False*, *aspect='equal'*, ***kw*`)`

Given output from post2d.data, plot the scalar as discrete or smooth plot.

For raw discrete data, plot filled circles with radii of particles, colored by the scalar value.

For smooth discrete data, plot image with optional contours and contour labels.

For vector data (raw or smooth), plot quiver (vector field), with arrows colored by the magnitude.

Parameters

- **`axes`** – matplotlib.axesinstance where the figure will be plotted; if None, will be created from scratch.
- **`data`** – value returned by *post2d.data*
- **`clabel`** (*bool*) – show contour labels (smooth mode only), or annotate cells with numbers inside (with perArea==2)
- **`cbar`** (*bool*) – show colorbar (equivalent to calling pylab.colorbar(mappable) on the returned mappable)

Returns tuple of `(axes,mappable)`; mappable can be used in further calls to pylab.colorbar.

9.9 yade.qt module

9.10 yade.timing module

Functions for accessing timing information stored in engines and functors.

See *Timing* section of the programmer's manual, wiki page for some examples.

`yade.timing.reset()`
Zero all timing data.

`yade.timing.stats()`
Print summary table of timing information from engines and functors. Absolute times as well as percentages are given. Sample output:

```
Name                                                    Count                 Time            Rel. time
-------------------------------------------------------------------------------------------------------
ForceResetter                                       102               2150us                0.02%
"collider"                                            5              64200us                0.60%
InteractionLoop                                     102           10571887us               98.49%
"combEngine"                                        102               8362us                0.08%
"newton"                                            102              73166us                0.68%
"cpmStateUpdater"                                     1               9605us                0.09%
PyRunner                                              1                136us                0.00%
"plotDataCollector"                                   1                291us                0.00%
TOTAL                                                             10733564us              100.00%
```

sample output (compiled with -DENABLE_PROFILING=1 option):

```
Name                                                    Count                 Time            Rel. time
-------------------------------------------------------------------------------------------------------
ForceResetter                                       102               2150us                0.02%
"collider"                                            5              64200us                0.60%
InteractionLoop                                     102           10571887us               98.49%
  Ig2_Sphere_Sphere_ScGeom                      1222186            1723168us                 16.30%
    Ig2_Sphere_Sphere_ScGeom                      1222186            1723168us                 100.00%
  Ig2_Facet_Sphere_ScGeom                           753               1157us                 0.01%
    Ig2_Facet_Sphere_ScGeom                           753               1157us                 100.00%
  Ip2_CpmMat_CpmMat_CpmPhys                       11712              26015us                 0.25%
    end of Ip2_CpmPhys                              11712              26015us                 100.00%
  Ip2_FrictMat_CpmMat_FrictPhys                       0                  0us                 0.00%
  Law2_ScGeom_CpmPhys_Cpm                       3583872            4819289us                 45.59%
    GO A                                          1194624            1423738us                  29.54%
    GO B                                          1194624            1801250us                  37.38%
    rest                                          1194624            1594300us                  33.08%
    TOTAL                                         3583872            4819289us                 100.00%
  Law2_ScGeom_FrictPhys_CundallStrack                 0                  0us                 0.00%
"combEngine"                                        102               8362us                0.08%
"newton"                                            102              73166us                0.68%
"cpmStateUpdater"                                     1               9605us                0.09%
PyRunner                                              1                136us                0.00%
"plotDataCollector"                                   1                291us                0.00%
TOTAL                                                             10733564us              100.00%
```

9.11 yade.utils module

Heap of functions that don't (yet) fit anywhere else.

Devs: please DO NOT ADD more functions here, it is getting too crowded!

`yade.utils.NormalRestitution2DampingRate(`*en*`)`
Compute the normal damping rate as a function of the normal coefficient of restitution e_n. For

$e_n \in \langle 0,1 \rangle$ damping rate equals

$$-\frac{\log e_n}{\sqrt{e_n^2+\pi^2}}$$

yade.utils.**SpherePWaveTimeStep**(*radius*, *density*, *young*)
Compute P-wave critical timestep for a single (presumably representative) sphere, using formula for P-Wave propagation speed $\Delta t_c = \frac{r}{\sqrt{E/\rho}}$. If you want to compute minimum critical timestep for all spheres in the simulation, use *utils.PWaveTimeStep* instead.

```
>>> SpherePWaveTimeStep(1e-3,2400,30e9)
2.8284271247461903e-07
```

class yade.utils.**TableParamReader**
Class for reading simulation parameters from text file.

Each parameter is represented by one column, each parameter set by one line. Colums are separated by blanks (no quoting).

First non-empty line contains column titles (without quotes). You may use special column named 'description' to describe this parameter set; if such colum is absent, description will be built by concatenating column names and corresponding values (`param1=34,param2=12.22,param4=foo`)

- from columns ending in `!` (the `!` is not included in the column name)
- from all columns, if no columns end in `!`.

Empty lines within the file are ignored (although counted); `#` starts comment till the end of line. Number of blank-separated columns must be the same for all non-empty lines.

A special value `=` can be used instead of parameter value; value from the previous non-empty line will be used instead (works recursively).

This class is used by *utils.readParamsFromTable*.

__init__()
Setup the reader class, read data into memory.

paramDict()
Return dictionary containing data from file given to constructor. Keys are line numbers (which might be non-contiguous and refer to real line numbers that one can see in text editors), values are dictionaries mapping parameter names to their values given in the file. The special value '=' has already been interpreted, `!` (bangs) (if any) were already removed from column titles, `description` column has already been added (if absent).

class yade.utils.**UnstructuredGrid**
EXPERIMENTAL. Class representing triangulated FEM-like unstructured grid. It is used for transfereing data from ad to YADE and external FEM program. The main purpose of this class is to store information about individual grid vertices/nodes coords (since facets stores only coordinates of vertices in local coords) and to avaluate and/or apply nodal forces from contact forces (from actual contact force and contact point the force is distributed to nodes using linear approximation).

TODO rewrite to C++ TODO better docs

Parameters

- **vertices** (dict) – dict of {internal vertex label:vertex}, e.g. {5:(0,0,0),22:(0,1,0),23:(1,0,0)}
- **connectivityTable** (dict) – dict of {internal element label:[indices of vertices]}, e.g. {88:[5,22,23]}

build()

getForcesOfNodes()
Computes forces for each vertex/node. The nodal force is computed from contact force and contact point using linear approximation

resetForces()

setPositionsOfNodes()
Sets new position of nodes and also updates all elements in the simulation

:param [Vector3] newPoss: list of new positions

setup()
Sets new information to receiver

:param dict vertices: see constructor for explanation :param dict connectivityTable: see constructor for explanation :param bool toSimulation: if new information should be inserted to Yade simulation (create new bodies or not) :param [[int]]|None bodies: list of list of bodies indices to be appended as clumps (thus no contact detection is done within one body)

toSimulation()
Insert all elements to Yade simulation

updateElements()
Updates positions of all elements in the simulation

yade.utils.**aabbDim**(*cutoff=0.0*, *centers=False*)
Return dimensions of the axis-aligned bounding box, optionally with relative part *cutoff* cut away.

yade.utils.**aabbExtrema2d**(*pts*)
Return 2d bounding box for a sequence of 2-tuples.

yade.utils.**aabbWalls**(*extrema=None*, *thickness=0*, *oversizeFactor=1.5*, ***kw*)
Return 6 boxes that will wrap existing packing as walls from all sides; extrema are extremal points of the Aabb of the packing (will be calculated if not specified) thickness is wall thickness (will be 1/10 of the X-dimension if not specified) Walls will be enlarged in their plane by oversizeFactor. returns list of 6 wall Bodies enclosing the packing, in the order minX,maxX,minY,maxY,minZ,maxZ.

yade.utils.**avgNumInteractions**(*cutoff=0.0*, *skipFree=False*, *considerClumps=False*)
Return average numer of interactions per particle, also known as *coordination number* Z. This number is defined as

$$\mathsf{Z} = 2\mathsf{C}/\mathsf{N}$$

where C is number of contacts and N is number of particles. When clumps are present, number of particles is the sum of standalone spheres plus the sum of clumps. Clumps are considered in the calculation if cutoff != 0 or skipFree = True. If cutoff=0 (default) and skipFree=False (default) one needs to set considerClumps=True to consider clumps in the calculation.

With *skipFree*, particles not contributing to stable state of the packing are skipped, following equation (8) given in *[Thornton2000]*:

$$\mathsf{Z}_m = \frac{2\mathsf{C} - \mathsf{N}_1}{\mathsf{N} - \mathsf{N}_0 - \mathsf{N}_1}$$

Parameters

- **cutoff** – cut some relative part of the sample's bounding box away.
- **skipFree** – see above.
- **considerClumps** – also consider clumps if cutoff=0 and skipFree=False; for further explanation see above.

yade.utils.**box**(*center*, *extents*, *orientation=Quaternion((1, 0, 0), 0)*, *dynamic=None*, *fixed=False*, *wire=False*, *color=None*, *highlight=False*, *material=-1*, *mask=1*)
Create box (cuboid) with given parameters.

Parameters **extents** (*Vector3*) – half-sizes along x,y,z axes

See *utils.sphere*'s documentation for meaning of other parameters.

yade.utils.**chainedCylinder**(*begin=Vector3(0, 0, 0)*, *end=Vector3(1, 0, 0)*, *radius=0.2*, *dynamic=None*, *fixed=False*, *wire=False*, *color=None*, *highlight=False*, *material=-1*, *mask=1*)
Create and connect a chainedCylinder with given parameters. The shape generated by repeted calls of this function is the Minkowski sum of polyline and sphere.

Parameters

- **radius** (*Real*) – radius of sphere in the Minkowski sum.
- **begin** (*Vector3*) – first point positioning the line in the Minkowski sum
- **last** (*Vector3*) – last point positioning the line in the Minkowski sum

In order to build a correct chain, last point of element of rank N must correspond to first point of element of rank N+1 in the same chain (with some tolerance, since bounding boxes will be used to create connections.

Returns Body object with the *ChainedCylinder shape*.

class yade.utils.**clumpTemplate**

Create a clump template by a list of relative radii and a list of relative positions. Both lists must have the same length.

Parameters

- **relRadii** (*[float,float,...]*) – list of relative radii (minimum length = 2)
- **relPositions** (*[Vector3,Vector3,...]*) – list of relative positions (minimum length = 2)

yade.utils.**defaultMaterial**()

Return default material, when creating bodies with *utils.sphere* and friends, material is unspecified and there is no shared material defined yet. By default, this function returns:

```
.. code-block:: python
```

FrictMat(density=1e3,young=1e7,poisson=.3,frictionAngle=.5,label='defaultMat')

yade.utils.**facet**(*vertices*, *dynamic=None*, *fixed=True*, *wire=True*, *color=None*, *highlight=False*, *noBound=False*, *material=-1*, *mask=1*, *chain=-1*)

Create facet with given parameters.

Parameters

- **vertices** (*[Vector3,Vector3,Vector3]*) – coordinates of vertices in the global coordinate system.
- **wire** (*bool*) – if `True`, facets are shown as skeleton; otherwise facets are filled
- **noBound** (*bool*) – set *Body.bounded*
- **color** (*Vector3-or-None*) – color of the facet; random color will be assigned if `None`.

See *utils.sphere*'s documentation for meaning of other parameters.

yade.utils.**fractionalBox**(*fraction=1.0*, *minMax=None*)

retrurn (min,max) that is the original minMax box (or aabb of the whole simulation if not specified) linearly scaled around its center to the fraction factor

yade.utils.**gridConnection**(*id1*, *id2*, *radius*, *wire=False*, *color=None*, *highlight=False*, *material=-1*, *mask=1*, *cellDist=None*)

yade.utils.**gridNode**(*center*, *radius*, *dynamic=None*, *fixed=False*, *wire=False*, *color=None*, *highlight=False*, *material=-1*)

yade.utils.**loadVars**(*mark=None*)

Load variables from *utils.saveVars*, which are saved inside the simulation. If `mark==None`, all save variables are loaded. Otherwise only those with the mark passed.

yade.utils.**makeVideo**(*frameSpec*, *out*, *renameNotOverwrite=True*, *fps=24*, *kbps=6000*, *bps=None*)

Create a video from external image files using mencoder. Two-pass encoding using the default mencoder codec (mpeg4) is performed, running multi-threaded with number of threads equal to number of OpenMP threads allocated for Yade.

Parameters

- **frameSpec** – wildcard | sequence of filenames. If list or tuple, filenames to be encoded in given order; otherwise wildcard understood by mencoder's mf:// URI option (shell wildcards such as `/tmp/snap-*.png` or and printf-style pattern like `/tmp/snap-%05d.png`)
- **out** (*str*) – file to save video into
- **renameNotOverwrite** (*bool*) – if True, existing same-named video file will have *-number* appended; will be overwritten otherwise.
- **fps** (*int*) – Frames per second (`-mf fps=…`)
- **kbps** (*int*) – Bitrate (`-lavcopts vbitrate=…`) in kb/s

yade.utils.**perpendicularArea**(*axis*)
Return area perpendicular to given axis (0=x,1=y,2=z) generated by bodies for which the function consider returns True (defaults to returning True always) and which is of the type *Sphere*.

yade.utils.**plotDirections**(*aabb=()*, *mask=0*, *bins=20*, *numHist=True*, *noShow=False*, *sphSph=False*)
Plot 3 histograms for distribution of interaction directions, in yz,xz and xy planes and (optional but default) histogram of number of interactions per body. If sphSph only sphere-sphere interactions are considered for the 3 directions histograms.

Returns If *noShow* is `False`, displays the figure and returns nothing. If *noShow*, the figure object is returned without being displayed (works the same way as *plot.plot*).

yade.utils.**plotNumInteractionsHistogram**(*cutoff=0.0*)
Plot histogram with number of interactions per body, optionally cutting away *cutoff* relative axis-aligned box from specimen margin.

yade.utils.**polyhedron**(*vertices*, *dynamic=True*, *fixed=False*, *wire=True*, *color=None*, *highlight=False*, *noBound=False*, *material=-1*, *mask=1*, *chain=-1*)
Create polyhedron with given parameters.

Parameters **vertices** (*[[Vector3]]*) – coordinates of vertices in the global coordinate system.

See *utils.sphere*'s documentation for meaning of other parameters.

yade.utils.**psd**(*bins=5*, *mass=True*, *mask=-1*)
Calculates particle size distribution.

Parameters

- **bins** (*int*) – number of bins
- **mass** (*bool*) – if true, the mass-PSD will be calculated
- **mask** (*int*) – *Body.mask* for the body

Returns

- binsSizes: list of bin's sizes
- binsProc: how much material (in percents) are in the bin, cumulative
- binsSumCum: how much material (in units) are in the bin, cumulative

binsSizes, binsProc, binsSumCum

yade.utils.**randomColor**()
Return random Vector3 with each component in interval 0…1 (uniform distribution)

yade.utils.**randomizeColors**(*onlyDynamic=False*)
Assign random colors to *Shape::color*.

If onlyDynamic is true, only dynamic bodies will have the color changed.

yade.utils.**readParamsFromTable**(*tableFileLine=None*, *noTableOk=True*, *unknownOk=False*, ***kw*)
Read parameters from a file and assign them to ___builtin___ variables.

The format of the file is as follows (commens starting with # and empty lines allowed):

```
# commented lines allowed anywhere
name1 name2 … # first non-blank line are column headings
                        # empty line is OK, with or without comment
val1    val2    … # 1st parameter set
val2    val2    … # 2nd
…
```

Assigned tags (the `description` column is synthesized if absent,see *utils.TableParamReader*);

O.tags['description']=... # assigns the description column; might be synthesized O.tags['params']="name1=val1,name2=val2,..." # all explicitly assigned parameters O.tags['defaultParams']="unassignedName1=defaultValue1,..." # parameters that were left at their defaults O.tags['d.id']=O.tags['id']+'.'+O.tags['description'] O.tags['id.d']=O.tags['description']+'.'+O.tags['id']

All parameters (default as well as settable) are saved using *utils.saveVars*`('table')`.

Parameters

- **tableFile** – text file (with one value per blank-separated columns)
- **tableLine** (*int*) – number of line where to get the values from
- **noTableOk** (*bool*) – if False, raise exception if the file cannot be open; use default values otherwise
- **unknownOk** (*bool*) – do not raise exception if unknown column name is found in the file, and assign it as well

Returns number of assigned parameters

`yade.utils.replaceCollider(`*colliderEngine*`)`
Replaces collider (Collider) engine with the engine supplied. Raises error if no collider is in engines.

