Yade Documentation

Release 0.60.3

Václav Šmilauer

July 28, 2011

Contents

Note: Please consult changes to Yade documention with documentation manager (whoever that is), even if you have commit permissions.

See older tentative contents

Chapter 1

Installation

Yade can be installed from packages (precompiled 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 source code.

1.1 Packages

Packages are (as of now) provided for several Ubuntu versions from Yade package archive. Different version of Yade can be installed alongisde each other. The `yade` vitual package always depends on the latest stable package, while `yade-snapshot` will pull the latest snapshot package. To install quickly, run the following:

```
sudo add-apt-repository ppa:yade-users/ppa
sudo add-apt-repository ppa:yade-users/external   # optional (updates of other packages)
sudo apt-get update
sudo apt-get install yade
```

More detailed instructions are available at the archive page

1.2 Source code

1.2.1 Download

If you want to install from source, you can install either a release (numbered version, which is frozen) or the current developement 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 Launchpad. We use Bazaar (the `bzr` command) for code management (install the `bzr` package in your distribution):

   ```
   bzr checkout lp:yade
   ```

   will download the whole code repository of `trunk`. Check out Quick Bazaar tutorial wiki page for more. For those behind firewall, daily snapshot of the repository (as compressed archive) is provided.

Release and trunk sources are compiled in the same way.

1.2.2 Prerequisities

Yade relies on a number of external software to run; its installation is checked before the compilation starts.

- scons build system
- gcc compiler (g++); other compilers will not work; you need g++>=4.2 for openMP support
- boost 1.35 or later
- qt3 library
- freeglut3
- libQGLViewer
- python, numpy, ipython
- matplotlib
- eigen2 algebra library
- gdb debugger
- sqlite3 database engine
- Loki library
- VTK library (optional but recommended)

Most of the list above is very likely already packaged for your distribution. In Ubuntu, it can be all installed by the following command (cut&paste to the terminal):

```
sudo apt-get install scons freeglut3-dev libloki-dev \
libboost-date-time-dev libboost-filesystem-dev libboost-thread-dev \
libboost-regex-dev fakeroot dpkg-dev build-essential g++ \
libboost-iostreams-dev liblog4cxx10-dev python-dev libboost-python-dev ipython \
python-matplotlib libsqlite3-dev python-numeric python-tk gnuplot doxygen \
libgts-dev python-pygraphviz libvtk5-dev python-scientific bzr libeigen2-dev \
binutils-gold python-xlib python-qt4 pyqt4-dev-tools \
libqglviewer-qt4-dev
```

command line (cut&paste to the terminal under root privileges) for Fedora (not good tested yet!)::

> yum install scons qt3-devel freeglut-devel boost-devel boost-date-time boost-filesystem boost-thread boost-regex fakeroot gcc gcc-c++ boost-iostreams log4cxx log4cxx-devel python-devel boost-python ipython python-matplotlib sqlite-devel python-numeric ScientificPython-tk gnuplot doxygen gts-devel graphviz-python vtk-devel ScientificPython bzr eigen2-devel libQGLViewer-devel loki-lib-devel python-xlib PyQt4 PyQt4-devel

1.2.3 Compilation

Inside the directory where you downloaded the sources (ex “yade” if you use bazaar), install Yade to your home directory (without root priviledges):

```
scons PREFIX=/home/username/YADE
```

If you have a machine that you are the only user on, you can instead change permission on `/usr/local` and install subsequently without specifying the `PREFIX`:

```
sudo chown user: /usr/local    # replace "user" with your login name
scons
```

There is a number of options for compilation you can change; run `scons -h` to see them (see also *scons-parameters* in the *Programmer’s manual*)

The compilation process can take a long time, be patient.

Decreasing RAM usage furing compilation

Yade demands a large amount of memory for compilation (due to extensive template use). If you have less than 2GB of RAM, it will be, you might encounter difficulties such as the computer being apparently stalled, compilation taking very long time (hours) or erroring out. This command will minimize RAM usage, but the compilation will take longer – only one file will be compiled simultaneously and files will be "chunked" together one by one:

```
scons jobs=1 chunkSize=1
```

Chapter 2

Introduction

2.1 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.

2.1.1 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 bzr1984
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

- version (`bzr1984`); always state this version you use if you seek help in the community or report bug;
- some information for *Remote control*, which you are unlikely to need now;
- basic help for the command-line that just appeared (`Yade [1]:`).

[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.

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

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 script.py
Welcome to Yade bzr1986
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.

2.1.2 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 [4]: TriaxialTest(numberOfGrains=200).load()

Yade [5]: len(O.bodies)
 ->  [5]: 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*.

2.1.3 Running simulation

As explained above, 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 [7]: O.iter
 ->  [7]: 0

Yade [8]: O.time
 ->  [8]: 0.0
```

[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.

```
Yade [9]: O.dt=1e-4

Yade [10]: O.step()

Yade [11]: O.iter
 ->  [11]: 1

Yade [12]: O.time
 ->  [12]: 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 [13]: O.run()

Yade [14]: O.pause()

Yade [15]: O.iter
 ->  [15]: 1

Yade [16]: O.run(100000,True)

Yade [17]: O.iter
 ->  [17]: 100001

Yade [18]: O.stopAtIter=500000

Yade [19]: O.wait()

Yade [20]: O.iter
 ->  [20]: 100001
```

2.1.4 Saving and loading

Simulation can be saved at any point to (optionally compressed) XML file. With some limitations, it is generally possible to load the XML later and resume the simulation as if it were not interrupted. Note that since XML is merely readable dump of Yade's internal objects, it might not (probably will not) open with different Yade version.

```
Yade [21]: O.save('/tmp/a.xml.bz2')

Yade [22]: O.reload()

Yade [24]: O.load('/tmp/another.xml.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:

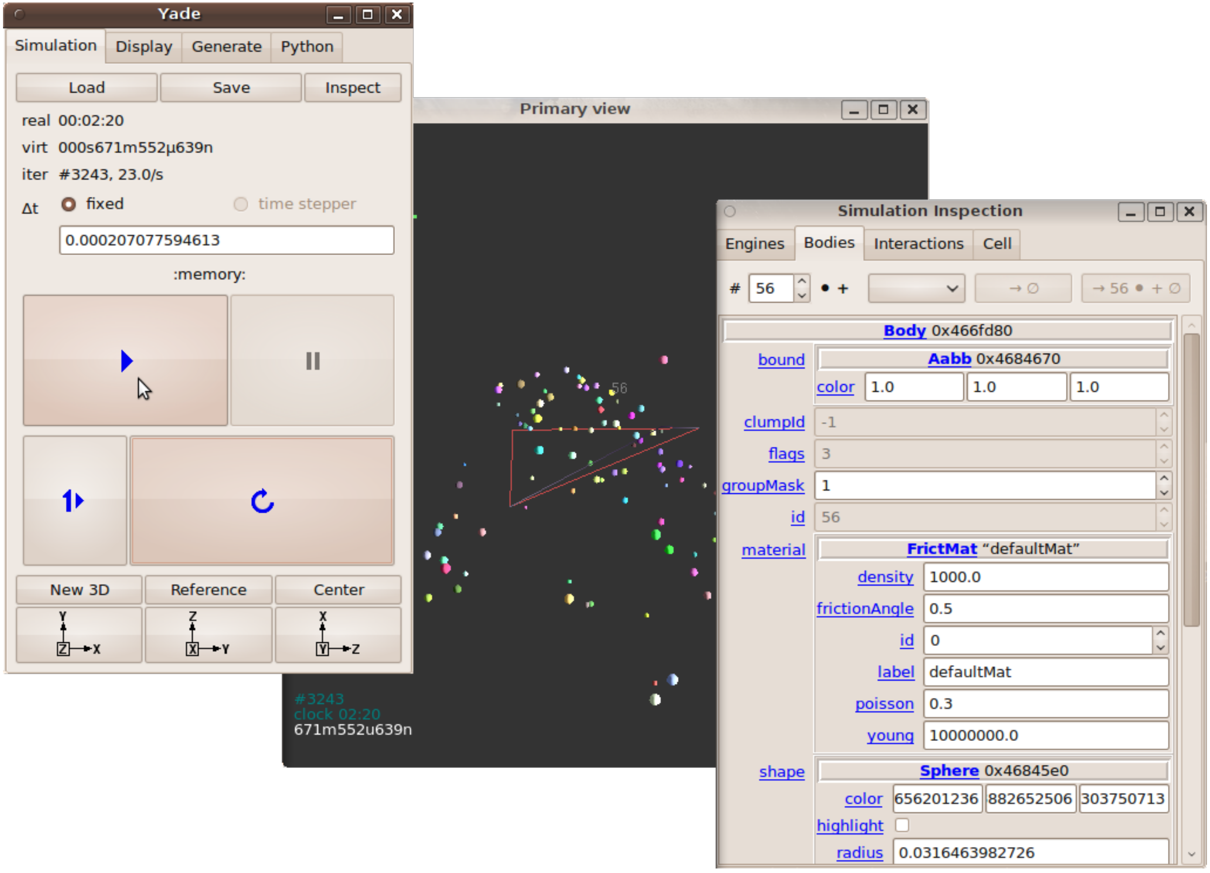
```
Yade [25]: O.saveTmp()

Yade [26]: O.loadTmp()

Yade [27]: O.saveTmp('init') ## named memory slot

Yade [28]: 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.

2.1.5 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.



real 00:02:20
virt 000s671m552µ639n
iter #3243, 23.0/s
Δt ● fixed ○ time stepper
0.000207077594613
:memory:

#3243
clock 02:20
671m552u639n

56
Body 0x466fd80
bound Aabb 0x4684670
color 1.0 1.0 1.0
clumpId -1
flags 3
groupMask 1
id 56
material FrictMat "defaultMat"
density 1000.0
frictionAngle 0.5
id 0
label defaultMat
poisson 0.3
young 10000000.0
shape Sphere 0x46845e0
color 656201236 882652506 303750713
highlight □
radius 0.0316463982726

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.

2.2 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.

2.2.1 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 [32]: O.bodies       # some data components
 ->  [32]: <yade.wrapper.BodyContainer object at 0xd0828ec>

Yade [33]: len(O.bodies)  # there are no bodies as of yet
 ->  [33]: 0

Yade [34]: O.engines      # functional components, empty at the moment
 ->  [34]: []
```

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 [35]: s=utils.sphere(center=[0,0,0],radius=1)

Yade [36]: s.shape, s.state, s.mat, s.bound
 ->  [36]:
(<Sphere instance at 0xb2c6080>,
 <State instance at 0xcad14b0>,
 <FrictMat instance at 0x9f31868>,
 None)

Yade [37]: s.state.pos
 ->  [37]: Vector3(0,0,0)

Yade [38]: s.shape.radius
 ->  [38]: 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 [39]: O.bodies.append(s)
 ->  [39]: 0

Yade [40]: O.bodies.append(utils.sphere([0,0,2],.5))
 ->  [40]: 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 [41]: O.bodies[1].state.pos
 ->  [41]: Vector3(0,0,2)

Yade [42]: O.bodies[100]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

/build/buildd/yade-0.60.3/doc/sphinx/<ipython console> in <module>()

IndexError: Body id out of range.
```

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

```
Yade [43]: O.bodies.append(s)
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

/build/buildd/yade-0.60.3/doc/sphinx/<ipython console> in <module>()

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 [44]: 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, Facet+Sphere etc.

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 [48]: O.interactions[0,1]
 ->  [48]: <Interaction instance at 0xc5806b0>

Yade [49]: O.interactions[1,0]      # order of ids is not important
 ->  [49]: <Interaction instance at 0xc5806b0>

Yade [50]: i=O.interactions[0,1]

Yade [51]: i.id1,i.id2
 ->  [51]: (0, 1)

Yade [52]: i.geom
 ->  [52]: <Dem3DofGeom_SphereSphere instance at 0xc6ebe30>

Yade [53]: i.phys
 ->  [53]: <FrictPhys instance at 0xbf55050>

Yade [54]: O.interactions[100,10111]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

/build/buildd/yade-0.60.3/doc/sphinx/<ipython console> in <module>()

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 [55]: O.forces.f(0)
 ->  [55]: Vector3(0,0,0)

Yade [56]: O.forces.addF(0,Vector3(1,2,3))

Yade [57]: O.forces.f(0)
 ->  [57]: 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.