`yade.utils.runningInBatch()`
Tell whether we are running inside the batch or separately.

`yade.utils.saveVars(`*mark=''*, *loadNow=True*, ***kw*`)`
Save passed variables into the simulation so that it can be recovered when the simulation is loaded again.

For example, variables *a*, *b* and *c* are defined. To save them, use:

```
>>> saveVars('something',a=1,b=2,c=3)
>>> from yade.params.something import *
>>> a,b,c
(1, 2, 3)
```

those variables will be save in the .xml file, when the simulation itself is saved. To recover those variables once the .xml is loaded again, use ‘‘loadVars(‘something’)‘‘and they will be defined in the yade.params.*mark* module. The *loadNow* parameter calls *utils.loadVars* after saving automatically. If ‘something’ already exists, given variables will be inserted.

`yade.utils.sphere(`*center*, *radius*, *dynamic=None*, *fixed=False*, *wire=False*, *color=None*, *highlight=False*, *material=-1*, *mask=1*`)`
Create sphere with given parameters; mass and inertia computed automatically.

Last assigned material is used by default (*material* = -1), and utils.defaultMaterial() will be used if no material is defined at all.

Parameters

- **center** (*Vector3*) – center
- **radius** (*float*) – radius
- **dynamic** (*float*) – deprecated, see “fixed”
- **fixed** (*float*) – generate the body with all DOFs blocked?
- **material** –

specify *Body.material*; different types are accepted:

- int: O.materials[material] will be used; as a special case, if material==-1 and there is no shared materials defined, utils.defaultMaterial() will be assigned to O.materials[0]
- string: label of an existing material that will be used
- *Material* instance: this instance will be used
- callable: will be called without arguments; returned Material value will be used (Material factory object, if you like)

- **mask** (*int*) – *Body.mask* for the body
- **wire** – display as wire sphere?
- **highlight** – highlight this body in the viewer?
- **Vector3-or-None** – body's color, as normalized RGB; random color will be assigned if None.

Returns A Body instance with desired characteristics.

Creating default shared material if none exists neither is given:

```
>>> O.reset()
>>> from yade import utils
>>> len(O.materials)
0
>>> s0=utils.sphere([2,0,0],1)
>>> len(O.materials)
1
```

Instance of material can be given:

```
>>> s1=utils.sphere([0,0,0],1,wire=False,color=(0,1,0),material=ElastMat(young=30e9,density=2e3))
>>> s1.shape.wire
False
>>> s1.shape.color
Vector3(0,1,0)
>>> s1.mat.density
2000.0
```

Material can be given by label:

```
>>> O.materials.append(FrictMat(young=10e9,poisson=.11,label='myMaterial'))
1
>>> s2=utils.sphere([0,0,2],1,material='myMaterial')
>>> s2.mat.label
'myMaterial'
>>> s2.mat.poisson
0.11
```

Finally, material can be a callable object (taking no arguments), which returns a Material instance. Use this if you don't call this function directly (for instance, through yade.pack.randomDensePack), passing only 1 *material* parameter, but you don't want material to be shared.

For instance, randomized material properties can be created like this:

```
>>> import random
>>> def matFactory(): return ElastMat(young=1e10*random.random(),density=1e3+1e3*random.random())
...
>>> s3=utils.sphere([0,2,0],1,material=matFactory)
>>> s4=utils.sphere([1,2,0],1,material=matFactory)
```

yade.utils.**tetra**(*vertices*, *strictCheck=True*, *dynamic=True*, *fixed=False*, *wire=True*, *color=None*, *highlight=False*, *noBound=False*, *material=-1*, *mask=1*, *chain=-1*)

Create tetrahedron with given parameters.

Parameters

- **vertices** (*[Vector3,Vector3,Vector3,Vector3]*) – coordinates of vertices in the global coordinate system.
- **strictCheck** (*bool*) – checks vertices order, raise RuntimeError for negative volume

See *utils.sphere*'s documentation for meaning of other parameters.

yade.utils.**tetraPoly**(*vertices*, *dynamic=True*, *fixed=False*, *wire=True*, *color=None*, *highlight=False*, *noBound=False*, *material=-1*, *mask=1*, *chain=-1*)

Create tetrahedron (actually simple Polyhedra) with given parameters.

Parameters **vertices** (*[Vector3,Vector3,Vector3,Vector3]*) – coordinates of vertices in the global coordinate system.

See *utils.sphere*'s documentation for meaning of other parameters.

yade.utils.**trackPerfomance**(*updateTime=5*)

Track perfomance of a simulation. (Experimental) Will create new thread to produce some plots. Useful for track perfomance of long run simulations (in bath mode for example).

yade.utils.**typedEngine**(*name*)

Return first engine from current O.engines, identified by its type (as string). For example:

```
>>> from yade import utils
>>> O.engines=[InsertionSortCollider(),NewtonIntegrator(),GravityEngine()]
>>> utils.typedEngine("NewtonIntegrator") == O.engines[1]
True
```

yade.utils.**uniaxialTestFeatures**(*filename=None*, *areaSections=10*, *axis=-1*, *distFactor=2.2*, ***kw*)

Get some data about the current packing useful for uniaxial test:

1. Find the dimensions that is the longest (uniaxial loading axis)
2. Find the minimum cross-section area of the specimen by examining several (areaSections) sections perpendicular to axis, computing area of the convex hull for each one. This will work also for non-prismatic specimen.
3. Find the bodies that are on the negative/positive boundary, to which the straining condition should be applied.

Parameters

- **filename** – if given, spheres will be loaded from this file (ASCII format); if not, current simulation will be used.
- **areaSection** (*float*) – number of section that will be used to estimate cross-section
- **axis** (*{0,1,2}*) – if given, force strained axis, rather than computing it from predominant length

Returns dictionary with keys **negIds**, **posIds**, **axis**, **area**.

Warning: The function *utils.approxSectionArea* uses convex hull algorithm to find the area, but the implementation is reported to be *buggy* (bot works in some cases). Always check this number, or fix the convex hull algorithm (it is documented in the source, see py/_utils.cpp).

yade.utils.**vmData**()

Return memory usage data from Linux's /proc/[pid]/status, line VmData.

yade.utils.**voxelPorosityTriaxial**(*triax*, *resolution=200*, *offset=0*)

Calculate the porosity of a sample, given the TriaxialCompressionEngine.

A function *utils.voxelPorosity* is invoked, with the volume of a box enclosed by TriaxialCompressionEngine walls. The additional parameter offset allows using a smaller volume inside the box,

where each side of the volume is at offset distance from the walls. By this way it is possible to find a more precise porosity of the sample, since at walls' contact the porosity is usually reduced.

A recommended value of offset is bigger or equal to the average radius of spheres inside.

The value of resolution depends on size of spheres used. It can be calibrated by invoking voxelPorosityTriaxial with offset=0 and comparing the result with TriaxialCompressionEngine.porosity. After calibration, the offset can be set to radius, or a bigger value, to get the result.

Parameters

- **triax** – the TriaxialCompressionEngine handle
- **resolution** – voxel grid resolution
- **offset** – offset distance

Returns the porosity of the sample inside given volume

Example invocation:

```
from yade import utils
rAvg=0.03
TriaxialTest(numberOfGrains=200,radiusMean=rAvg).load()
O.dt=-1
O.run(1000)
O.engines[4].porosity
0.44007807740143889
utils.voxelPorosityTriaxial(O.engines[4],200,0)
0.44055412500000002
utils.voxelPorosityTriaxial(O.engines[4],200,rAvg)
0.36798199999999998
```

yade.utils.**waitIfBatch**()
Block the simulation if running inside a batch. Typically used at the end of script so that it does not finish prematurely in batch mode (the execution would be ended in such a case).

yade.utils.**wall**(*position*, *axis*, *sense=0*, *color=None*, *material=-1*, *mask=1*)
Return ready-made wall body.

Parameters

- **position** (*float-or-Vector3*) – center of the wall. If float, it is the position along given axis, the other 2 components being zero
- **axis** (*{0,1,2}*) – orientation of the wall normal (0,1,2) for x,y,z (sc. planes yz, xz, xy)
- **sense** (*{-1,0,1}*) – sense in which to interact (0: both, -1: negative, +1: positive; see *Wall*)

See *utils.sphere*'s documentation for meaning of other parameters.

yade.utils.**xMirror**(*half*)
Mirror a sequence of 2d points around the x axis (changing sign on the y coord). The sequence should start up and then it will wrap from y downwards (or vice versa). If the last point's x coord is zero, it will not be duplicated.

yade._utils.**PWaveTimeStep**() → float
Get timestep accoring to the velocity of P-Wave propagation; computed from sphere radii, rigidities and masses.

yade._utils.**RayleighWaveTimeStep**() → float
Determination of time step according to Rayleigh wave speed of force propagation.

yade._utils.**TetrahedronCentralInertiaTensor**(*(object)arg1*) → Matrix3
TODO

yade._utils.**TetrahedronInertiaTensor**(*(object)arg1*) → Matrix3
TODO

yade._utils.**TetrahedronSignedVolume**(*(object)arg1*) → float
TODO

yade._utils.**TetrahedronVolume**(*(object)arg1*) → float
TODO

yade._utils.**TetrahedronWithLocalAxesPrincipal**(*(Body)arg1*) → Quaternion
TODO

yade._utils.**aabbExtrema**([*(float)cutoff=0.0*[, *(bool)centers=False*]]) → tuple
Return coordinates of box enclosing all bodies

Parameters

- **centers** (*bool*) – do not take sphere radii in account, only their centroids
- **cutoff** (*float ⟨0…1⟩*) – relative dimension by which the box will be cut away at its boundaries.

Returns (lower corner, upper corner) as (Vector3,Vector3)

yade._utils.**angularMomentum**([*(Vector3)origin=Vector3(0, 0, 0)*]) → Vector3
TODO

yade._utils.**approxSectionArea**(*(float)arg1*, *(int)arg2*) → float
Compute area of convex hull when when taking (swept) spheres crossing the plane at coord, perpendicular to axis.

yade._utils.**bodyNumInteractionsHistogram**(*(tuple)aabb*) → tuple

yade._utils.**bodyStressTensors**() → list
Compute and return a table with per-particle stress tensors. Each tensor represents the average stress in one particle, obtained from the contour integral of applied load as detailed below. This definition is considering each sphere as a continuum. It can be considered exact in the context of spheres at static equilibrium, interacting at contact points with negligible volume changes of the solid phase (this last assumption is not restricting possible deformations and volume changes at the packing scale).

Proof:

First, we remark the identity: $\sigma_{ij} = \delta_{ik}\sigma_{kj} = x_{i,k}\sigma_{kj} = (x_i\sigma_{kj})_{,k} - x_i\sigma_{kj,k}$.

At equilibrium, the divergence of stress is null: $\sigma_{kj,k} = 0$. Consequently, after divergence theorem: $\frac{1}{V}\int_V \sigma_{ij}dV = \frac{1}{V}\int_V (x_i\sigma_{kj})_{,k}dV = \frac{1}{V}\int_{\partial V} x_i\sigma_{kj}n_k dS = \frac{1}{V}\sum_b x_i^b f_j^b$.

The last equality is implicitely based on the representation of external loads as Dirac distributions whose zeros are the so-called *contact points*: 0-sized surfaces on which the *contact forces* are applied, located at x_i in the deformed configuration.

A weighted average of per-body stresses will give the average stress inside the solid phase. There is a simple relation between the stress inside the solid phase and the stress in an equivalent continuum in the absence of fluid pressure. For porosity n, the relation reads: $\sigma_{ij}^{equ.} = (1-n)\sigma_{ij}^{solid}$.

This last relation may not be very useful if porosity is not homogeneous. If it happens, one can define the equivalent bulk stress a the particles scale by assigning a volume to each particle. This volume can be obtained from *TesselationWrapper* (see e.g. *[Catalano2014a]*)

yade._utils.**calm**([*(int)mask=-1*]) → None
Set translational and rotational velocities of bodies to zero. Applied to all bodies by default. To calm only some bodies, use mask parameter, it will calm only bodies with groupMask compatible to given value

yade._utils.**coordsAndDisplacements**(*(int)axis*[, *(tuple)Aabb=()*]) → tuple
Return tuple of 2 same-length lists for coordinates and displacements (coordinate minus reference coordinate) along given axis (1st arg); if the Aabb=((x_min,y_min,z_min),(x_max,y_max,z_max)) box is given, only bodies within this box will be considered.

yade._utils.**createInteraction**(*(int)id1*, *(int)id2*) → Interaction
Create interaction between given bodies by hand.

Current engines are searched for *IGeomDispatcher* and *IPhysDispatcher* (might be both hidden in *InteractionLoop*). Geometry is created using `force` parameter of the *geometry dispatcher*, wherefore the interaction will exist even if bodies do not spatially overlap and the functor would return `false` under normal circumstances.

Warning: This function will very likely behave incorrectly for periodic simulations (though it could be extended it to handle it farily easily).

`yade._utils.fabricTensor`([*(bool)splitTensor=False*[, *(bool)revertSign=False*[, *(float)thresholdForce=nan*]]]) → tuple

Compute the fabric tensor of the periodic cell. The original paper can be found in *[Satake1982]*.

Parameters

- **splitTensor** (*bool*) – split the fabric tensor into two parts related to the strong and weak contact forces respectively.
- **revertSign** (*bool*) – it must be set to true if the contact law's convention takes compressive forces as positive.
- **thresholdForce** (*Real*) – if the fabric tensor is split into two parts, a threshold value can be specified otherwise the mean contact force is considered by default. It is worth to note that this value has a sign and the user needs to set it according to the convention adopted for the contact law. To note that this value could be set to zero if one wanted to make distinction between compressive and tensile forces.

`yade._utils.flipCell`([*(Matrix3)flip=Matrix3(0, 0, 0, 0, 0, 0, 0, 0, 0)*]) → Matrix3

Flip periodic cell so that angles between $\mathbb{R}^3$ axes and transformed axes are as small as possible. This function relies on the fact that periodic cell defines by repetition or its corners regular grid of points in $\mathbb{R}^3$; however, all cells generating identical grid are equivalent and can be flipped one over another. This necessiatates adjustment of *Interaction.cellDist* for interactions that cross boundary and didn't before (or vice versa), and re-initialization of collider. The *flip* argument can be used to specify desired flip: integers, each column for one axis; if zero matrix, best fit (minimizing the angles) is computed automatically.

In c++, this function is accessible as `Shop::flipCell`.

Warning: This function is currently broken and should not be used.

`yade._utils.forcesOnCoordPlane`(*(float)arg1*, *(int)arg2*) → Vector3

`yade._utils.forcesOnPlane`(*(Vector3)planePt*, *(Vector3)normal*) → Vector3

Find all interactions deriving from *NormShearPhys* that cross given plane and sum forces (both normal and shear) on them.

Parameters

- **planePt** (*Vector3*) – a point on the plane
- **normal** (*Vector3*) – plane normal (will be normalized).

`yade._utils.getBodyIdsContacts`([*(int)bodyID=0*]) → list

Get a list of body-ids, which contacts the given body.