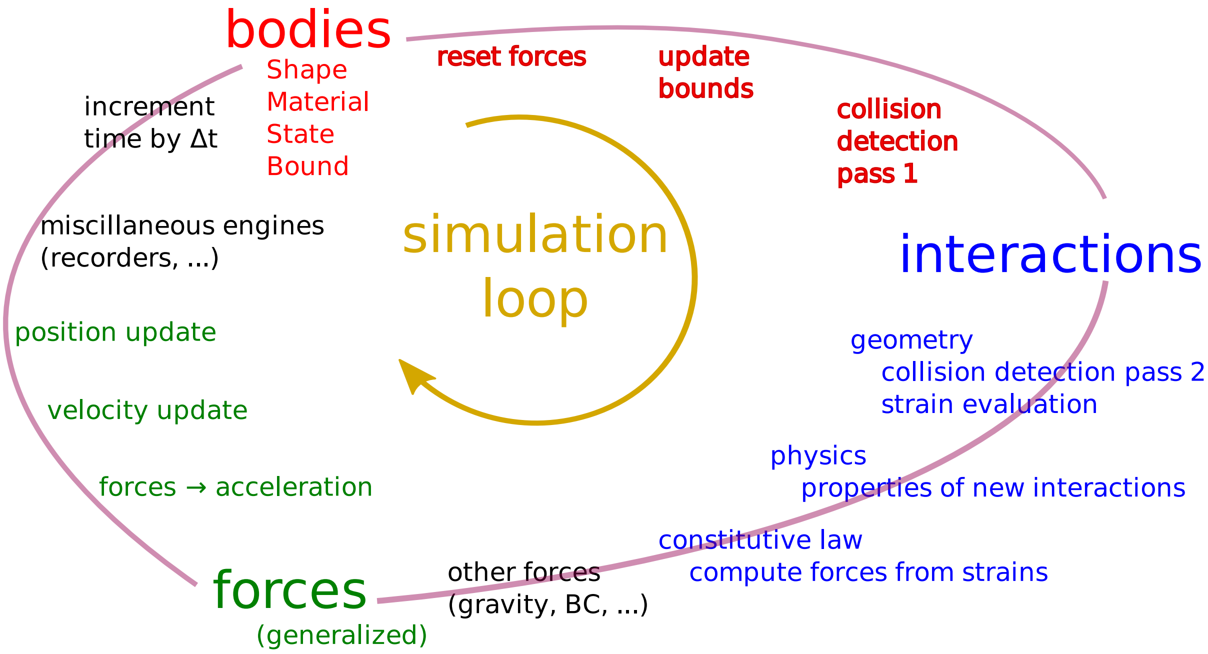


Figure 2.1: Typical simulation loop; each step begins at body-cented bit at 11 o'clock, continues with interaction bit, force application bit, miscillanea and ends with time update.

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_Dem3DofGeom(),Ig2_Facet_Sphere_Dem3DofGeom()],
                [Ip2_FrictMat_FrictMat_FrictPhys()],
                [Law2_Dem3Dof_Elastic_Elastic()],
        ),
        # 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_Dem3DofGeom(),Ig2_Facet_Sphere_Dem3DofGeom()],
        [Ip2_FrictMat_FrictMat_FrictPhys()],
        [Law2_Dem3Dof_Elastic_Elastic()],
),
```

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_Dem3DofGeom, is called on interaction of 2 Spheres and creates Dem3DofGeom instance, if appropriate.
2. The second functor, Ig2_Facet_Sphere_Dem3DofGeom, is called for interaction of Facet with Sphere and might create (again) a Dem3DofGeom 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.

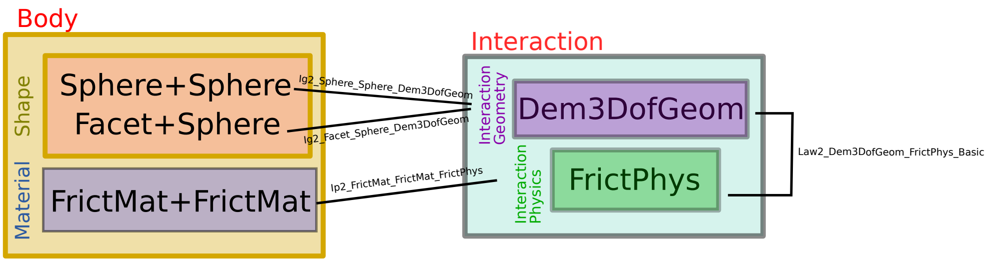


Figure 2.2: Chain of functors producing and accepting certain types. In the case shown, the `Ig2` functors produce Dem3DofGeom instances from all handled Shape combinations; the `Ig2` functor produces FrictMat. The constitutive law functor `Law2` accepts the combination of types produced. Note that the types are stated in the functor's class names.

Chapter 3

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.

3.1 Collision detection

3.1.1 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

$$\forall \mathbf{x} \in R^3 : x \in P_i \Rightarrow x \in \tilde{P}_i \tag{3.1}$$

   (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*,

$$\left(\tilde{P}_i \cap \tilde{P}_j\right) = \emptyset \Rightarrow \left(P_i \cap P_j\right) = \emptyset \tag{3.2}$$

   which is a candidate exclusion rule in the proper sense.

2. By filtering away impossible collisions in (3.2), 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.

3.1.2 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. *sect-formulation-dt*) 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 (3.1) 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.

3.1.3 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):

$$\left(\tilde{P}_i \cap \tilde{P}_j\right) \neq \emptyset \Leftrightarrow \bigwedge_{w \in \{x,y,z\}} \left[\left(\left(\tilde{P}_i^{w0}, \tilde{P}_i^{w1}\right) \cap \left(\tilde{P}_j^{w0}, \tilde{P}_j^{w1}\right)\right) \neq \emptyset\right]$$

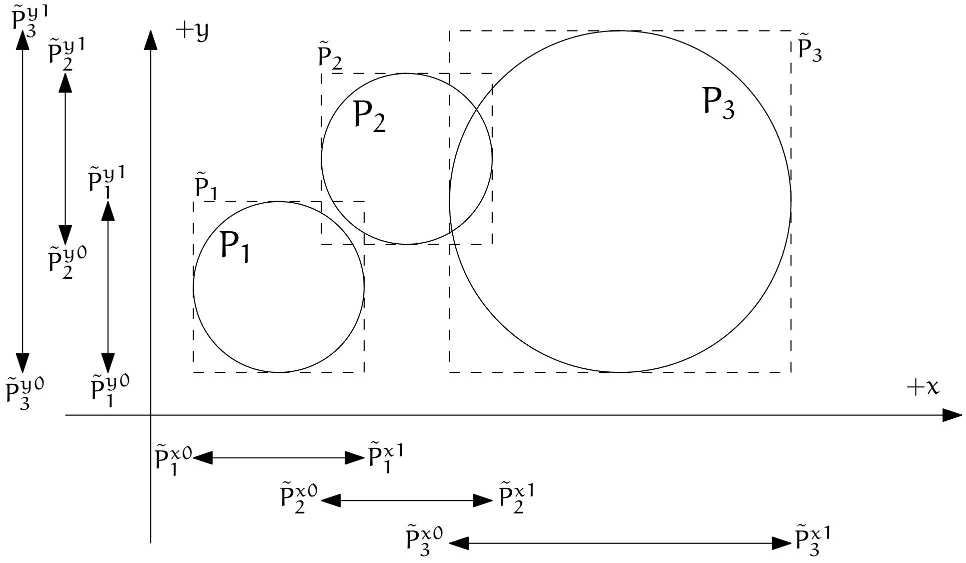


Figure 3.1: Sweep and prune algorithm (shown in 2D), where Aabb of each sphere is represented by minimum and maximum value along each axis. Spatial overlap of Aabb's is present if they overlap along all axes. In this case, $\tilde{P}_1 \cap \tilde{P}_2 \neq \emptyset$ (but note that $P_1 \cap P_2 = \emptyset$) and $\tilde{P}_2 \cap \tilde{P}_3 \neq \emptyset$.}

where (a, b) denotes interval in $\mathbb{R}$.

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:

(≤)	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 $\boxed{i}$)

$$\| \quad \boxed{3} \qquad 7 \qquad 2 \qquad 4 \quad \|.$$

It obviously immediately satisfies (‖), and we move to the next element:

$$\| \quad 3 \underset{\leq}{\xleftarrow{\quad}} \boxed{7} \qquad 2 \qquad 4 \quad \|.$$

Condition (≤) holds, therefore we move to the right. The $\boxed{2}$ is not in order (violating (≤)) and two inversions take place; after that, (‖) holds:

$$\begin{aligned}
&\| \quad 3 \qquad 7 \underset{\nleq}{\xleftarrow{\quad}} \boxed{2} \qquad 4 \quad \|,\\
&\| \quad 3 \underset{\nleq}{\xleftarrow{\quad}} \boxed{2} \qquad 7 \qquad 4 \quad \|,\\
&\| \quad \boxed{2} \qquad 3 \qquad 7 \qquad 4 \quad \|.
\end{aligned}$$

The last element $\boxed{4}$ first violates (≤), but satisfies it after one inversion

$$\begin{aligned}
&\| \quad 2 \qquad 3 \qquad 7 \underset{\nleq}{\xleftarrow{\quad}} \boxed{4} \quad \|,\\
&\| \quad 2 \qquad 3 \underset{\leq}{\xleftarrow{\quad}} \boxed{4} \qquad 7 \quad \|.
\end{aligned}$$

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 $\mathcal{O}(n)$ 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 *sect-periodic-insertion-sort*, 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 $\tilde{P}_i$.

In Yade, this is achieved by enlarging Aabb of particles by fixed relative length in all dimensions ΔL (InsertionSortCollider.sweepLength). Suppose the collider run last time at step m and the current step is

n. NewtonIntegrator tracks maximum distance traversed by particles (via maximum velocity magnitudes $v^{\circ}_{\max} = \max |\dot{u}^{\circ}_{i}|$ in each step, with the initial cummulative distance $L_{\max} = 0$,

$$L^{\circ}_{\max} = L^{-}_{\max} + v^{\circ}_{\max}\Delta t^{\circ} \tag{3.3}$$

triggering the collider re-run as soon as

$$L^{\circ}_{\max} > \Delta L. \tag{3.4}$$