`yade._utils.getCapillaryStress`([*(float)volume=0*[, *(bool)mindlin=False*]]) → Matrix3

Compute and return Love-Weber capillary stress tensor:

$\sigma_{ij}^{cap} = \frac{1}{V} \sum_b l_i^b f_j^{cap,b}$, where the sum is over all interactions, with l the branch vector (joining centers of the bodies) and f^{cap} is the capillary force. V can be passed to the function. If it is not, it will be equal to one in non-periodic cases, or equal to the volume of the cell in periodic cases. Only the CapillaryPhys interaction type is supported presently. Using this function with physics MindlinCapillaryPhys needs to pass True as second argument.

yade._utils.getDepthProfiles(*(float)volume*, *(int)nCell*, *(float)dz*, *(float)zRef*, *(bool)activateCond*, *(float)radiusPy*) → tuple

Compute and return the particle velocity and solid volume fraction (porosity) depth profile. For each defined cell z, the k component of the average particle velocity reads:

$< v_k >^z = \sum_p V^p v_k^p / \sum_p V^p$,

where the sum is made over the particles contained in the cell, v_k^p is the k component of the velocity associated to particle p, and V^p is the part of the volume of the particle p contained inside the cell. This definition allows to smooth the averaging, and is equivalent to taking into account the center of the particles only when there is a lot of particles in each cell. As for the solid volume fraction, it is evaluated in the same way: for each defined cell z, it reads:

$< \varphi >^z = \frac{1}{V_{cell}} \sum_p V^p$, where V_{cell} is the volume of the cell considered, and V^p is the volume of particle p contained in cell z. This function gives depth profiles of average velocity and solid volume fraction, returning the average quantities in each cell of height dz, from the reference horizontal plane at elevation zRef (input parameter) until the plane of elevation zRef+nCell*dz (input parameters). If the argument activateCond is set to true, do the average only on particles of radius equal to radiusPy (input parameter)

yade._utils.getSpheresMass([*(int)mask=-1*]) → float

Compute the total mass of spheres in the simulation (might crash for now if dynamic bodies are not spheres), mask parameter is considered

yade._utils.getSpheresVolume([*(int)mask=-1*]) → float

Compute the total volume of spheres in the simulation (might crash for now if dynamic bodies are not spheres), mask parameter is considered

yade._utils.getSpheresVolume2D([*(int)mask=-1*]) → float

Compute the total volume of discs in the simulation (might crash for now if dynamic bodies are not discs), mask parameter is considered

yade._utils.getStress([*(float)volume=0*]) → Matrix3

Compute and return Love-Weber stress tensor:

$\sigma_{ij} = \frac{1}{V} \sum_b f_i^b l_j^b$, where the sum is over all interactions, with f the contact force and l the branch vector (joining centers of the bodies). Stress is negativ for repulsive contact forces, i.e. compression. V can be passed to the function. If it is not, it will be equal to the volume of the cell in periodic cases, or to the one deduced from utils.aabbDim() in non-periodic cases.

yade._utils.getStressAndTangent([*(float)volume=0*[, *(bool)symmetry=True*]]) → tuple

Compute overall stress of periodic cell using the same equation as function getStress. In addition, the tangent operator is calculated using the equation published in [Kruyt and Rothenburg1998]_:

$$S_{ijkl} = \frac{1}{V} \sum_c (k_n n_i l_j n_k l_l + k_t t_i l_j t_k l_l)$$

Parameters

- **volume** (*float*) – same as in function getStress
- **symmetry** (*bool*) – make the tensors symmetric.

Returns macroscopic stress tensor and tangent operator as py::tuple

yade._utils.getStressProfile(*(float)volume*, *(int)nCell*, *(float)dz*, *(float)zRef*, *(object)vPartAverageX*, *(object)vPartAverageY*, *(object)vPartAverageZ*) → tuple

Compute and return the stress tensor depth profile, including the contribution from Love-Weber stress tensor and the dynamic stress tensor taking into account the effect of particles inertia. For each defined cell z, the stress tensor reads:

$\sigma_{ij}^z = \frac{1}{V} \sum_c f_i^c l_j^{c,z} - \frac{1}{V} \sum_p m^p u_i'^p u_j'^p$,

where the first sum is made over the contacts which are contained or cross the cell z, f^c is the contact force from particle 1 to particle 2, and l^{c,z} is the part of the branch vector from particle 2 to particle 1, contained in the cell. The second sum is made over the particles, and u'^p is the velocity fluctuations of the particle p with respect to the spatial averaged particle velocity at this point (given as input parameters). The expression of the stress tensor is the same as the one given in getStress plus the inertial contribution. Apart from that, the main difference with getStress stands in the fact that it gives a depth profile of stress tensor, i.e. from the reference horizontal plane at elevation zRef (input parameter) until the plane of elevation zRef+nCell*dz (input parameters), it is computing the stress tensor for each cell of height dz. For the love-Weber stress contribution, the branch vector taken into account in the calculations is only the part of the branch vector contained in the cell considered. To validate the formulation, it has been checked that activating only the Love-Weber stress tensor, and suming all the contributions at the different altitude, we recover the same stress tensor as when using getStress. For my own use, I have troubles with strong overlap between fixed object, so that I made a condition to exclude the contribution to the stress tensor of the fixed objects, this can be desactivated easily if needed (and should be desactivated for the comparison with getStress).

yade._utils.**getViscoelasticFromSpheresInteraction**(*(float)tc*, *(float)en*, *(float)es*) → dict

Attention! The function is deprecated! Compute viscoelastic interaction parameters from analytical solution of a pair spheres collision problem:

$$k_n = \frac{m}{t_c^2}\left(\pi^2 + (\ln e_n)^2\right)$$

$$c_n = -\frac{2m}{t_c}\ln e_n$$

$$k_t = \frac{2}{7}\frac{m}{t_c^2}\left(\pi^2 + (\ln e_t)^2\right)$$

$$c_t = -\frac{2}{7}\frac{m}{t_c}\ln e_t$$

where k_n, c_n are normal elastic and viscous coefficients and k_t, c_t shear elastic and viscous coefficients. For details see *[Pournin2001]*.

Parameters

- **m** (*float*) – sphere mass m
- **tc** (*float*) – collision time t_c
- **en** (*float*) – normal restitution coefficient e_n
- **es** (*float*) – tangential restitution coefficient e_s

Returns dictionary with keys **kn** (the value of k_n), **cn** (c_n), **kt** (k_t), **ct** (c_t).

yade._utils.**growParticle**(*(int)bodyID*, *(float)multiplier*[, *(bool)updateMass=True*]) → None

Change the size of a single sphere (to be implemented: single clump). If updateMass=True, then the mass is updated.

yade._utils.**growParticles**(*(float)multiplier*[, *(bool)updateMass=True*[, *(bool)dynamicOnly=True*]]) → None

Change the size of spheres and sphere clumps by the multiplier. If updateMass=True, then the mass and inertia are updated. dynamicOnly=True will select dynamic bodies.

yade._utils.**highlightNone**() → None

Reset *highlight* on all bodies.

yade._utils.**inscribedCircleCenter**(*(Vector3)v1*, *(Vector3)v2*, *(Vector3)v3*) → Vector3

Return center of inscribed circle for triangle given by its vertices *v1*, *v2*, *v3*.

yade._utils.**interactionAnglesHistogram**(*(int)axis*, *(int)mask*, *(int)bins*, *(tuple)aabb*, *(bool)sphSph*, *(float)minProjLen*) → tuple

yade._utils.**intrsOfEachBody**() → list
returns list of lists of interactions of each body

yade._utils.**kineticEnergy**([*(bool)findMaxId=False*]) → object
Compute overall kinetic energy of the simulation as

$$\sum \frac{1}{2}\left(m_i v_i^2 + \boldsymbol{\omega}(\mathbf{I}\boldsymbol{\omega}^T)\right).$$

For *aspherical* bodies, the inertia tensor $\mathbf{I}$ is transformed to global frame, before multiplied by $\boldsymbol{\omega}$, therefore the value should be accurate.

yade._utils.**maxOverlapRatio**() → float
Return maximum overlap ration in interactions (with *ScGeom*) of two *spheres*. The ratio is computed as $\frac{u_N}{2(r_1 r_2)/r_1+r_2}$, where u_N is the current overlap distance and r_1, r_2 are radii of the two spheres in contact.

yade._utils.**momentum**() → Vector3
TODO

yade._utils.**negPosExtremeIds**(*(int)axis*, *(float)distFactor*) → tuple
Return list of ids for spheres (only) that are on extremal ends of the specimen along given axis; distFactor multiplies their radius so that sphere that do not touch the boundary coordinate can also be returned.

yade._utils.**normalShearStressTensors**([*(bool)compressionPositive=False*[, *(bool)splitNormalTensor=False*[, *(float)thresholdForce=nan*]]]) → tuple
Compute overall stress tensor of the periodic cell decomposed in 2 parts, one contributed by normal forces, the other by shear forces. The formulation can be found in *[Thornton2000]*, eq. (3):

$$\boldsymbol{\sigma}_{ij} = \frac{2}{V}\sum RN\mathbf{n}_i\mathbf{n}_j + \frac{2}{V}\sum RT\mathbf{n}_i\mathbf{t}_j$$

where V is the cell volume, R is "contact radius" (in our implementation, current distance between particle centroids), $\mathbf{n}$ is the normal vector, $\mathbf{t}$ is a vector perpendicular to $\mathbf{n}$, N and T are norms of normal and shear forces.

Parameters

- **splitNormalTensor** (*bool*) – if true the function returns normal stress tensor split into two parts according to the two subnetworks of strong an weak forces.
- **thresholdForce** (*Real*) – threshold value according to which the normal stress tensor can be split (e.g. a zero value would make distinction between tensile and compressive forces).

yade._utils.**numIntrsOfEachBody**() → list
returns list of number of interactions of each body

yade._utils.**pointInsidePolygon**(*(tuple)arg1*, *(object)arg2*) → bool

yade._utils.**porosity**([*(float)volume=-1*]) → float
Compute packing porosity $\frac{V-V_s}{V}$ where V is overall volume and V_s is volume of spheres.

Parameters **volume** (*float*) – overall volume V. For periodic simulations, current volume of the *Cell* is used. For aperiodic simulations, the value deduced from utils.aabbDim() is used. For compatibility reasons, positive values passed by the user are also accepted in this case.

yade._utils.**ptInAABB**(*(Vector3)arg1*, *(Vector3)arg2*, *(Vector3)arg3*) → bool
Return True/False whether the point p is within box given by its min and max corners

yade._utils.**scalarOnColorScale**(*(float)arg1*, *(float)arg2*, *(float)arg3*) → Vector3

yade._utils.**setContactFriction**(*(float)angleRad*) → None
Modify the friction angle (in radians) inside the material classes and existing contacts. The friction for non-dynamic bodies is not modified.

yade._utils.**setNewVerticesOfFacet**(*(Body)b*, *(Vector3)v1*, *(Vector3)v2*, *(Vector3)v3*) → None
Sets new vertices (in global coordinates) to given facet.

yade._utils.**setRefSe3**() → None
Set reference *positions* and *orientations* of all *bodies* equal to their current *positions* and *orientations*.

yade._utils.**shiftBodies**(*(list)ids*, *(Vector3)shift*) → float
Shifts bodies listed in ids without updating their velocities.

yade._utils.**spiralProject**(*(Vector3)pt*, *(float)dH_dTheta*[, *(int)axis=2*[, *(float)periodStart=nan*[, *(float)theta0=0*]]]) → tuple

yade._utils.**sumFacetNormalForces**(*(object)ids*[, *(int)axis=-1*]) → float
Sum force magnitudes on given bodies (must have *shape* of the *Facet* type), considering only part of forces perpendicular to each *facet's* face; if *axis* has positive value, then the specified axis (0=x, 1=y, 2=z) will be used instead of facet's normals.

yade._utils.**sumForces**(*(list)ids*, *(Vector3)direction*) → float
Return summary force on bodies with given *ids*, projected on the *direction* vector.

yade._utils.**sumTorques**(*(list)ids*, *(Vector3)axis*, *(Vector3)axisPt*) → float
Sum forces and torques on bodies given in *ids* with respect to axis specified by a point *axisPt* and its direction *axis*.

yade._utils.**totalForceInVolume**() → tuple
Return summed forces on all interactions and average isotropic stiffness, as tuple (Vector3,float)

yade._utils.**unbalancedForce**([*(bool)useMaxForce=False*]) → float
Compute the ratio of mean (or maximum, if *useMaxForce*) summary force on bodies and mean force magnitude on interactions. For perfectly static equilibrium, summary force on all bodies is zero (since forces from interactions cancel out and induce no acceleration of particles); this ratio will tend to zero as simulation stabilizes, though zero is never reached because of finite precision computation. Sufficiently small value can be e.g. 1e-2 or smaller, depending on how much equilibrium it should be.

yade._utils.**voidratio2D**([*(float)zlen=1*]) → float
Compute 2D packing void ratio $\frac{V-V_s}{V_s}$ where V is overall volume and V_s is volume of disks.

Parameters **zlen** (*float*) – length in the third direction.

yade._utils.**voxelPorosity**([*(int)resolution=200*[, *(Vector3)start=Vector3(0, 0, 0)*[, *(Vector3)end=Vector3(0, 0, 0)*]]]) → float
Compute packing porosity $\frac{V-V_v}{V}$ where V is a specified volume (from start to end) and V_v is volume of voxels that fall inside any sphere. The calculation method is to divide whole volume into a dense grid of voxels (at given resolution), and count the voxels that fall inside any of the spheres. This method allows one to calculate porosity in any given sub-volume of a whole sample. It is properly excluding part of a sphere that does not fall inside a specified volume.

Parameters

- **resolution** (*int*) – voxel grid resolution, values bigger than resolution=1600 require a 64 bit operating system, because more than 4GB of RAM is used, a resolution=800 will use 500MB of RAM.
- **start** (*Vector3*) – start corner of the volume.
- **end** (*Vector3*) – end corner of the volume.

yade._utils.**wireAll**() → None
Set *Shape::wire* on all bodies to True, rendering them with wireframe only.

yade._utils.**wireNoSpheres**() → None
Set *Shape::wire* to True on non-spherical bodies (*Facets*, *Walls*).

yade._utils.**wireNone**() → None
Set *Shape::wire* on all bodies to False, rendering them as solids.

9.12 yade.ymport module

Import geometry from various formats ('import' is python keyword, hence the name 'ymport').

`yade.ymport.ele`(*nodeFileName*, *eleFileName*, *shift=(0, 0, 0)*, *scale=1.0*, ***kw*)
Import tetrahedral mesh from .ele file, return list of created tetrahedrons.

Parameters

- **nodeFileName** (*string*) – name of .node file
- **eleFileName** (*string*) – name of .ele file
- **shift** (*(float,float,float)|Vector3*) – (X,Y,Z) parameter moves the specimen.
- **scale** (*float*) – factor scales the given data.
- ****kw** – (unused keyword arguments) is passed to *utils.polyhedron*

`yade.ymport.gengeo`(*mntable*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, ***kw*)
Imports geometry from LSMGenGeo library and creates spheres. Since 2012 the package is available in Debian/Ubuntu and known as python-demgengeo http://packages.qa.debian.org/p/python-demgengeo.html

Parameters

***mntable*: mntable** object, which creates by LSMGenGeo library, see example

***shift*: [float,float,float]** [X,Y,Z] parameter moves the specimen.

***scale*: float** factor scales the given data.

*****kw*: (unused keyword arguments)** is passed to *utils.sphere*

LSMGenGeo library allows one to create pack of spheres with given [Rmin:Rmax] with null stress inside the specimen. Can be useful for Mining Rock simulation.

Example: examples/packs/packs.py, usage of LSMGenGeo library in examples/test/genCylLSM.py.

- https://answers.launchpad.net/esys-particle/+faq/877
- http://www.access.edu.au/lsmgengeo_python_doc/current/pythonapi/html/GenGeo-module.html
- https://svn.esscc.uq.edu.au/svn/esys3/lsm/contrib/LSMGenGeo/

`yade.ymport.gengeoFile`(*fileName='file.geo'*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, *orientation=Quaternion((1, 0, 0), 0)*, ***kw*)
Imports geometry from LSMGenGeo .geo file and creates spheres. Since 2012 the package is available in Debian/Ubuntu and known as python-demgengeo http://packages.qa.debian.org/p/python-demgengeo.html

Parameters

***filename*: string** file which has 4 colums [x, y, z, radius].

***shift*: Vector3** Vector3(X,Y,Z) parameter moves the specimen.

***scale*: float** factor scales the given data.

***orientation*: quaternion** orientation of the imported geometry

*****kw*: (unused keyword arguments)** is passed to *utils.sphere*

Returns list of spheres.

LSMGenGeo library allows one to create pack of spheres with given [Rmin:Rmax] with null stress inside the specimen. Can be useful for Mining Rock simulation.

Example: examples/packs/packs.py, usage of LSMGenGeo library in examples/test/genCylLSM.py.

- https://answers.launchpad.net/esys-particle/+faq/877

- http://www.access.edu.au/lsmgengeo_python_doc/current/pythonapi/html/GenGeo-module.html
- https://svn.esscc.uq.edu.au/svn/esys3/lsm/contrib/LSMGenGeo/

`yade.ymport.gmsh`(*meshfile='file.mesh'*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, *orientation=Quaternion((1, 0, 0), 0)*, ***kw*)

Imports geometry from mesh file and creates facets.