The disadvantage of this approach is that even one fast particle determines $v^{\circ}_{\max}$.

A solution is to track maxima per particle groups. The possibility of tracking each particle separately (that is what ESyS-Particle does) seemed to us too fine-grained. Instead, we assign particles to b_n (InsertionSortCollider.nBins) *velocity bins* based on their current velocity magnitude. The bins' limit values are geometrical with the coefficient $b_c > 1$ (InsertionSortCollider.binCoeff), the maximum velocity being the current global velocity maximum $v^{\circ}_{\max}$ (with some constraints on its change rate, to avoid large oscillations); for bin $i \in \{0, \ldots, b_n\}$ and particle j:

$$v^{\circ}_{\max} b_c^{-(i+1)} \leq |\dot{u}^{\circ}_{j}| < v_{\max} b_c^{-i}.$$

(note that in this case, superscripts of b_c mean exponentiation). Equations (3.3)–(3.4) are used for each bin separately; however, when (3.4) is satisfied, full collider re-run is necessary and all bins' distances are reset.

Particles in high-speed oscillatory motion could be put into a slow bin if they happen to be at the point where their instantaneous speed is low, causing the necessity of early collider re-run. This is avoided by allowing particles to only go slower by one bin rather than several at once.

Results of using Verlet distance depend highly on the nature of simulation and choice of parameters InsertionSortCollider.nBins and InsertionSortColldier.binCoeff. The binning algorithm was specifically designed for simulating local fracture of larger concrete specimen; in that way, only particles in the fracturing zone, with greater velocities, had the Aabb's enlarged, without affecting quasi-still particles outside of this zone. In such cases, up to 50% overall computation time savings were observed, collider being run every 100 steps in average.

3.2 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.

3.2.1 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).

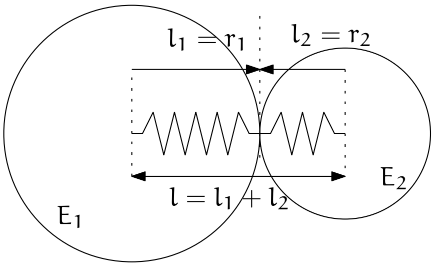


Figure 3.2: Series of 2 springs representing normal stiffness of contact between 2 spheres.

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 spehre 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.

3.2.2 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.

3.3 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.

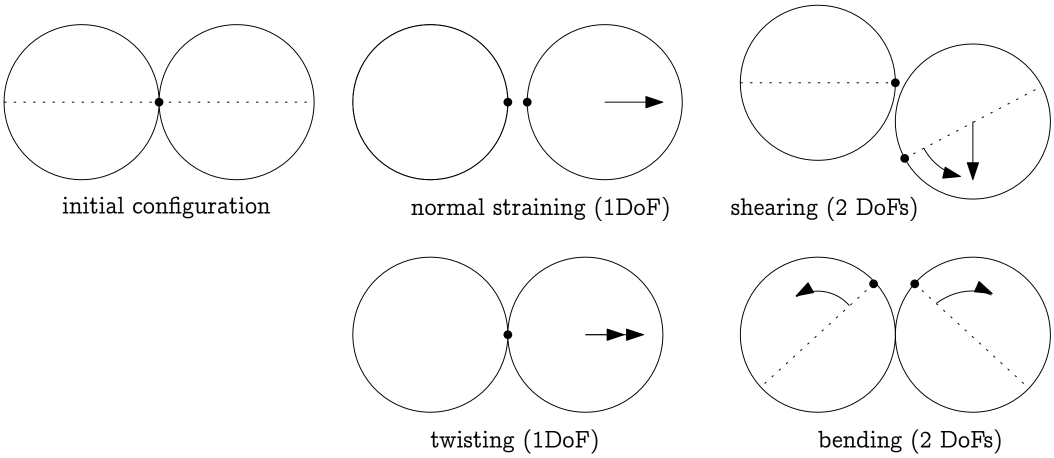


Figure 3.3: Degrees of freedom of configuration of two spheres. Normal strain appears if there is a difference of linear velocity along the interaction axis (n); shearing originates from the difference of linear velocities perpendicular to n *and* from the part of $\boldsymbol{\omega}_1 + \boldsymbol{\omega}_2$ perpendicular to n; twisting is caused by the part of $\boldsymbol{\omega}_1 - \boldsymbol{\omega}_2$ parallel with n; bending comes from the part of $\boldsymbol{\omega}_1 - \boldsymbol{\omega}_2$ perpendicular to n.

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.

3.3.1 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

$$d_0 = |\bar{\mathbf{C}}_2 - \bar{\mathbf{C}}_1|$$
$$d_1 = r_1 + \frac{d_0 - r_1 - r_2}{2}, \qquad d_2 = d_0 - d_1.$$

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*.

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.

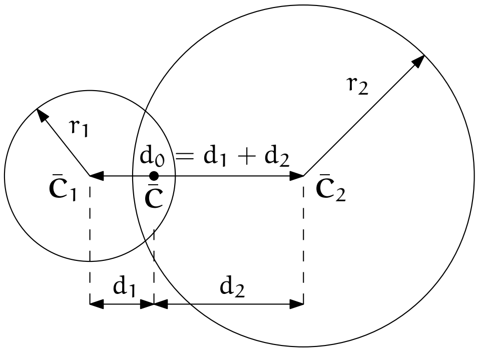


Figure 3.4: Geometry of the initial contact of 2 spheres; this case pictures spheres which already overlap when the contact is created (which can be the case at the beginning of a simulation) for the sake of generality. The initial contact point $\bar{\mathbf{C}}$ is in the middle of the overlap zone.

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

$$d_0 \leq R_I(r_1 + r_2). \tag{3.5}$$

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_Dem3DofGeom_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

$$A_{eq} = \pi \min(r_1, r_2)^2 \tag{3.6}$$

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):

$$\mathbf{n}^\circ = \frac{\mathbf{C}_2^\circ - \mathbf{C}_1^\circ}{|\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|} \equiv \widehat{\mathbf{C}_2^\circ - \mathbf{C}_1^\circ} \tag{3.7}$$

and

$$\mathbf{C}^\circ = \mathbf{C}_1^\circ + \left(d_1 - \frac{d_0 - |\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|}{2}\right)\mathbf{n}. \tag{3.8}$$

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).

3.3.2 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.

These two possibilities lead to two algorithms of computing shear strains. They should give the same results (disregarding numerical imprecision), but there is a trade-off between computational cost of the incremental method and robustness of the total one.