Parameters

***shift*: [float,float,float]** [X,Y,Z] parameter moves the specimen.

***scale*: float** factor scales the given data.

***orientation*: quaternion** orientation of the imported mesh

*****kw*: (unused keyword arguments)** is passed to *utils.facet*

Returns list of facets forming the specimen.

mesh files can be easily created with GMSH. Example added to examples/regular-sphere-pack/regular-sphere-pack.py

Additional examples of mesh-files can be downloaded from http://www-roc.inria.fr/gamma/download/download.php

`yade.ymport.gts`(*meshfile*, *shift=(0, 0, 0)*, *scale=1.0*, ***kw*)

Read given meshfile in gts format.

Parameters

***meshfile*: string** name of the input file.

***shift*: [float,float,float]** [X,Y,Z] parameter moves the specimen.

***scale*: float** factor scales the given data.

*****kw*: (unused keyword arguments)** is passed to *utils.facet*

Returns list of facets.

`yade.ymport.iges`(*fileName*, *shift=(0, 0, 0)*, *scale=1.0*, *returnConnectivityTable=False*, ***kw*)

Import triangular mesh from .igs file, return list of created facets.

Parameters

- **`fileName`** (*string*) – name of iges file
- **`shift`** (*(float,float,float)|Vector3*) – (X,Y,Z) parameter moves the specimen.
- **`scale`** (*float*) – factor scales the given data.
- **`**kw`** – (unused keyword arguments) is passed to *utils.facet*
- **`returnConnectivityTable`** (*bool*) – if True, apart from facets returns also nodes (list of (x,y,z) nodes coordinates) and elements (list of (id1,id2,id3) element nodes ids). If False (default), returns only facets

`yade.ymport.stl`(*file*, *dynamic=None*, *fixed=True*, *wire=True*, *color=None*, *highlight=False*, *noBound=False*, *material=-1*)

Import geometry from stl file, return list of created facets.

`yade.ymport.text`(*fileName*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, ***kw*)

Load sphere coordinates from file, returns a list of corresponding bodies; that may be inserted to the simulation with O.bodies.append().

Parameters

- **`filename`** (*string*) – file which has 4 colums [x, y, z, radius].
- **`shift`** (*[float,float,float]*) – [X,Y,Z] parameter moves the specimen.
- **`scale`** (*float*) – factor scales the given data.
- **`**kw`** – (unused keyword arguments) is passed to *utils.sphere*

Returns list of spheres.

Lines starting with # are skipped

yade.ymport.**textClumps**(*fileName*, *shift=Vector3(0, 0, 0)*, *discretization=0*, *orientation=Quaternion((1, 0, 0), 0)*, *scale=1.0*, ***kw*)

Load clumps-members from file, insert them to the simulation.

Parameters

- **filename** (*str*) – file name
- **format** (*str*) – the name of output format. Supported *x_y_z_r'(default), 'x_y_z_r_clumpId*
- **shift** (*[float,float,float]*) – [X,Y,Z] parameter moves the specimen.
- **scale** (*float*) – factor scales the given data.
- ****kw** – (unused keyword arguments) is passed to *utils.sphere*

Returns list of spheres.

Lines starting with # are skipped

yade.ymport.**textExt**(*fileName*, *format='x_y_z_r'*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, ***kw*)

Load sphere coordinates from file in specific format, returns a list of corresponding bodies; that may be inserted to the simulation with O.bodies.append().

Parameters

- **filename** (*str*) – file name
- **format** (*str*) – the name of output format. Supported *x_y_z_r'(default), 'x_y_z_r_matId*
- **shift** (*[float,float,float]*) – [X,Y,Z] parameter moves the specimen.
- **scale** (*float*) – factor scales the given data.
- ****kw** – (unused keyword arguments) is passed to *utils.sphere*

Returns list of spheres.

Lines starting with # are skipped

yade.ymport.**unv**(*fileName*, *shift=(0, 0, 0)*, *scale=1.0*, *returnConnectivityTable=False*, ***kw*)

Import geometry from unv file, return list of created facets.

param string fileName name of unv file

param (float,float,float)|Vector3 shift (X,Y,Z) parameter moves the specimen.

param float scale factor scales the given data.

param **kw (unused keyword arguments) is passed to *utils.facet*

param bool returnConnectivityTable if True, apart from facets returns also nodes (list of (x,y,z) nodes coordinates) and elements (list of (id1,id2,id3) element nodes ids). If False (default), returns only facets

unv files are mainly used for FEM analyses (are used by OOFEM and Abaqus), but triangular elements can be imported as facets. These files cen be created e.g. with open-source free software Salome.

Example: examples/test/unv-read/unvRead.py.

Chapter 10

Parallel hierarchical multiscale modeling of granular media by coupling FEM and DEM with open-source codes Escript and YADE

Authors: Ning Guo and Jidong Zhao

Institution: Hong Kong University of Science and Technology

Escript download page: https://launchpad.net/escript-finley

mpi4py download page (optional, require MPI): https://bitbucket.org/mpi4py/mpi4py

Tested platforms: Desktop with Ubuntu 10.04, 32 bit; Server with Ubuntu 12.04, 14.04, 64 bit; Cluster with Centos 6.2, 6.5, 64 bit;

10.1 Introduction

The code is built upon two open source packages: Yade for DEM modules and Escript for FEM modules. It implements the hierarchical multiscale model (FEMxDEM) for simulating the boundary value problem (BVP) of granular media. FEM is used to discretize the problem domain. Each Gauss point of the FEM mesh is embedded a representative volume element (RVE) packing simulated by DEM which returns local material constitutive responses to FEM. Typically, hundreds to thousands of RVEs are involved in a medium-sized problem which is critically time consuming. Hence parallelization is achieved in the code through either multiprocessing on a supercomputer or mpi4py on a HPC cluster (require MPICH or Open MPI). The MPI implementation in the code is quite experimental. The "mpipool.py" is contributed by Lisandro Dalcin, the author of mpi4py package. Please refer to the examples for the usage of the code.

10.2 Work on the YADE side

The version of YADE should be at least rev3682 in which Bruno added the stringToScene function. Before installation, I added some functions to the source code (in "yade" subfolder). But only one function ("Shop::getStressAndTangent" in "./pkg/dem/Shop.cpp") is necessary for the FEMxDEM coupling, which returns the stress tensor and the tangent operator of a discrete packing. The former is homogenized using the Love's formula and the latter is homogenized as the elastic modulus. After installation and we get the executable file: yade-versionNo. We then generate a .py file linked to the executable

file by "ln yade-versionNo yadeimport.py". This .py file will serve as a wrapped library of YADE. Later on, we will import all YADE functions into the python script through "from yadeimport import *" (see simDEM.py file).

Open a python terminal. Make sure you can run

```
import sys
sys.path.append('where you put yadeimport.py')
from yadeimport import *
Omega().load('your initial RVE packing, e.g. 0.yade.gz')
```

If you are successful, you should also be able to run

```
from simDEM import *
```

10.3 Work on the Escript side

No particular requirement. But make sure the modules are callable in python, which means the main folder of Escript should be in your PYTHONPATH and LD_LIBRARY_PATH. The modules are wrapped as a class in msFEM*.py.

Open a python terminal. Make sure you can run:

```
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE
from esys.finley import Rectangle
```

(Note: Escript is used for the current implementation. It can be replaced by any other FEM package provided with python bindings, e.g. FEniCS (http://fenicsproject.org). But the interface files "msFEM*.py" need to be modified.)

10.4 Example tests

After Steps 1 & 2, one should be able to run all the scripts for the multiscale analysis. The initial RVE packing (default name "0.yade.gz") should be provided by the user (e.g. using YADE to prepare a consolidated packing), which will be loaded by simDEM.py when the problem is initialized. The sample is initially uniform as long as the same RVE packing is assigned to all the Gauss points in the problem domain. It is also possible for the user to specify different RVEs at different Gauss points to generate an inherently inhomogeneous sample.

While simDEM.py is always required, only one msFEM*.py is needed for a single test. For example, in a 2D (3D) dry test, msFEM2D.py (msFEM3D.py) is needed; similarly for a coupled hydro-mechanical problem (2D only, saturated), msFEMup.py is used which incorporates the u-p formulation. Multiprocessing is used by default. To try MPI parallelization, please set useMPI=True when constructing the problem in the main script. Example tests given in the "example" subfolder are listed below. Note: The initial RVE packing (named 0.yade.gz by default) needs to be generated, e.g. using prepareRVE.py in "example" subfolder for a 2D packing (similarly for 3D).

1. **2D drained biaxial compression test on dry dense sand** (biaxialSmooth.py) *Note*: Test description and result were presented in *[Guo2014]* and *[Guo2014c]*.
2. **2D passive failure under translational mode of dry sand retained by a rigid and frictionless wall** (retainingSmooth.py) *Note:* Rolling resistance model (CohFrictMat) is used in the RVE packing. Test description and result were presented in *[Guo2015]*.
3. **2D half domain footing settlement problem with mesh generated by Gmsh** (footing.py, footing.msh) *Note:* Rolling resistance model (CohFrictMat) is used in the RVE packing. Six-node triangle element is generated by Gmsh with three Gauss points each. Test description and result were presented in *[Guo2015]*.

4. **3D drained conventional triaxial compression test on dry dense sand using MPI parallelism** (triaxialRough.py) *Note 1:* The simulation is very time consuming. It costs ~4.5 days on one node using multiprocessing (16 processes, 2.0 GHz CPU). When useMPI is switched to True (as in the example script) and four nodes are used (80 processes, 2.2 GHz CPU), the simulation costs less than 24 hours. The speedup is about 4.4 in our test. *Note 2:* When MPI is used, mpi4py is required to be installed. The MPI implementation can be either MPICH or Open MPI. The file "mpipool.py" should also be placed in the main folder. Our test is based on openmpi-1.6.5. This is an on-going work. Test description and result will be presented later.
5. **2D globally undrained biaxial compression test on saturated dense sand with changing permeability using MPI parallelism** (undrained.py) *Note:* This is an on-going work. Test description and result will be presented later.

10.5 Disclaim

This work extensively utilizes and relies on some third-party packages as mentioned above. Their contributions are acknowledged. Feel free to use and redistribute the code. But there is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Chapter 11

Acknowledging Yade

In order to let users cite Yade consistently in publications, we provide a list of bibliographic references for the different parts of the documentation, as in the citation model pushed by CGAL. This way of acknowledging Yade is also a way to make developments and documentation of Yade more attractive for researchers, who are evaluated on the basis of citations of their work by others. We therefore kindly ask users to cite Yade as accurately as possible in their papers. A more detailed discussion of the citation model and its application to Yade can be found here.

If new developments are presented and explained in self-contained papers (at the end of a PhD, typically), we will be glad to include them in the documentation and to reference them in the list below. Any other substantial improvement is welcome and can be discussed in the yade-dev mailing list.

11.1 Citing the Yade Project as a whole (the lazy citation method)

If it is not possible to choose the right chapter (but please try), you may cite the documentation *[yade:doc]* as a whole:

22. Šmilauer, E. Catalano, B. Chareyre, S. Dorofeenko, J. Duriez, A. Gladky, J. Kozicki, C. Modenese, L. Scholtès, L. Sibille, J. Stránský, and K. Thoeni, Yade Documentation (V. Šmilauer, ed.), The Yade Project, 1st ed., 2010. http://yade-dem.org/doc/.

11.2 Citing chapters of Yade Documentation

The first edition of Yade documentation is seen as a collection with the three volumes (or "chapters") listed below, also provided as bibtex entries. Please cite the chapter that is the most relevant in your case. For instance, a paper using one of the documented contact laws will cite the reference documentation *[yade:reference]*; if programing concepts are discussed, Yade's manual *[yade:manual]* will be cited; the theoretical background *[yade:background]* can be used as the refence for contact detection, time-step determination, or periodic boundary conditions.

- **The reference documentation includes details on equations and algorithms found at the highest lev**

  22. Šmilauer, E. Catalano, B. Chareyre, S. Dorofeenko, J. Duriez, A. Gladky, J. Kozicki, C. Modenese, L. Scholtès, L. Sibille, J. Stránský, and K. Thoeni, "Yade Reference Documentation," in Yade Documentation (V. Šmilauer, ed.), The Yade Project, 1st ed., 2010. http://yade-dem.org/doc/.

- **Software design, user's and programmer's manuals are in (pdf version):**

  22. Šmilauer, A. Gladky, J. Kozicki, C. Modenese, and J. Stránský, "Yade Using and Programming," in Yade Documentation (V. Šmilauer, ed.), The Yade Project, 1st ed., 2010. http://yade-dem.org/doc/.

- **Fundamentals of the DEM as implemented in Yade are explained in (pdf version):**

22. Šmilauer and B. Chareyre, "Yade Dem Formulation", in Yade Documentation (V. Šmilauer, ed.), The Yade Project, 1st ed., 2010. http://yade-dem.org/doc/formulation.html.

Chapter 12

Publications on Yade

This page should be a relatively complete list of publications on Yade itself or done with Yade. If you publish something, do not hesitate to add it on the list. If you don't have direct access to the source code, please send the reference (as a bibtex item) to the editorial board. If PDF is freely available, add url for direct fulltext downlad. If not, consider uploading fulltext in PDF, either to Yade wiki or to other website, if legally permitted.

The first section gives the references that we kindly ask you to use for citing Yade in publications, as explained in the "Acknowledging Yade" section.

Note: This file is generated from doc/yade-articles.bib, doc/yade-conferences.bib, doc/yade-theses.bib and doc/yade-docref.bib.

12.1 Citing Yade

Corresponding bibtex entries here. Pdf versions are available for each of them. See also "Acknowledging Yade".

12.2 Journal articles

12.3 Conference materials and book chapters

12.4 Master and PhD theses

Chapter 13

References

All external articles referenced in Yade documentation.

Note: This file is generated from doc/references.bib.

Chapter 14

Indices and tables

- genindex
- modindex
- search

Bibliography

[yade:background] V. Šmilauer, B. Chareyre (2010), **Yade dem formulation**. In *Yade Documentation* (V. Šmilauer, ed.), The Yade Project , 1st ed. (fulltext) (http://yade-dem.org/doc/formulation.html)

[yade:doc] V. Šmilauer, E. Catalano, B. Chareyre, S. Dorofeenko, J. Duriez, A. Gladky, J. Kozicki, C. Modenese, L. Scholtès, L. Sibille, J. Stránský, K. Thoeni (2010), **Yade Documentation**. The Yade Project. (http://yade-dem.org/doc/)

[yade:manual] V. Šmilauer, A. Gladky, J. Kozicki, C. Modenese, J. Stránský (2010), **Yade, using and programming**. In *Yade Documentation* (V. Šmilauer, ed.), The Yade Project , 1st ed. (fulltext) (http://yade-dem.org/doc/)

[yade:reference] V. Šmilauer, E. Catalano, B. Chareyre, S. Dorofeenko, J. Duriez, A. Gladky, J. Kozicki, C. Modenese, L. Scholtès, L. Sibille, J. Stránský, K. Thoeni (2010), **Yade Reference Documentation**. In *Yade Documentation* (V. Šmilauer, ed.), The Yade Project , 1st ed. (fulltext) (http://yade-dem.org/doc/)

[Bance2014] Bance, S., Fischbacher, J., Schrefl, T., Zins, I., Rieger, G., Cassignol, C. (2014), **Micromagnetics of shape anisotropy based permanent magnets**. *Journal of Magnetism and Magnetic Materials* (363), pages 121–124.

[Bonilla2015] Bonilla-Sierra, V., Scholtès, L., Donzé, F.V., Elmouttie, M.K. (2015), **Rock slope stability analysis using photogrammetric data and dfn–dem modelling**. *Acta Geotechnica*, pages 1-15. DOI 10.1007/s11440-015-0374-z (fulltext)

[Boon2012a] Boon,C.W., Houlsby, G.T., Utili, S. (2012), **A new algorithm for contact detection between convex polygonal and polyhedral particles in the discrete element method**. *Computers and Geotechnics* (44), pages 73 - 82. DOI 10.1016/j.compgeo.2012.03.012 (fulltext)

[Boon2012b] Boon,C.W., Houlsby, G.T., Utili, S. (2013), **A new contact detection algorithm for three-dimensional non-spherical particles**. *Powder Technology*. DOI 10.1016/j.powtec.2012.12.040 (fulltext)

[Boon2014] Boon, C.W., Houlsby, G.T., Utili, S. (2014), **New insights into the 1963 vajont slide using 2d and 3d distinct-element method analyses**. *Géotechnique* (64), pages 800–816.