Geometrical meaning of shear strain is shown in fig-shear-2d.

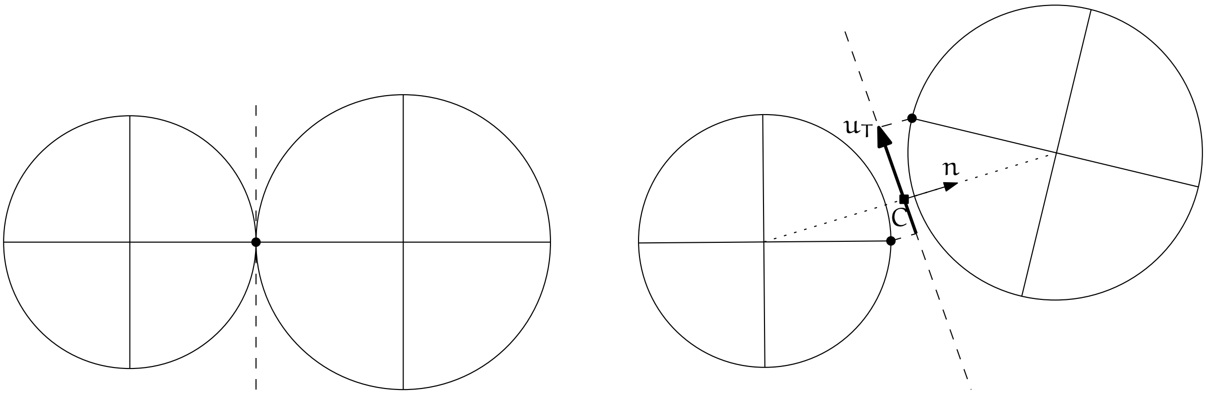


Figure 3.5: Evolution of shear displacement $\mathbf{u}_T$ due to mutual motion of spheres, both linear and rotational. Left configuration is the initial contact, right configuration is after displacement and rotation of one particle.

Incremental algorithm

The 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 (3.8) and (3.7). $\mathbf{u}_T^-$ 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

$$\mathbf{u}_T^\circ = \mathbf{u}_T^- + (\Delta\mathbf{u}_T)_1 + (\Delta\mathbf{u}_T)_2 + (\Delta\mathbf{u}_T)_3.$$

Total alogithm

The following algorithm, aiming at stabilization of response even with large rotation speeds or Δt approaching stability limit, was designed by the author of this thesis. (A similar algorithm based on total formulation, which covers additionally bending and torsion, was proposed in [Wang2009].) It is based on tracking original contact points (with zero shear) in the particle-local frame.

In this section, variable symbols implicitly denote their current values unless explicitly stated otherwise.

Shear strain may have two sources: mutual rotation of spheres or transversal displacement of one sphere with respect to the other. Shear strain does not change if both spheres move or rotate but are not in linear or angular motion mutually. To accurately and reliably model this situation, for every new contact the initial contact point $\bar{\mathbf{C}}$ is mapped into local sphere coordinates ($\mathbf{p}_{01}$, $\mathbf{p}_{02}$). As we want to determine the distance between both points (i.e. how long the trajectory in on both spheres' surfaces together), the shortest path from current $\mathbf{C}$ to the initial locally mapped point on the sphere's surface is „unrolled" to the contact plane ($\mathbf{p}'_{01}$, $\mathbf{p}'_{02}$); then we can measure their linear distance $\mathbf{u}_T$ and define shear strain $\boldsymbol{\varepsilon}_T = \mathbf{u}_T/d_0$ (fig. fig-shear-displacement).

More formally, taking $\bar{\mathbf{C}}_i$, $\bar{q}_i$ for the sphere initial positions and orientations (as quaterions) in global coordinates, the initial sphere-local contact point *orientation* (relative to sphere-local axis $\hat{x}$) is remembered:

$$\bar{\mathbf{n}} = \widehat{\mathbf{C}_1 - \mathbf{C}_2},$$
$$\bar{q}_{01} = \text{Align}(\hat{x}, \bar{q}_1^* \bar{\mathbf{n}} \bar{q}_1^{**}),$$
$$\bar{q}_{02} = \text{Align}(\hat{x}, \bar{q}_2^* (-\bar{\mathbf{n}}) \bar{q}_2^{**}).$$

After some spheres motion, the original point can be "unrolled" to the current contact plane:

$$q = \text{Align}(\mathbf{n}, q_1 \bar{q}_{01} \hat{x} (q_1 \bar{q}_{01})^*) \quad \text{(auxiliary)}$$
$$\mathbf{p}'_{01} = q_\vartheta d_1 (q_{\mathbf{u}} \times \mathbf{n})$$

where $q_{\mathbf{u}}$, q_ϑ are axis and angle components of q and p'_{01} is the unrolled point. Similarly,

$$q = \text{Align}(\mathbf{n}, q_2 \bar{q}_{02} \hat{x} (q_2 \bar{q}_{02})^*)$$
$$\mathbf{p}'_{02} = q_\vartheta d_1 (q_{\mathbf{u}} \times (-\mathbf{n})).$$

Shear displacement and strain are then computed easily:

$$\mathbf{u}_T = \mathbf{p}'_{02} - \mathbf{p}'_{01}$$
$$\varepsilon_T = \frac{\mathbf{u}_T}{d_0}$$

When using material law with plasticity in shear, it may be necessary to limit maximum shear strain, in which case the mapped points are moved closer together to the requested distance (without changing $\hat{\mathbf{u}}_T$). This allows us to remember the previous strain direction and also avoids summation of increments of plastic strain at every step (fig-shear-slip).

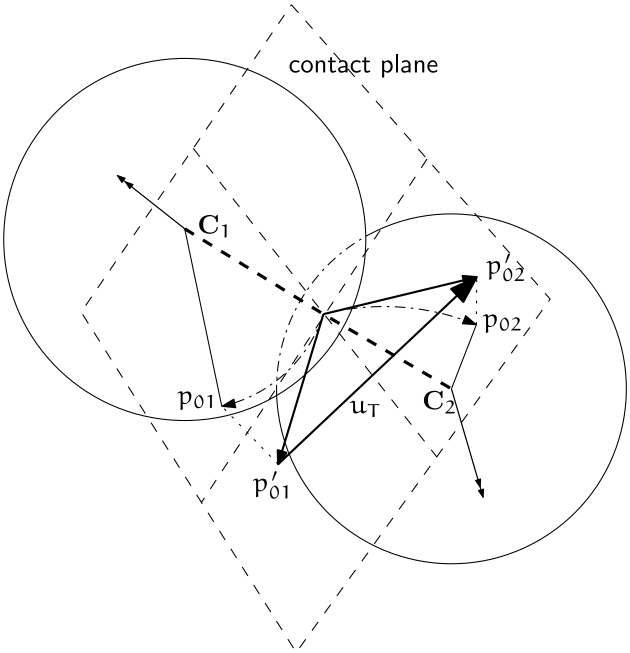


Figure 3.6: Shear displacement computation for two spheres in relative motion.

This algorithm is straightforwardly modified to facet-sphere interactions. In Yade, it is implemented by Dem3DofGeom and related classes.

3.4 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_Dem3DofGeom_FrictPhys_Basic (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

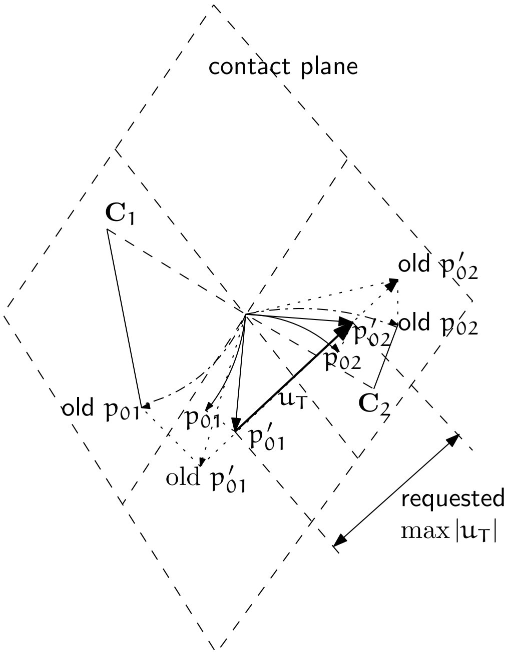


Figure 3.7: Shear plastic slip for two spheres.

$u_N > 0$, the contact is deleted without generating any forces):

$$\begin{aligned} \mathbf{F}_N &= K_N u_N \boldsymbol{n}, \\ \mathbf{F}_T^t &= K_T \boldsymbol{u}_T \end{aligned}$$

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

$$\mathbf{F}_T = \begin{cases} \mathbf{F}_T^t \frac{|\mathbf{F}_N| \tan \varphi}{\mathbf{F}_T^t} & \text{if } |\mathbf{F}_T| > |\mathbf{F}_N| \tan \varphi, \\ \mathbf{F}_T^t & \text{otherwise.} \end{cases}$$

S ummary 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 sphes' centers, torque generated by $\mathbf{F}$ must also be considered.