[Boon2015] Boon, C.W., Houlsby, G.T., Utili, S. (2015), **A new rock slicing method based on linear programming**. *Computers and Geotechnics* (65), pages 12–29.

[Bourrier2013] Bourrier, F., Kneib, F., Chareyre, B., Fourcaud, T. (2013), **Discrete modeling of granular soils reinforcement by plant roots**. *Ecological Engineering*. DOI 10.1016/j.ecoleng.2013.05.002 (fulltext)

[Bourrier2015] Bourrier, F., Lambert, S., Baroth, J. (2015), **A reliability-based approach for the design of rockfall protection fences**. *Rock Mechanics and Rock Engineering* (48), pages 247–259.

[Catalano2014a] Catalano, E., Chareyre, B., Barthélémy, E. (2014), **Pore-scale modeling of fluid-particles interaction and emerging poromechanical effects**. *International Journal for Numerical and Analytical Methods in Geomechanics* (38), pages 51–71. DOI 10.1002/nag.2198 (fulltext) (http://arxiv.org/pdf/1304.4895.pdf)

[Chareyre2012a] Chareyre, B., Cortis, A., Catalano, E., Barthélemy, E. (2012), **Pore-scale modeling of viscous flow and induced forces in dense sphere packings**. *Transport in Porous Media* (92), pages 473-493. DOI 10.1007/s11242-011-9915-6 (fulltext)

[Chen2007] Chen, F., Drumm, E. C., Guiochon, G. (2007), **Prediction/verification of particle motion in one dimension with the discrete-element method**. *International Journal of Geomechanics, ASCE* (7), pages 344–352. DOI 10.1061/(ASCE)1532-3641(2007)7:5(344)

[Chen2011a] Chen, F., Drumm, E., Guiochon G. (2011), **Coupled discrete element and finite volume solution of two classical soil mechanics problems**. *Computers and Geotechnics*. DOI 10.1016/j.compgeo.2011.03.009 (fulltext)

[Chen2012] Chen, Jingsong, Huang, Baoshan, Chen, Feng, Shu, Xiang (2012), **Application of discrete element method to superpave gyratory compaction**. *Road Materials and Pavement Design* (13), pages 480-500. DOI 10.1080/14680629.2012.694160 (fulltext)

[Chen2014] Chen, J., Huang, B., Shu, X., Hu, C. (2014), **Dem simulation of laboratory compaction of asphalt mixtures using an open source code**. *Journal of Materials in Civil Engineering*.

[Dang2010a] Dang, H. K., Meguid, M. A. (2010), **Algorithm to generate a discrete element specimen with predefined properties**. *International Journal of Geomechanics* (10), pages 85-91. DOI 10.1061/(ASCE)GM.1943-5622.0000028

[Dang2010b] Dang, H. K., Meguid, M. A. (2010), **Evaluating the performance of an explicit dynamic relaxation technique in analyzing non-linear geotechnical engineering problems**. *Computers and Geotechnics* (37), pages 125 - 131. DOI 10.1016/j.compgeo.2009.08.004

[Donze2008] Donzé, F.V. (2008), **Impacts on cohesive frictional geomaterials**. *European Journal of Environmental and Civil Engineering* (12), pages 967–985.

[Duriez2011] Duriez,J., Darve, F., Donzé, F.V. (2011), **A discrete modeling-based constitutive relation for infilled rock joints**. *International Journal of Rock Mechanics & Mining Sciences* (48), pages 458–468. DOI 10.1016/j.ijrmms.2010.09.008

[Duriez2013] Duriez, J., Darve, F., Donzé, F.V. (2013), **Incrementally non-linear plasticity applied to rock joint modelling**. *International Journal for Numerical and Analytical Methods in Geomechanics* (37), pages 453–477. DOI 10.1002/nag.1105 (fulltext)

[Dyck2015] Dyck, N. J, Straatman, A.G. (2015), **A new approach to digital generation of spherical void phase porous media microstructures**. *International Journal of Heat and Mass Transfer* (81), pages 470–477.

[Elias2014] Jan Elias (2014), **Simulation of railway ballast using crushable polyhedral particles**. *Powder Technology* (264), pages 458–465. DOI 10.1016/j.powtec.2014.05.052

[Epifancev2013] Epifancev, K., Nikulin, A., Kovshov, S., Mozer, S., Brigadnov, I. (2013), **Modeling of peat mass process formation based on 3d analysis of the screw machine by the code yade**. *American Journal of Mechanical Engineering* (1), pages 73–75. DOI 10.12691/ajme-1-3-3 (fulltext)

[Epifantsev2012] Epifantsev, K., Mikhailov, A., Gladky, A. (2012), **Proizvodstvo kuskovogo torfa, ekstrudirovanie, forma zakhodnoi i kalibriruyushchei chasti fil'ery matritsy, metod diskretnykh elementov [rus]**. *Mining informational and analytical bulletin (scientific and technical journal)*, pages 212-219.

[Favier2009a] Favier, L., Daudon, D., Donzé, F.V., Mazars, J. (2009), **Predicting the drag coefficient of a granular flow using the discrete element method**. *Journal of Statistical Mechanics: Theory and Experiment* (2009), pages P06012.

[Favier2012] Favier, L., Daudon, D., Donzé, F.V. (2012), **Rigid obstacle impacted by a supercritical cohesive granular flow using a 3d discrete element model**. *Cold Regions Science and Technology* (85), pages 232–241. (fulltext)

[Gladky2014] Gladkyy, Anton, Schwarze, Rüdiger (2014), **Comparison of different capillary bridge models for application in the discrete element method**. *Granular Matter*, pages 1-10. DOI 10.1007/s10035-014-0527-z (fulltext)

[Grujicic2013] Grujicic, M, Snipes, JS, Ramaswami, S, Yavari, R (2013), **Discrete element modeling and analysis of structural collapse/survivability of a building subjected to improvised explosive device (ied) attack**. *Advances in Materials Science and Applications* (2), pages 9–24.

[Guo2014] Guo, Ning, Zhao, Jidong (2014), **A coupled fem/dem approach for hierarchical multiscale modelling of granular media**. *International Journal for Numerical Methods in Engineering* (99), pages 789–818. DOI 10.1002/nme.4702 (fulltext)

[Guo2015] N. Guo, J. Zhao (2015), **Multiscale insights into classical geomechanics problems**. *International Journal for Numerical and Analytical Methods in Geomechanics*. (under review)

[Gusenbauer2012] Gusenbauer, M., Kovacs, A., Reichel, F., Exl, L., Bance, S., Özelt, H., Schrefl, T. (2012), **Self-organizing magnetic beads for biomedical applications**. *Journal of Magnetism and Magnetic Materials* (324), pages 977–982.

[Gusenbauer2014] Gusenbauer, M., Nguyen, H., Reichel, F., Exl, L., Bance, S., Fischbacher, J., Özelt, H., Kovacs, A., Brandl, M., Schrefl, T. (2014), **Guided self-assembly of magnetic beads for biomedical applications**. *Physica B: Condensed Matter* (435), pages 21–24.

[Hadda2013] Hadda, Nejib, Nicot, François, Bourrier, Franck, Sibille, Luc, Radjai, Farhang, Darve, Félix (2013), **Micromechanical analysis of second order work in granular media**. *Granular Matter* (15), pages 221–235. DOI 10.1007/s10035-013-0402-3 (fulltext)

[Hadda2015] Hadda, N., Nicot, F., Wan, R., Darve, F. (2015), **Microstructural self-organization in granular materials during failure**. *Comptes Rendus Mécanique*.

[Harthong2009] Harthong, B., Jerier, J.F., Doremus, P., Imbault, D., Donzé, F.V. (2009), **Modeling of high-density compaction of granular materials by the discrete element method**. *International Journal of Solids and Structures* (46), pages 3357–3364. DOI 10.1016/j.ijsolstr.2009.05.008

[Harthong2012b] Harthong, B., Jerier, J.-F., Richefeu, V., Chareyre, B., Doremus, P., Imbault, D., Donzé, F.V. (2012), **Contact impingement in packings of elastic–plastic spheres, application to powder compaction**. *International Journal of Mechanical Sciences* (61), pages 32–43.

[Hartong2012a] Harthong, B., Scholtès, L., Donzé, F.-V. (2012), **Strength characterization of rock masses, using a coupled dem–dfn model**. *Geophysical Journal International* (191), pages 467–480. DOI 10.1111/j.1365-246X.2012.05642.x (fulltext)

[Hassan2010] Hassan, A., Chareyre, B., Darve, F., Meyssonier, J., Flin, F. (2010 (submitted)), **Microtomography-based discrete element modelling of creep in snow**. *Granular Matter*.

[Hilton2013] Hilton, J. E., Tordesillas, A. (2013), **Drag force on a spherical intruder in a granular bed at low froude number**. *Phys. Rev. E* (88), pages 062203. DOI 10.1103/PhysRevE.88.062203 (fulltext)

[Jerier2009] Jerier, J.-F., Imbault, D.and Donzé, F.V., Doremus, P. (2009), **A geometric algorithm based on tetrahedral meshes to generate a dense polydisperse sphere packing**. *Granular Matter* (11). DOI 10.1007/s10035-008-0116-0

[Jerier2010a] Jerier, J.-F., Richefeu, V., Imbault, D., Donzé, F.V. (2010), **Packing spherical discrete elements for large scale simulations**. *Computer Methods in Applied Mechanics and Engineering*. DOI 10.1016/j.cma.2010.01.016

[Jerier2010b] Jerier, J.-F., Hathong, B., Richefeu, V., Chareyre, B., Imbault, D., Donzé, F.-V., Doremus, P. (2010), **Study of cold powder compaction by using the discrete element method**. *Powder Technology* (In Press). DOI 10.1016/j.powtec.2010.08.056

[Kozicki2006a] Kozicki, J., Tejchman, J. (2006), **2D lattice model for fracture in brittle materials**. *Archives of Hydro-Engineering and Environmental Mechanics* (53), pages 71–88. (fulltext)

[Kozicki2007a] Kozicki, J., Tejchman, J. (2007), **Effect of aggregate structure on fracture process in concrete using 2d lattice model"**. *Archives of Mechanics* (59), pages 365–384. (fulltext)

[Kozicki2008] Kozicki, J., Donzé, F.V. (2008), **A new open-source software developed for numerical simulations using discrete modeling methods**. *Computer Methods in Applied Mechanics and Engineering* (197), pages 4429–4443. DOI 10.1016/j.cma.2008.05.023 (fulltext)

[Kozicki2009] Kozicki, J., Donzé, F.V. (2009), **Yade-open dem: an open-source software using a discrete element method to simulate granular material**. *Engineering Computations* (26), pages 786–805. DOI 10.1108/02644400910985170 (fulltext)

[Lomine2013] Lominé, F., Scholtès, L., Sibille, L., Poullain, P. (2013), **Modelling of fluid-solid interaction in granular media with coupled lb/de methods: application to piping erosion**.

International Journal for Numerical and Analytical Methods in Geomechanics (37), pages 577-596. DOI 10.1002/nag.1109

[Marzougui2015] Marzougui, Donia, Chareyre, Bruno, Chauchat, Julien (2015), **Microscopic origins of shear stress in dense fluid–grain mixtures**. *Granular Matter*, pages 1-13. DOI 10.1007/s10035-015-0560-6 (fulltext)

[Nicot2011] Nicot, F., Hadda, N., Bourrier, F., Sibille, L., Darve, F. (2011), **Failure mechanisms in granular media: a discrete element analysis**. *Granular Matter* (13), pages 255-260. DOI 10.1007/s10035-010-0242-3

[Nicot2012] Nicot, F., Sibille, L., Darve, F. (2012), **Failure in rate-independent granular materials as a bifurcation toward a dynamic regime**. *International Journal of Plasticity* (29), pages 136-154. DOI 10.1016/j.ijplas.2011.08.002

[Nicot2013a] Nicot, F., Hadda, N., Darve, F. (2013), **Second-order work analysis for granular materials using a multiscale approach**. *International Journal for Numerical and Analytical Methods in Geomechanics*. DOI 10.1002/nag.2175

[Nicot2013b] Nicot, F., Hadda, N., Guessasma, M., Fortin, J., Millet, O. (2013), **On the definition of the stress tensor in granular media**. *International Journal of Solids and Structures*. DOI 10.1016/j.ijsolstr.2013.04.001 (fulltext)

[Nitka2015] Nitka, M., Tejchman, J. (2015), **Modelling of concrete behaviour in uniaxial compression and tension with dem**. *Granular Matter*, pages 1–20.

[Puckett2011] Puckett, J.G., Lechenault, F., Daniels, K.E. (2011), **Local origins of volume fraction fluctuations in dense granular materials**. *Physical Review E* (83), pages 041301. DOI 10.1103/PhysRevE.83.041301 (fulltext)

[Sayeed2011] Sayeed, M.A., Suzuki, K., Rahman, M.M., Mohamad, W.H.W., Razlan, M.A., Ahmad, Z., Thumrongvut, J., Seangatith, S., Sobhan, MA, Mofiz, SA, others (2011), **Strength and deformation characteristics of granular materials under extremely low to high confining pressures in triaxial compression**. *International Journal of Civil & Environmental Engineering IJCEE-IJENS* (11).

[Scholtes2009a] Scholtès, L., Chareyre, B., Nicot, F., Darve, F. (2009), **Micromechanics of granular materials with capillary effects**. *International Journal of Engineering Science* (47), pages 64–75. DOI 10.1016/j.ijengsci.2008.07.002 (fulltext)

[Scholtes2009b] Scholtès, L., Hicher, P.-Y., Chareyre, B., Nicot, F., Darve, F. (2009), **On the capillary stress tensor in wet granular materials**. *International Journal for Numerical and Analytical Methods in Geomechanics* (33), pages 1289–1313. DOI 10.1002/nag.767 (fulltext)

[Scholtes2009c] Scholtès, L., Chareyre, B., Nicot, F., Darve, F. (2009), **Discrete modelling of capillary mechanisms in multi-phase granular media**. *Computer Modeling in Engineering and Sciences* (52), pages 297–318. (fulltext)

[Scholtes2010] Scholtès, L., Hicher, P.-Y., Sibille, L. (2010), **Multiscale approaches to describe mechanical responses induced by particle removal in granular materials**. *Comptes Rendus Mécanique* (338), pages 627–638. DOI 10.1016/j.crme.2010.10.003 (fulltext)

[Scholtes2011] Scholtès, L., Donzé, F.V., Khanal, M. (2011), **Scale effects on strength of geomaterials, case study: coal**. *Journal of the Mechanics and Physics of Solids* (59), pages 1131–1146. DOI 10.1016/j.jmps.2011.01.009 (fulltext)

[Scholtes2012] Scholtès, L., Donzé, F.V. (2012), **Modelling progressive failure in fractured rock masses using a 3d discrete element method**. *International Journal of Rock Mechanics and Mining Sciences* (52), pages 18–30. DOI 10.1016/j.ijrmms.2012.02.009 (fulltext)

[Scholtes2013] Scholtès, L., Donzé, F.V. (2013), **A DEM model for soft and hard rocks: role of grain interlocking on strength**. *Journal of the Mechanics and Physics of Solids* (61), pages 352–369. DOI 10.1016/j.jmps.2012.10.005 (fulltext)

[Scholtes2015a] Scholtès, L., Chareyre, B., Michallet, H., Catalano, E., Marzougui, D. (2015), **Modeling wave-induced pore pressure and effective stress in a granular seabed**. *Continuum Mechanics and Thermodynamics* (27), pages 305–323. DOI http://dx.doi.org/10.1007/s00161-014-0377-2

[Scholtes2015b] Scholtès, L., Donzé, F., V. (2015), **A dem analysis of step-path failure in jointed rock slopes**. *Comptes rendus - Mécanique* (343), pages 155–165. DOI http://dx.doi.org/10.1016/j.crme.2014.11.002

[Shiu2008] Shiu, W., Donzé, F.V., Daudeville, L. (2008), **Compaction process in concrete during missile impact: a dem analysis**. *Computers and Concrete* (5), pages 329–342.