$$\begin{aligned} \mathbf{F}_1 +&= \mathbf{F} & \mathbf{F}_2 +&= -\mathbf{F} \\ \mathbf{T}_1 +&= d_1(-\boldsymbol{n}) \times \mathbf{F} & \mathbf{T}_2 +&= d_2 \boldsymbol{n} \times \mathbf{F}. \end{aligned}$$

3.5 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.

3.5.1 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 F acting on the particle in question and its mass m, is simply

$$\ddot{\mathbf{u}}^\circ = \mathsf{F}/m.$$

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 $\mathbf{u}^+$:

$$\begin{aligned}\dot{\mathbf{u}}^\oplus &= \dot{\mathbf{u}}^\ominus + \ddot{\mathbf{u}}^\circ \Delta t \\ \mathbf{u}^+ &= \mathbf{u}^\circ + \dot{\mathbf{u}}^\oplus \Delta t.\end{aligned}$$

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.

3.5.2 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 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

$$\dot{\omega}_i^\circ = \mathsf{T}_i/\mathbf{I}_{11},$$

We use the same approximation scheme, obtaining an equation analogous to (??)

$$\omega^{\oplus} = \omega^{\ominus} + \Delta t \dot{\omega}^{\circ}.$$

The quaternion Δq representing rotation vector $\omega^{\oplus}\Delta t$ is constructed, i.e. such that

$$(\Delta q)_{\vartheta} = |\omega^{\oplus}|,$$
$$(\Delta q)_{u} = \widehat{\omega^{\oplus}}$$

Finally, we compute the next orientation q^{+} by rotation composition

$$q^{+} = \Delta q q^{\circ}.$$

3.5.3 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):

$$\mathbf{T}_i = \mathbf{I}_{ii}\dot{\omega}_i + (\mathbf{I}_{kk} - \mathbf{I}_{jj})\omega_j\omega_k.$$

Due to the presence of the current values of both ω and $\dot{\omega}$, 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 $\mathbf{T} = \mathbf{I}\dot{\omega}$).

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 $\mathbf{A}$. Therefore, every vector a is transformed as $\widetilde{a} = qaq^{*} = \mathbf{A}a$. Since $\mathbf{A}$ is a rotation (orthogonal) matrix, the inverse rotation $\mathbf{A}^{-1} = \mathbf{A}^{T}$.

For given particle in question, we know

- $\widetilde{\mathbf{I}}^{\circ}$ (constant) inertia matrix; diagonal, since in local, principal coordinates,
- $\mathbf{T}^{\circ}$ external torque,
- q° current orientation (and its equivalent rotation matrix $\mathbf{A}$),
- $\omega^{\ominus}$ mid-step angular velocity,
- $\mathbf{L}^{\ominus}$ 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 $\mathbf{L}^{\oplus}$, q^{+}, $\omega^{\oplus}$. We first estimate current angular momentum and compute current local angular velocity:

$$\mathbf{L}^{\circ} = \mathbf{L}^{\ominus} + \mathbf{T}^{\circ}\frac{\Delta t}{2}, \qquad \widetilde{\mathbf{L}}^{\circ} = \mathbf{A}\mathbf{L}^{\circ},$$
$$\mathbf{L}^{\oplus} = \mathbf{L}^{\ominus} + \mathbf{T}^{\circ}\Delta t, \qquad \widetilde{\mathbf{L}}^{\oplus} = \mathbf{A}\mathbf{L}^{\oplus},$$
$$\widetilde{\omega}^{\circ} = \widetilde{\mathbf{I}}^{\circ\,-1}\widetilde{\mathbf{L}}^{\circ},$$
$$\widetilde{\omega}^{\oplus} = \widetilde{\mathbf{I}}^{\circ\,-1}\widetilde{\mathbf{L}}^{\oplus}.$$

Then we compute $\dot{q}^{\circ}$, using q° and $\tilde{\omega}^{\circ}$:

$$\begin{pmatrix} \dot{q}^{\circ}_w \\ \dot{q}^{\circ}_x \\ \dot{q}^{\circ}_y \\ \dot{q}^{\circ}_z \end{pmatrix} = \frac{1}{2} \begin{pmatrix} q^{\circ}_w & -q^{\circ}_x & -q^{\circ}_y & -q^{\circ}_z \\ q^{\circ}_x & q^{\circ}_w & -q^{\circ}_z & q^{\circ}_y \\ q^{\circ}_y & q^{\circ}_z & q^{\circ}_w & -q^{\circ}_x \\ q^{\circ}_z & -q^{\circ}_y & q^{\circ}_x & q^{\circ}_w \end{pmatrix} \begin{pmatrix} 0 \\ \tilde{\omega}^{\circ}_x \\ \tilde{\omega}^{\circ}_y \\ \tilde{\omega}^{\circ}_z \end{pmatrix},$$

$$q^{\oplus} = q^{\circ} + \dot{q}^{\circ}\frac{\Delta t}{2}.$$

We evaluate $\dot{q}^{\oplus}$ from $q^{\oplus}$ and $\tilde{\omega}^{\oplus}$ in the same way as in (**??**) but shifted by $\Delta t/2$ ahead. Then we can finally compute the desired values

$$q^{+} = q^{\circ} + \dot{q}^{\oplus}\Delta t,$$
$$\omega^{\oplus} = A^{-1}\tilde{\omega}^{\oplus}$$

3.5.4 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: the clump's mass m_c is summed over members, the inertia tensor $\mathbf{I}_c$ with respect to the clump's centroid is computed using the parallel axes theorem; 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

$$\mathbf{F}_c = \sum F_i/m_c,$$
$$\mathbf{T}_c = \sum r_i \times F_i + T_i.$$

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.

3.5.5 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, *sect-formulation-stress-cundall*) 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):