[Shiu2009] Shiu, W., Donzé, F.V., Daudeville, L. (2009), **Discrete element modelling of missile impacts on a reinforced concrete target**. *International Journal of Computer Applications in Technology* (34), pages 33–41.

[Sibille2014] Sibille, L., Lominé, F., Poullain, P., Sail, Y., Marot, D. (2014), **Internal erosion in granular media: direct numerical simulations and energy interpretation**. *Hydrological Processes*. DOI 10.1002/hyp.10351 (fulltext) (First published online Oct. 2014)

[Sibille2015] Sibille, L., Hadda, N., Nicot, F., Tordesillas, A., Darve, F. (2015), **Granular plasticity, a contribution from discrete mechanics**. *Journal of the Mechanics and Physics of Solids* (75), pages 119–139. DOI 10.1016/j.jmps.2014.09.010 (fulltext)

[Smilauer2006] Václav Šmilauer (2006), **The splendors and miseries of yade design**. *Annual Report of Discrete Element Group for Hazard Mitigation*. (fulltext)

[Thoeni2013] K. Thoeni, C. Lambert, A. Giacomini, S.W. Sloan (2013), **Discrete modelling of hexagonal wire meshes with a stochastically distorted contact model**. *Computers and Geotechnics* (49), pages 158–169. DOI 10.1016/j.compgeo.2012.10.014 (fulltext)

[Thoeni2014] K. Thoeni, A. Giacomini, C. Lambert, S.W. Sloan, J.P. Carter (2014), **A 3D discrete element modelling approach for rockfall analysis with drapery systems**. *International Journal of Rock Mechanics and Mining Sciences* (68), pages 107–119. DOI 10.1016/j.ijrmms.2014.02.008 (fulltext)

[Tong2012] Tong, A.-T., Catalano, E., Chareyre, B. (2012), **Pore-scale flow simulations: model predictions compared with experiments on bi-dispersed granular assemblies**. *Oil & Gas Science and Technology - Rev. IFP Energies nouvelles*. DOI 10.2516/ogst/2012032 (fulltext)

[Tran2011] Tran, V.T., Donzé, F.V., Marin, P. (2011), **A discrete element model of concrete under high triaxial loading**. *Cement and Concrete Composites*.

[Tran2012] Tran, V.D.H., Meguid, M.A., Chouinard, L.E. (2012), **An algorithm for the propagation of uncertainty in soils using the discrete element method**. *The Electronic Journal of Geotechnical Engineering*. (fulltext)

[Tran2012c] Tran, V.D.H., Meguid, M.A., Chouinard, L.E. (2012), **Discrete element and experimental investigations of the earth pressure distribution on cylindrical shafts**. *International Journal of Geomechanics*. DOI 10.1061/(ASCE)GM.1943-5622.0000277

[Tran2013] Tran, V.D.H., Meguid, M.A., Chouinard, L.E. (2013), **A finite–discrete element framework for the 3d modeling of geogrid–soil interaction under pullout loading conditions**. *Geotextiles and Geomembranes* (37), pages 1-9. DOI 10.1016/j.geotexmem.2013.01.003

[Tran2014] Tran, VDH, Meguid, MA, Chouinard, LE (2014), **Three-dimensional analysis of geogrid-reinforced soil using a finite-discrete element framework**. *International Journal of Geomechanics*.

[Valera2015] Valera, Roberto Rosello, Morales, Irvin Perez, Vanmaercke, Simon, Morfa, Carlos Recarey, Cortes, Lucia Arguelles, Casanas, Harold Diaz-Guzman (2015), **Modified algorithm for generating high volume fraction sphere packings**. *Computational Particle Mechanics*, pages 1–12. DOI 10.1007/s40571-015-0045-8

[Wan2014] Wan, R, Khosravani, S, Pouragha, M (2014), **Micromechanical analysis of force transport in wet granular soils**. *Vadose Zone Journal* (13).

[Wang2014] Wang, XiaoLiang, Li, JiaChun (2014), **Simulation of triaxial response of granular materials by modified dem**. *Science China Physics, Mechanics & Astronomy* (57), pages 2297–2308.

[Zhao2015] J. Zhao, N. Guo (2015), **The interplay between anisotropy and strain localisation in granular soils: a multiscale insight**. *Géotechnique*. (under review)

[kozicki2014] Kozicki, Jan, Tejchman, Jacek, Mühlhaus, Hans-Bernd (2014), **Discrete simulations of a triaxial compression test for sand by dem**. *International Journal for Numerical and Analytical Methods in Geomechanics* (38), pages 1923–1952.

[1stYadeWorkshop] **Booklet of presentations of the 1st Yade Workshop** (2014). (fulltext)

[Albaba2015] Albaba, Adel, Lambert, Stéphane, Nicot, François, Chareyre, Bruno (2015), **Modeling the impact of granular flow against an obstacle**. In *Recent Advances in Modeling Landslides and Debris Flows*.

[Bonilla2014] Bonilla-Sierra, V, Donzé, FV, Scholtès, L, Elmouttie, M (2014), **Coupling photogrammetric data with a discrete element model for rock slope stability assessment**. In *Rock Engineering and Rock Mechanics: Structures in and on Rock Masses*.

[Bourrier2015b] Bourrier, Franck, Baroth, Julien, Lambert, Stéphane (2015), **How can reliability-based approaches improve the design of rockfall protection fences?**. In *Engineering Geology for Society and Territory-Volume 2*.

[Catalano2009] E. Catalano, B. Chareyre, E. Barthélémy (2009), **Fluid-solid coupling in discrete models**. In *Alert Geomaterials Workshop 2009*.

[Catalano2010a] E. Catalano, B. Chareyre, E. Barthélémy (2010), **A coupled model for fluid-solid interaction analysis in geomaterials**. In *Alert Geomaterials Workshop 2010*.

[Catalano2010b] E. Catalano, B. Chareyre, E. Barthélémy (2010), **Pore scale modelling of stokes flow**. In *GdR MeGe*.

[Catalano2011a] E. Catalano, B. Chareyre, A. Cortis, E. Barthélémy (2011), **A pore-scale hydro-mechanical coupled model for geomaterials**. In *II International Conference on Particle-based Methods - Fundamentals and Applications*. (fulltext)

[Catalano2013b] Catalano E., Chareyre B., Barthélémy E. (2013), **Dem-pfv analysis of solid-fluid transition in granular sediments under the action of waves**. In *Powders and Grains 2013: Proceedings of the 6th International Conference on Micromechanics of Granular Media. AIP Conference Proceedings*. DOI 10.1063/1.4812118 (fulltext)

[Chareyre2009] Chareyre B., Scholtès L. (2009), **Micro-statics and micro-kinematics of capillary phenomena in dense granular materials**. In *POWDERS AND GRAINS 2009: Proceedings of the 6th International Conference on Micromechanics of Granular Media. AIP Conference Proceedings*. DOI 10.1063/1.3180083

[Chareyre2011] B. Chareyre, E. Catalano, E. Barthélémy (2011), **Numerical simulation of hydromechanical couplings by combined discrete element method and finite-volumes**. In *International Conference on Flows and Mechanics in Natural Porous Media from Pore to Field Scale - Pore2Field*. (fulltext)

[Chareyre2012b] B. Chareyre, L. Scholtès, F. Darve (2012), **The properties of some stress tensors investigated by dem simulations**. In *Euromech Colloquium 539; Mechanics of Unsaturated Porous Media: Effective Stress principle; from micromechanics to thermodynamics* (fulltext)

[Chareyre2012c] B. Chareyre (2012), **Micro-poromechanics: recent advances in numerical models and perspectives**. In *ICACM symposium 2012, The role of structure on emerging material properties*

[Chen2008a] Chen, F., Drumm, E.C., Guiochon, G., Suzuki, K. (2008), **Discrete element simulation of 1d upward seepage flow with particle-fluid interaction using coupled open source software**. In *Proceedings of The 12th International Conference of the International Association for Computer Methods and Advances in Geomechanics (IACMAG) Goa, India*.

[Chen2009b] Chen, F., Drumm, E.C., Guiochon, G. (2009), **3d dem analysis of graded rock fill sinkhole repair: particle size effects on the probability of stability**. In *Transportation Research Board Conference (Washington DC)*.

[Dang2008a] Dang, H.K., Mohamed, M.A. (2008), **An algorithm to generate a specimen for discrete element simulations with a predefined grain size distribution.**. In *61th Canadian Geotechnical Conference, Edmonton, Alberta*.

[Dang2008b] Dang, H.K., Mohamed, M.A. (2008), **3d simulation of the trap door problem using the discrete element method.**. In *61th Canadian Geotechnical Conference, Edmonton, Alberta*.

[Effeindzourou2015] A. Effeindzourou, K. Thoeni, A. Giacomini, S.W. Sloan (2015), **A discrete model for rock impacts on muckpiles**. In *Computer Methods and Recent Advances in Geomechanics*. (fulltext)

[Elias2013] Eliáš, J. (2013), **Dem simulation of railway ballast using polyhedral elemental shapes**. In *Particle-Based Methods III: Fundamentals and Applications - Proceedings of the 3rd International Conference on Particle-based MethodsFundamentals and Applications, Particles 2013*. (fulltext)

[Favier2009b] L. Favier, D. Daudon, F. Donzé, J. Mazars (2009), **Validation of a dem granular flow model aimed at forecasting snow avalanche pressure**. In *AIP Conference Proceedings*. DOI 10.1063/1.3180002 (fulltext)

[Gillibert2009] Gillibert L., Flin F., Rolland du Roscoat S., Chareyre B., Philip A., Lesaffre B., Meyssonier J. (2009), **Curvature-driven grain segmentation: application to snow images from x-ray microtomography**. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Miami, USA)*.

[Gladky2015a] Gladky, Anton, Lieberwirth, Holger, Schwarze, Ruediger (2015), **Dem simulation of the dry and weakly wetted bulk flow on a pelletizing table**. In *13th U.S. National Congress on Computational Mechanics*.

[Gladky2015b] Gladky, Anton, Roy, Sudeshna, Weinhart, Thomas, Luding, Stefan, Schwarze, Ruediger (2015), **Dem simulations of weakly wetted granular materials: implementation of capillary bridge models**. In *Fourth Conference on Particle-Based Methods (PARTICLES 2015)*. (fulltext)

[Guo2013] Ning Guo, Jidong Zhao (2013), **A hierarchical model for cross-scale simulation of granular media**. In *Powders and Grains 2013: Proceedings of the 6th International Conference on Micromechanics of Granular Media. AIP Conference Proceedings*. DOI 10.1063/1.4812158 (fulltext)

[Guo2014b] Guo, Ning, Zhao, Jidong (2015), **A multiscale investigation of strain localization in cohesionless sand**. In *Bifurcation and Degradation of Geomaterials in the New Millennium* (Chau, Kam-Tim and Zhao, Jidong, ed.), DOI 10.1007/978-3-319-13506-9_18 (fulltext)

[Hadda2013b] Nejib Hadda, François Nicot, Luc Sibille, Farhang Radjai, Antoinette Tordesillas, Félix Darve (2013), **A multiscale description of failure in granular materials**. In *Powders and Grains 2013: Proceedings of the 6th International Conference on Micromechanics of Granular Media. AIP Conference Proceedings*. DOI 10.1063/1.4811999 (fulltext)

[Hadda2015b] Hadda, N, Bourrier, F, Sibille, L, Nicot, F, Wan, R, Darve, F (2015), **A microstructural cluster-based description of diffuse and localized failures**. In *Geomechanics from Micro to Macro: Proceedings of the International Symposium on Geomechanics from Micro and Macro (IS-Cambridge 2014)*.

[Harthong2013] Barthélémy Harthong, Richard G Wan (2009), **Directional plastic flow and fabric dependencies in granular materials**. In *Powders and Grains 2013: Proceedings of the 6th International Conference on Micromechanics of Granular Media. AIP Conference Proceedings*. DOI http://dx.doi.org/10.1063/1.4811900 (fulltext)

[Hasan2010b] A. Hasan, B. Chareyre, J. Kozicki, F. Flin, F. Darve, J. Meyssonnier (2010), **Microtomography-based discrete element modeling to simulate snow microstructure deformation**. In *AGU Fall Meeting Abstracts*

[Hicher2009] Hicher P.-Y., Scholtès L., Chareyre B., Nicot F., Darve F. (2008), **On the capillary stress tensor in wet granular materials**. In *Inaugural International Conference of the Engineering Mechanics Institute (EM08) - (Minneapolis, USA)*.

[Hicher2011] Hicher, P.Y, Scholtès, L., Sibille, L. (2011), **Multiscale modeling of particle removal impact on granular material behavior**. In *Engineering Mechanics Institute, EMI 2011*.

[Hilton2013b] J. E. Hilton, P. W. Cleary, A. Tordesillas (2013), **Unitary stick-slip motion in granular beds**. In *Powders and Grains 2013: Proceedings of the 6th International Conference on Micromechanics of Granular Media. AIP Conference Proceedings*. DOI 10.1063/1.4812063 (fulltext)

[Kozicki2003a] J. Kozicki, J. Tejchman (2003), **Discrete methods to describe the behaviour of quasi-brittle and granular materials**. In *16th Engineering Mechanics Conference, University of Washington, Seattle, CD–ROM*.

[Kozicki2003c] J. Kozicki, J. Tejchman (2003), **Lattice method to describe the behaviour of quasi-brittle materials**. In *CURE Workshop, Effective use of building materials, Sopot*.

[Kozicki2004a] J. Kozicki, J. Tejchman (2004), **Study of fracture process in concrete using a discrete lattice model**. In *CURE Workshop, Simulations in Urban Engineering, Gdansk*.

[Kozicki2005a] Kozicki, J. (2005), **Discrete lattice model used to describe the fracture process of concrete**. In *Discrete Element Group for Risk Mitigation Annual Report 1, Grenoble University of Joseph Fourier, France*. (fulltext)

[Kozicki2005b] J. Kozicki, J. Tejchman (2005), **Simulations of fracture in concrete elements using a discrete lattice model**. In *Proc. Conf. Computer Methods in Mechanics (CMM 2005), Czestochowa, Poland*.

[Kozicki2005c] J. Kozicki, J. Tejchman (2005), **Simulation of the crack propagation in concrete with a discrete lattice model**. In *Proc. Conf. Analytical Models and New Concepts in Concrete and Masonry Structures (AMCM 2005), Gliwice, Poland*.

[Kozicki2006b] J. Kozicki, J. Tejchman (2006), **Modelling of fracture process in brittle materials using a lattice model**. In *Computational Modelling of Concrete Structures, EURO-C (eds.: G. Meschke, R. de Borst, H. Mang and N. Bicanic), Taylor anf Francis*.

[Kozicki2006c] J. Kozicki, J. Tejchman (2006), **Lattice type fracture model for brittle materials**. In *35th Solid Mechanics Conference (SOLMECH 2006), Krakow*.

[Kozicki2007c] J. Kozicki, J. Tejchman (2007), **Simulations of fracture processes in concrete using a 3d lattice model**. In *Int. Conf. on Computational Fracture and Failure of Materials and Structures (CFRAC 2007), Nantes*. (fulltext)

[Kozicki2007d] J. Kozicki, J. Tejchman (2007), **Effect of aggregate density on fracture process in concrete using 2d discrete lattice model**. In *Proc. Conf. Computer Methods in Mechanics (CMM 2007), Lodz-Spala*.

[Kozicki2007e] J. Kozicki, J. Tejchman (2007), **Modelling of a direct shear test in granular bodies with a continuum and a discrete approach**. In *Proc. Conf. Computer Methods in Mechanics (CMM 2007), Lodz-Spala*.

[Kozicki2007f] J. Kozicki, J. Tejchman (2007), **Investigations of size effect in tensile fracture of concrete using a lattice model**. In *Proc. Conf. Modelling of Heterogeneous Materials with Applications in Construction and Biomedical Engineering (MHM 2007), Prague*.

[Kozicki2011] J. Kozicki, J. Tejchman (2011), **Numerical simulation of sand behaviour using dem with two different descriptions of grain roughness**. In *II International Conference on Particle-based Methods - Fundamentals and Applications*. (fulltext)

[Kozicki2013] Jan Kozicki, Jacek Tejchman, Danuta Lesniewska (2013), **Study of some micro-structural phenomena in granular shear zones**. In *Powders and Grains 2013: Proceedings of the 6th International Conference on Micromechanics of Granular Media. AIP Conference Proceedings*. DOI 10.1063/1.4811976 (fulltext)

[Li2014] Li, W, Vincens, E, Reboul, N, Chareyre, B (2014), **Constrictions and filtration of fine particles in numerical granular filters: influence of the fabric within the material**. In *Scour and Erosion: Proceedings of the 7th International Conference on Scour and Erosion, Perth, Australia, 2-4 December 2014*.

[Lomine2010a] Lominé, F., Scholtès, L., Poullain, P., Sibille, L. (2010), **Soil microstructure changes induced by internal fluid flow: investigations with coupled de/lb methods**. In *Proc. of 3rd Euromediterranean Symposium on Advances in Geomaterials and Structures, AGS'10*.

[Lomine2010b] Lominé, F., Poullain, P., Sibille, L. (2010), **Modelling of fluid-solid interaction in granular media with coupled lb/de methods: application to solid particle detachment under hydraulic loading**. In *19th Discrete Simulation of Fluid Dynamics, DSFD 2010*.

[Lomine2011] Lominé, F., Sibille, L., Marot, D. (2011), **A coupled discrete element - lattice botzmann method to investigate internal erosion in soil**. In *Proc. 2nd Int. Symposium on Computational Geomechanics (ComGeo II).*

[Maurin2013] R. Maurin, B. Chareyre, J. Chauchat, P. Frey (2013), **Discrete element modelling of bedload transport**. In *Proceedings of THESIS 2013, Two-pHase modElling for Sediment dynamIcS in Geophysical Flows.* (fulltext)

[Maurin2014a] R. Maurin, J. Chauchat, B. Chareyre, P. Frey (2014), **Dem-cfd coupling applied to bedload transport**. In *Modelling Granular Media Across Scales, Montpellier.*

[Maurin2014b] R. Maurin, J. Chauchat, B. Chareyre, P. Frey (2014), **Dem-cfd coupling applied to bedload transport**. In *Condensed Matter in Paris.*

[Maurin2014c] R. Maurin, J. Chauchat, B. Chareyre, P. Frey (2014), **Dem-fluid coupling applied to bedload transport**. In *Yade workshop, Grenoble.*

[Michallet2012] H. Michallet, E. Catalano, C. Berni, V. Rameliarison, E. Barthélémy (2012), **Physical and numerical modelling of sand liquefaction in waves interacting with a vertical wall**. In *ICSE6 - 6th International conference on Scour and Erosion*

[Modenese2012] C. Modenese, S. Utili, G.T. Houlsby (2012), **Dem modelling of elastic adhesive particles with application to lunar soil**. In *Earth and Space 2012: Engineering, Science, Construction, and Operations in Challenging Environments ¬© 2012 ASCE.* DOI 10.1061/9780784412190.006 (fulltext)

[Modenese2012a] Modenese, C, Utili, S, Houlsby, G T (2012), **A study of the influence of surface energy on the mechanical properties of lunar soil using DEM**. In *Discrete Element Modelling of Particulate Media* (Wu, Chuan-Yu, ed.), Royal Society of Chemistry , DOI 10.1039/9781849735032-00069 (fulltext)

[Modenese2012b] Modenese, C, Utili, S, Houlsby, G T (2012), **A numerical investigation of quasi-static conditions for granular media**. In *Discrete Element Modelling of Particulate Media* (Wu, Chuan-Yu, ed.), Royal Society of Chemistry , DOI 10.1039/9781849735032-00187 (fulltext)

[Nicot2010] Nicot, F., Sibille, L., Daouadji, A., Hicher, P.Y., Darve, F. (2010), **Multiscale modeling of instability in granular materials**. In *Engineering Mechanics Institute, EMI 2010.*

[Nicot2011b] Nicot, F., Hadda, N., Bourrier, F., Sibille, L., Darve, F. (2011), **A discrete element analysis of collapse mechanisms in granular materials**. In *Proc. 2nd Int. Symposium on Computational Geomechanics (ComGeo II).*

[Nicot2015] Nicot, F, Hadda, N, Bourrier, F, Sibille, L, Tordesillas, A, Darve, F (2015), **Micromechanical analysis of second order work in granular media**. In *Bifurcation and Degradation of Geomaterials in the New Millennium.*

[Nitka2014] Nitka, M, Tejchman, J (2014), **Discrete modeling of micro-structure evolution during concrete fracture using dem**. In *Computational Modelling of Concrete Structures.*

[Nitka2014b] Nitka, M, Tejchman, J (2014), **Discrete modeling of micro-structure evolution during concrete fracture using dem**. In *Computational Modelling of Concrete Structures.*

[Nitka2014c] Nitka, M, Tejchman, J, Kozicki, J, Lesniewska, D (2014), **Effect of mean grain diameter on vortices, force chains and local volume changes in granular shear zones**. In *Digital Humanitarians: How Big Data Is Changing the Face of Humanitarian Response.*

[Nitka2015b] Nitka, Michal, Tejchman, Jacek, Kozicki, Jan (2015), **Discrete modelling of micro-structural phenomena in granular shear zones**. In *Bifurcation and Degradation of Geomaterials in the New Millennium.*

[Sari2011] H. Sari, B. Chareyre, E. Catalano, P. Philippe, E. Vincens (2011), **Investigation of internal erosion processes using a coupled dem-fluid method**. In *II International Conference on Particle-based Methods - Fundamentals and Applications.* (fulltext)

[Sayeed2014] Sayeed, Md Abu, Sazzad, Md Mahmud, Suzuki, Kiichi (2014), **Mechanical behavior of granular materials considering confining pressure dependency**. In *GeoCongress 2012.*

[Scholtes2007a] L. Scholtès, B. Chareyre, F. Nicot, F. Darve (2007), **Micromechanical modelling of unsaturated granular media**. In *Proceedings ECCOMAS-MHM07, Prague.*

[Scholtes2008a] L. Scholtès, B. Chareyre, F. Nicot, F. Darve (2008), **Capillary effects modelling in unsaturated granular materials**. In *8th World Congress on Computational Mechanics - 5th European Congress on Computational Methods in Applied Sciences and Engineering, Venice*.

[Scholtes2008b] L. Scholtès, P.-Y. Hicher, F.Nicot, B. Chareyre, F. Darve (2008), **On the capillary stress tensor in unsaturated granular materials**. In *EM08: Inaugural International Conference of the Engineering Mechanics Institute, Minneapolis*.

[Scholtes2009e] Scholtes L, Chareyre B, Darve F (2009), **Micromechanics of partialy saturated granular material**. In *Int. Conf. on Particle Based Methods, ECCOMAS-Particles*.

[Scholtes2011b] L. Scholtès, F. Donzé (2011), **Progressive failure mechanisms in jointed rock: insight from 3d dem modelling**. In *II International Conference on Particle-based Methods - Fundamentals and Applications*. (fulltext)

[Scholtes2011c] Scholtès, L., Hicher, P.Y., Sibille, L. (2011), **A micromechanical approach to describe internal erosion effects in soils**. In *Proc. of Geomechanics and Geotechnics: from micro to macro, IS-Shanghai 2011*.

[Shiu2007a] W. Shiu, F.V. Donze, L. Daudeville (2007), **Discrete element modelling of missile impacts on a reinforced concrete target**. In *Int. Conf. on Computational Fracture and Failure of Materials and Structures (CFRAC 2007), Nantes*.

[Sibille2009] Sibille, L., Scholtès, L. (2009), **Effects of internal erosion on mechanical behaviour of soils: a dem approach**. In *Proc. of International Conference on Particle-Based Methods, Particles 2009*.

[Skarzynski2014] Skarzynski, M Nitka, Tejchman, J (2014), **Two-scale model for concrete beams subjected to three point bending—numerical analyses and experiments**. In *Computational Modelling of Concrete Structures*.

[Smilauer2007a] V. Šmilauer (2007), **Discrete and hybrid models: applications to concrete damage**. In *Unpublished*. (fulltext)

[Smilauer2008] Václav Šmilauer (2008), **Commanding c++ with python**. In *ALERT Doctoral school talk*. (fulltext)

[Smilauer2010a] Václav Šmilauer (2010), **Yade: past, present, future**. In *Internal seminary in Laboratoire 3S-R, Grenoble*. (fulltext) (LaTeX sources)

[Stransky2010] Jan Stránský, Milan Jirásek, Václav Šmilauer (2010), **Macroscopic elastic properties of particle models**. In *Proceedings of the International Conference on Modelling and Simulation 2010, Prague*. (fulltext)

[Stransky2011] J. Stransky, M. Jirasek (2011), **Calibration of particle-based models using cells with periodic boundary conditions**. In *II International Conference on Particle-based Methods - Fundamentals and Applications*. (fulltext)

[Sweijen2014] Thomas Sweijen, Majid Hassanizadeh, Bruno Chareyre (2014), **Pore-scale modeling of swelling porous media; application to super absorbent polymers**. In *XX . International Conference on Computational Methods in Water Resources*. (fulltext)

[Tejchman2011] Tejchman, J. (2011), **Comparative modelling of shear zone patterns in granular bodies with finite and discrete element model**. *Advances in Bifurcation and Degradation in Geomaterials*, pages 255–260.

[Thoeni2011] K. Thoeni, C. Lambert, A. Giacomini, S.W. Sloan (2011), **Discrete modelling of a rockfall protective system**. In *Particles 2011: Fundamentals and Applications*. (fulltext)

[Thoeni2015] K. Thoeni, A. Giacomini, C. Lambert, S.W. Sloan (2015), **Rockfall Trajectory Analysis with Drapery Systems**. In *Engineering Geology for Society and Territory - Volume 2* (G. Lollino and D. Giordan and G.B. Crosta and J. Corominas and R. Azzam and J. Wasowski and N. Sciarra, ed.), Springer , DOI 10.1007/978-3-319-09057-3_356 (fulltext)

[Toraldo2015] Toraldo, Marcella, Chareyre, Bruno, Sibille, Luc (2015), **Numerical modelling of the localized fluidization in a saturated granular medium using the coupled method dem-pfv**. In *Annual Report 1* (Grenoble Geomechanics Group, ed.), (fulltext)

[Tran2012b] Tran, V.D.H, Meguid, M.A, Chouinard, L.E (2012), **A discrete element study of the earth pressure distribution on cylindrical shafts**. In *Tunnelling Association of Canada (TAC) Conference 2012, Montreal*

[Uhlmann2014] Uhlmann, Eckart, Dethlefs, Arne, Eulitz, Alexander (2014), **Investigation of material removal and surface topography formation in vibratory finishing**. In *Procedia CIRP*.

[Wan2015] Wan, Richard, Hadda, Nejib (2015), **Plastic deformations in granular materials with rotation of principal stress axes**. In *Bifurcation and Degradation of Geomaterials in the New Millennium*.

[Yuan2014] Chao Yuan, Bruno Chareyre, Félix Darve (2014), **Pore-scale simulations of drainage for two-phase flow in dense sphere packings**. In *XX . International Conference on Computational Methods in Water Resources*. (fulltext)

[Borrmann2014] Sebastian, Borrmann (2014), **Dem-cfd simulation: erprobung neuer kopplungsansätze in ausgewählten softwarepaketen (in german)**. Master thesis at *Institute of Mechanics and Fluid Dynamics, TU Bergakademie Freiberg*.

[Catalano2008a] E. Catalano (2008), **Infiltration effects on a partially saturated slope - an application of the discrete element method and its implementation in the open-source software yade**. Master thesis at *UJF-Grenoble*. (fulltext)

[Catalano2012] Emanuele Catalano (2012), **A pore-scale coupled hydromechanical model for biphasic granular media**. PhD thesis at *Université de Grenoble*. (fulltext)

[Charlas2013] Benoit Charlas (2013), **Etude du comportement mécanique d'un hydrure intermétallique utilisé pour le stockage d'hydrogène**. PhD thesis at *Université de Grenoble*. (fulltext)

[Chen2009a] Chen, F. (2009), **Coupled flow discrete element method application in granular porous media using open source codes**. PhD thesis at *University of Tennessee, Knoxville*. (fulltext)

[Chen2011b] Chen, J. (2011), **Discrete element method (dem) analyses for hot-mix asphalt (hma) mixture compaction**. PhD thesis at *University of Tennessee, Knoxville*. (fulltext)

[Duriez2009a] J. Duriez (2009), **Stabilité des massifs rocheux : une approche mécanique**. PhD thesis at *Institut polytechnique de Grenoble*. (fulltext)

[Favier2009c] Favier, L. (2009), **Approche numérique par éléments discrets 3d de la sollicitation d'un écoulement granulaire sur un obstacle**. PhD thesis at *Université Grenoble I – Joseph Fourier*.

[GaeblerMedack2013] Gäbler, Nils, Medack, Jörg (2013), **Experimental and simulative study of particle dynamics on a chute**. Project work at *Institute of Mechanics and Fluid Dynamics, TU Bergakademie Freiberg*.

[GentzAverhausVilbusch2014] Gentz, Julia, Averhaus, Jan, Vilbusch, Stephan (2014), **Numerical simulation of particle movement on a pelletizing disc – set-up and validation of a dem model**. Project work at *Institute of Mechanics and Fluid Dynamics, TU Bergakademie Freiberg*.

[Guo2014c] N. Guo (2014), **Multiscale characterization of the shear behavior of granular media**. PhD thesis at *The Hong Kong University of Science and Technology*.

[Jerier2009b] Jerier, J.F. (2009), **Modélisation de la compression haute densité des poudres métalliques ductiles par la méthode des éléments discrets (in french)**. PhD thesis at *Université Grenoble I – Joseph Fourier*. (fulltext)

[Kozicki2007b] J. Kozicki (2007), **Application of discrete models to describe the fracture process in brittle materials**. PhD thesis at *Gdansk University of Technology*. (fulltext)

[Marzougui2011] Marzougui, D. (2011), **Hydromechanical modeling of the transport and deformation in bed load sediment with discrete elements and finite volume**. Master thesis at *Ecole Nationale d'Ingénieur de Tunis*. (fulltext)

[Medack2014] Medack, Jörg (2014), **Untersuchungen zur beeinflussung der örtlichen aufgabegutverteilung auf der schurre eines hammerbrechers (in german)**. Master thesis at *Institute for Mineral Processing Machines, TU Bergakademie Freiberg*.

[Scholtes2009d] Luc Scholtès (2009), **modélisation micromécanique des milieux granulaires partiellement saturés**. PhD thesis at *Institut National Polytechnique de Grenoble*. (fulltext)

[Smilauer2010b] Václav Šmilauer (2010), **Cohesive particle model using the discrete element method on the yade platform**. PhD thesis at *Czech Technical University in Prague, Faculty of Civil Engineering & Université Grenoble I – Joseph Fourier, École doctorale I-MEP2*. (fulltext) (LaTeX sources)

[Smilauer2010c] Václav Šmilauer (2010), **Doctoral thesis statement**. *(PhD thesis summary)*. (fulltext) (LaTeX sources)

[Tran2011b] Van Tieng TRAN (2011), **Structures en béton soumises à des chargements mécaniques extrêmes: modélisation de la réponse locale par la méthode des éléments discrets (in french)**. PhD thesis at *Université Grenoble I – Joseph Fourier*. (fulltext)

[Addetta2001] G.A. D'Addetta, F. Kun, E. Ramm, H.J. Herrmann (2001), **From solids to granulates - Discrete element simulations of fracture and fragmentation processes in geomaterials.**. In *Continuous and Discontinuous Modelling of Cohesive-Frictional Materials*. (fulltext)

[Allen1989] M. P. Allen, D. J. Tildesley (1989), **Computer simulation of liquids**. Clarendon Press.

[Alonso2004] F. Alonso-Marroquin, R. Garcia-Rojo, H.J. Herrmann (2004), **Micro-mechanical investigation of the granular ratcheting**. In *Cyclic Behaviour of Soils and Liquefaction Phenomena*. (fulltext)

[Antypov2011] D. Antypov, J. A. Elliott (2011), **On an analytical solution for the damped hertzian spring**. *EPL (Europhysics Letters)* (94), pages 50004. (fulltext)

[Bagi2006] Katalin Bagi (2006), **Analysis of microstructural strain tensors for granular assemblies**. *International Journal of Solids and Structures* (43), pages 3166 - 3184. DOI 10.1016/j.ijsolstr.2005.07.016

[Bertrand2005] D. Bertrand, F. Nicot, P. Gotteland, S. Lambert (2005), **Modelling a geo-composite cell using discrete analysis**. *Computers and Geotechnics* (32), pages 564–577.

[Bertrand2008] D. Bertrand, F. Nicot, P. Gotteland, S. Lambert (2008), **Discrete element method (dem) numerical modeling of double-twisted hexagonal mesh**. *Canadian Geotechnical Journal* (45), pages 1104–1117.

[Calvetti1997] Calvetti, F., Combe, G., Lanier, J. (1997), **Experimental micromechanical analysis of a 2d granular material: relation between structure evolution and loading path**. *Mechanics of Cohesive-frictional Materials* (2), pages 121–163.

[Camborde2000a] F. Camborde, C. Mariotti, F.V. Donzé (2000), **Numerical study of rock and concrete behaviour by discrete element modelling**. *Computers and Geotechnics* (27), pages 225–247.

[Chan2011] D. Chan, E. Klaseboer, R. Manica (2011), **Film drainage and coalescence between deformable drops and bubbles.**. *Soft Matter* (7), pages 2235-2264.

[Chareyre2002a] B. Chareyre, L. Briancon, P. Villard (2002), **Theoretical versus experimental modeling of the anchorage capacity of geotextiles in trenches.**. *Geosynthet. Int.* (9), pages 97–123.

[Chareyre2002b] B. Chareyre, P. Villard (2002), **Discrete element modeling of curved geosynthetic anchorages with known macro-properties.**. In *Proc., First Int. PFC Symposium, Gelsenkirchen, Germany*.

[Chareyre2003] Bruno Chareyre (2003), **Modélisation du comportement d'ouvrages composites sol-géosynthétique par éléments discrets - application aux tranchées d'ancrage en tête de talus.**. PhD thesis at *Grenoble University*. (fulltext)

[Chareyre2005] Bruno Chareyre, Pascal Villard (2005), **Dynamic spar elements and discrete element methods in two dimensions for the modeling of soil-inclusion problems**. *Journal of Engineering Mechanics* (131), pages 689–698. DOI 10.1061/(ASCE)0733-9399(2005)131:7(689) (fulltext)

[CundallStrack1979] P.A. Cundall, O.D.L. Strack (1979), **A discrete numerical model for granular assemblies**. *Geotechnique* (), pages 47–65. DOI 10.1680/geot.1979.29.1.47

[Dallavalle1948] J. M. DallaValle (1948), **Micrometrics : the technology of fine particles**. Pitman Pub. Corp.

[DeghmReport2006] F. V. Donzé (ed.), **Annual report 2006** (2006). *Discrete Element Group for Hazard Mitigation*. Université Joseph Fourier, Grenoble (fulltext)

[Donze1994a] F.V. Donzé, P. Mora, S.A. Magnier (1994), **Numerical simulation of faults and shear zones**. *Geophys. J. Int.* (116), pages 46–52.

[Donze1995a] F.V. Donzé, S.A. Magnier (1995), **Formulation of a three-dimensional numerical model of brittle behavior**. *Geophys. J. Int.* (122), pages 790–802.

[Donze1999a] F.V. Donzé, S.A. Magnier, L. Daudeville, C. Mariotti, L. Davenne (1999), **Study of the behavior of concrete at high strain rate compressions by a discrete element method**. *ASCE J. of Eng. Mech* (125), pages 1154–1163. DOI 10.1016/S0266-352X(00)00013-6

[Donze2004a] F.V. Donzé, P. Bernasconi (2004), **Simulation of the blasting patterns in shaft sinking using a discrete element method**. *Electronic Journal of Geotechnical Engineering* (9), pages 1–44.

[GarciaRojo2004] R. García-Rojo, S. McNamara, H. J. Herrmann (2004), **Discrete element methods for the micro-mechanical investigation of granular ratcheting**. In *Proceedings ECCOMAS 2004*. (fulltext)

[Hentz2003] Séebastien Hentz (2003), **Modélisation d'une structure en béton armé soumise à un choc par la méthode des eléments discrets**. PhD thesis at *Université Grenoble 1 – Joseph Fourier*.

[Hentz2004a] S. Hentz, F.V. Donzé, L.Daudeville (2004), **Discrete element modelling of concrete submitted to dynamic loading at high strain rates**. *Computers and Structures* (82), pages 2509–2524. DOI 10.1016/j.compstruc.2004.05.016

[Hentz2004b] S. Hentz, L. Daudeville, F.V. Donzé (2004), **Identification and validation of a discrete element model for concrete**. *ASCE Journal of Engineering Mechanics* (130), pages 709–719. DOI 10.1061/(ASCE)0733-9399(2004)130:6(709)

[Hentz2005a] S. Hentz, F.V. Donzé, L.Daudeville (2005), **Discrete elements modeling of a reinforced concrete structure submitted to a rock impact**. *Italian Geotechnical Journal* (XXXIX), pages 83–94.

[Herminghaus2005] Herminghaus, S. (2005), **Dynamics of wet granular matter**. *Advances in Physics* (54), pages 221-261. DOI 10.1080/00018730500167855 (fulltext)

[Hubbard1996] Philip M. Hubbard (1996), **Approximating polyhedra with spheres for time-critical collision detection**. *ACM Trans. Graph.* (15), pages 179–210. DOI 10.1145/231731.231732

[Ivars2011] Diego Mas Ivars, Matthew E. Pierce, Caroline Darcel, Juan Reyes-Montes, David O. Potyondy, R. Paul Young, Peter A. Cundall (2011), **The synthetic rock mass approach for jointed rock mass modelling**. *International Journal of Rock Mechanics and Mining Sciences* (48), pages 219 - 244. DOI 10.1016/j.ijrmms.2010.11.014

[Johnson2008] Scott M. Johnson, John R. Williams, Benjamin K. Cook (2008), **Quaternion-based rigid body rotation integration algorithms for use in particle methods**. *International Journal for Numerical Methods in Engineering* (74), pages 1303–1313. DOI 10.1002/nme.2210

[Jung1997] Derek Jung, Kamal K. Gupta (1997), **Octree-based hierarchical distance maps for collision detection**. *Journal of Robotic Systems* (14), pages 789–806. DOI 10.1002/(SICI)1097-4563(199711)14:11<789::AID-ROB3>3.0.CO;2-Q

[Kettner2011] Lutz Kettner, Andreas Meyer, Afra Zomorodian (2011), **Intersecting sequences of dD iso-oriented boxes**. In *CGAL User and Reference Manual*. (fulltext)

[Klosowski1998] James T. Klosowski, Martin Held, Joseph S. B. Mitchell, Henry Sowizral, Karel Zikan (1998), **Efficient collision detection using bounding volume hierarchies of k-dops**. *IEEE Transactions on Visualization and Computer Graphics* (4), pages 21–36. (fulltext)

[Kuhl2001] E. Kuhl, G. A. D'Addetta, M. Leukart, E. Ramm (2001), **Microplane modelling and particle modelling of cohesive-frictional materials**. In *Continuous and Discontinuous Modelling of Cohesive-Frictional Materials*. DOI 10.1007/3-540-44424-6_3 (fulltext)

[Lambert2008] Lambert, Pierre, Chau, Alexandre, Delchambre, Alain, Régnier, Stéphane (2008), **Comparison between two capillary forces models**. *Langmuir* (24), pages 3157–3163.

[Lu1998] Ya Yan Lu (1998), **Computing the logarithm of a symmetric positive definite matrix**. *Appl. Numer. Math* (26), pages 483–496. DOI 10.1016/S0168-9274(97)00103-7 (fulltext)

[Lucy1977] Lucy, L.~B. (1977), **A numerical approach to the testing of the fission hypothesis**. *aj* (82), pages 1013-1024. DOI 10.1086/112164 (fulltext)

[Luding2008] Stefan Luding (2008), **Introduction to discrete element methods**. In *European Journal of Environmental and Civil Engineering*.

[Luding2008b] Luding, Stefan (2008), **Cohesive, frictional powders: contact models for tension**. *Granular Matter* (10), pages 235-246. DOI 10.1007/s10035-008-0099-x (fulltext)

[Magnier1998a] S.A. Magnier, F.V. Donzé (1998), **Numerical simulation of impacts using a discrete element method**. *Mech. Cohes.-frict. Mater.* (3), pages 257–276. DOI 10.1002/(SICI)1099-1484(199807)3:3<257::AID-CFM50>3.0.CO;2-Z

[Mani2013] Mani, Roman, Kadau, Dirk, Herrmann, HansJ. (2013), **Liquid migration in sheared unsaturated granular media**. *Granular Matter* (15), pages 447-454. DOI 10.1007/s10035-012-0387-3 (fulltext)

[McNamara2008] S. McNamara, R. García-Rojo, H. J. Herrmann (2008), **Microscopic origin of granular ratcheting**. *Physical Review E* (77). DOI 11.1103/PhysRevE.77.031304

[Monaghan1985] Monaghan, J.~J., Lattanzio, J.~C. (1985), **A refined particle method for astrophysical problems**. *aap* (149), pages 135-143. (fulltext)

[Monaghan1992] Monaghan, J.~J. (1992), **Smoothed particle hydrodynamics**. *araa* (30), pages 543-574. DOI 10.1146/annurev.aa.30.090192.002551

[Morris1997] (1997), **Modeling low reynolds number incompressible flows using {sph}**. *Journal of Computational Physics* (136), pages 214 - 226. DOI http://dx.doi.org/10.1006/jcph.1997.5776 (fulltext) ()

[Mueller2003] Müller, Matthias, Charypar, David, Gross, Markus (2003), **Particle-based fluid simulation for interactive applications**. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. (fulltext)

[Munjiza1998] A. Munjiza, K. R. F. Andrews (1998), **Nbs contact detection algorithm for bodies of similar size**. *International Journal for Numerical Methods in Engineering* (43), pages 131–149. DOI 10.1002/(SICI)1097-0207(19980915)43:1<131::AID-NME447>3.0.CO;2-S

[Munjiza2006] A. Munjiza, E. Rougier, N. W. M. John (2006), **Mr linear contact detection algorithm**. *International Journal for Numerical Methods in Engineering* (66), pages 46–71. DOI 10.1002/nme.1538

[Neto2006] Natale Neto, Luca Bellucci (2006), **A new algorithm for rigid body molecular dynamics**. *Chemical Physics* (328), pages 259–268. DOI 10.1016/j.chemphys.2006.07.009

[Omelyan1999] Igor P. Omelyan (1999), **A new leapfrog integrator of rotational motion. the revised angular-momentum approach**. *Molecular Simulation* (22). DOI 10.1080/08927029908022097 (fulltext)

[Pfc3dManual30] ICG (2003), **Pfc3d (particle flow code in 3d) theory and background manual, version 3.0**. Itasca Consulting Group.

[Pion2011] Sylvain Pion, Monique Teillaud (2011), **3D triangulations**. In *CGAL User and Reference Manual*. (fulltext)

[Potyondy2004] D.O. Potyondy, P.A. Cundall (2004), **A bonded-particle model for rock**. *International Journal of Rock Mechanics and Mining Sciences* (41), pages 1329 - 1364. DOI 10.1016/j.ijrmms.2004.09.011

[Pournin2001] L. Pournin, Th. M. Liebling, A. Mocellin (2001), **Molecular-dynamics force models for better control of energy dissipation in numerical simulations of dense granular media**. *Phys. Rev. E* (65), pages 011302. DOI 10.1103/PhysRevE.65.011302

[Price2007] Mathew Price, Vasile Murariu, Garry Morrison (2007), **Sphere clump generation and trajectory comparison for real particles**. In *Proceedings of Discrete Element Modelling 2007*. (fulltext)

[Rabinov2005] RABINOVICH Yakov I., ESAYANUR Madhavan S., MOUDGIL Brij M. (2005), **Capillary forces between two spheres with a fixed volume liquid bridge : theory and experiment**. *Langmuir* (21), pages 10992–10997. (fulltext) (eng)

[Radjai2011] Radjai, F., Dubois, F. (2011), **Discrete-element modeling of granular materials**. John Wiley & Sons. (fulltext)

[RevilBaudard2013] Revil-Baudard, T., Chauchat, J. (2013), **A two-phase model for sheet flow regime based on dense granular flow rheology**. *Journal of Geophysical Research: Oceans* (118), pages 619–634.

[Richarson1954] Richardson, J. F.,, W. N. Zaki (1954), **Sedimentation and fluidization: part i**. *Trans. Instn. Chem. Engrs* (32).

[Satake1982] M. Satake (1982), **Fabric tensor in granular materials.**. In *Proc., IUTAM Symp. on Deformation and Failure of Granular materials, Delft, The Netherlands*.

[Schmeeckle2007] Schmeeckle, Mark W., Nelson, Jonathan M., Shreve, Ronald L. (2007), **Forces on stationary particles in near-bed turbulent flows**. *Journal of Geophysical Research: Earth Surface* (112). DOI 10.1029/2006JF000536 (fulltext)

[Schwager2007] Schwager, Thomas, Pöschel, Thorsten (2007), **Coefficient of restitution and linear–dashpot model revisited**. *Granular Matter* (9), pages 465-469. DOI 10.1007/s10035-007-0065-z (fulltext)

[Soulie2006] Soulié, F., Cherblanc, F., El Youssoufi, M.S., Saix, C. (2006), **Influence of liquid bridges on the mechanical behaviour of polydisperse granular materials**. *International Journal for Numerical and Analytical Methods in Geomechanics* (30), pages 213–228. DOI 10.1002/nag.476 (fulltext)

[Thornton1991] Colin Thornton, K. K. Yin (1991), **Impact of elastic spheres with and without adhesion**. *Powder technology* (65), pages 153–166. DOI 10.1016/0032-5910(91)80178-L

[Thornton2000] Colin Thornton (2000), **Numerical simulations of deviatoric shear deformation of granular media**. *Géotechnique* (50), pages 43–53. DOI 10.1680/geot.2000.50.1.43

[Verlet1967] Loup Verlet (1967), **Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules**. *Phys. Rev.* (159), pages 98. DOI 10.1103/PhysRev.159.98

[Villard2004a] P. Villard, B. Chareyre (2004), **Design methods for geosynthetic anchor trenches on the basis of true scale experiments and discrete element modelling**. *Canadian Geotechnical Journal* (41), pages 1193–1205.

[Wang2009] Yucang Wang (2009), **A new algorithm to model the dynamics of 3-d bonded rigid bodies with rotations**. *Acta Geotechnica* (4), pages 117–127. DOI 10.1007/s11440-008-0072-1 (fulltext)

[Weigert1999] Weigert, Tom, Ripperger, Siegfried (1999), **Calculation of the liquid bridge volume and bulk saturation from the half-filling angle**. *Particle & Particle Systems Characterization* (16), pages 238–242. DOI 10.1002/(SICI)1521-4117(199910)16:5<238::AID-PPSC238>3.0.CO;2-E (fulltext)

[Wiberg1985] Wiberg, Patricia L., Smith, J. Dungan (1985), **A theoretical model for saltating grains in water**. *Journal of Geophysical Research: Oceans* (90), pages 7341–7354.

[Willett2000] Willett, Christopher D., Adams, Michael J., Johnson, Simon A., Seville, Jonathan P. K. (2000), **Capillary bridges between two spherical bodies**. *Langmuir* (16), pages 9396-9405. DOI 10.1021/la000657y (fulltext)

[Zhou1999536] Y.C. Zhou, B.D. Wright, R.Y. Yang, B.H. Xu, A.B. Yu (1999), **Rolling friction in the dynamic simulation of sandpile formation**. *Physica A: Statistical Mechanics and its Applications* (269), pages 536–553. DOI 10.1016/S0378-4371(99)00183-1 (fulltext)

[cgal] Jean-Daniel Boissonnat, Olivier Devillers, Sylvain Pion, Monique Teillaud, Mariette Yvinec (2002), **Triangulations in cgal**. *Computational Geometry: Theory and Applications* (22), pages 5–19.

Python Module Index