$$\frac{(\Delta F)_{dw}}{\mathbf{F}_w} = -\lambda_d \operatorname{sgn}(\mathbf{F}_w \dot{\mathbf{u}}^{\ominus}_w), \quad w \in \{x, y, z\}$$

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

$$\frac{(\Delta F)_{dw}}{F_w} = -\lambda_d \operatorname{sgn} F_w \underbrace{\left(\dot{u}_w^{\ominus} + \frac{\ddot{u}_w^{\circ}\Delta t}{2}\right)}_{\simeq \dot{u}_w^{\circ}}, \tag{3.9}$$

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 (3.9) is implemented in the NewtonIntegrator engine; the damping coefficient λ_d is NewtonIntegrator.damping.

3.5.6 Stability considerations

Critical timestep

In order to ensure stability for the explicit integration sceheme, an upper limit is imposed on Δt:

$$\Delta t_{cr} = \frac{2}{\omega_{max}} \tag{3.10}$$

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

$$\omega^{(1)} = \sqrt{\frac{K}{m}} \tag{3.11}$$

does not depend on initial conditions. Since there is one single mass, $\omega_{max}^{(1)} = \omega^{(1)}$. Plugging (3.11) into (3.10), we obtain

$$\Delta t_{cr}^{(1)} = 2/\omega_{max}^{(1)} = 2\sqrt{m/K}$$

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 $K_i^{(2)} = 2K_i$, giving maximum angular frequency of the whole system

$$\omega_{\max} = \max_i \sqrt{K_i^{(2)}/m_i}.$$

The overall critical timestep is then

$$\Delta t_{cr} = \frac{2}{\omega_{\max}} = \min_i 2\sqrt{\frac{m_i}{K_i^{(2)}}} = \min_i 2\sqrt{\frac{m_i}{2K_i}} = \min_i \sqrt{2}\sqrt{\frac{m_i}{K_i}}. \tag{3.12}$$

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 $K.M^{-1}$. The critical timestep is then associated to the eigen mode with highest frequency :

$$\Delta t_{cr} = \min \Delta t_{cr\,k}, \quad k \in \{1, ..., 6\}. \tag{3.13}$$

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 $K_{Tk} = \xi K_{Nk}$ 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

$$\mathbf{K}_{ij} = \sum_k (K_{Nk} - K_{Tk})\mathbf{n}_i\mathbf{n}_j + K_{Tk} = \sum_j K_{Nk}\left((1-\xi)\mathbf{n}_i\mathbf{n}_j + \xi\right) \tag{3.14}$$

with i and $j \in \{x, y, z\}$. Equations (3.13) and (3.14) 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 developed 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 (3.14) 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 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

$$\Delta t_{\mathrm{cr}}^{(c)} = l_{\min}\sqrt{\frac{\rho}{E}}.$$

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

$$\Delta t_{\mathrm{cr}}^{(p)} = \min_i R_i\sqrt{\frac{\rho_i}{E_i}}.$$

This algorithm is implemented in the utils.PWaveTimeStep function.

Let us compare this result to (3.12); 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

  $$K_N = E\pi'R', \tag{3.15}$$

  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 (3.14). It is a sum over all interactions where K_N and ξ are scalars that will not rotate with interaction, while n_w is w-th component of unit interaction normal $\boldsymbol{n}$. Since we supposed uniform spatial distribution, we can replace n_w^2 by its average value $\overline{n}_w^2$. Recognizing components of $\boldsymbol{n}$ as direction cosines, the average values of 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 $K = \boldsymbol{K}_w$ 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

$$K = NK_N\frac{1-2\xi}{3},$$

we substitute K into (3.12) using (3.15):

$$\Delta t_{cr} = \sqrt{2}\sqrt{\frac{m}{K}} = \sqrt{2}\sqrt{\frac{\frac{4}{3}\pi R^3\rho}{NE\pi' R\frac{1-2\xi}{3}}} = \underbrace{R\sqrt{\frac{\rho}{E}}}_{\Delta t_{cr}^{(p)}}2\sqrt{\frac{\pi/\pi'}{N(1-2\xi)}}.$$

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 (3.6),

$$A_{eq} = \pi R^2 K_N \qquad\qquad = \frac{A_{eq}E}{d_0}.$$

d_0 is the initial contact length, which will be, for interaction radius (3.5) $R_I > 1$, in average larger than $2R$. For $R_I = 1.5$ (sect.~ref{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 (3.15).

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.~ref{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 ref{sect-formulation-stress-cundall}. 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.

3.6 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 cell is determined by

- item the reference size $\mathbf{s}$ (Cell.refSize), giving reference cell configuration (which is always perpendicular): axis-aligned cuboid with corners $(0, 0, 0)$ and $\mathbf{s}$;
- item the transformation matrix $\mathbf{T}$ (Cell.trsf).

The transformation matrix $\mathbf{T}$ can hold arbitrary linear transformation composed of scaling, rotation and shear. Volume change of the cell is given by $\det \mathbf{T}$. The cell can be manipulated by directly changing its transformation matrix $\mathbf{T}$ and its reference size $\mathbf{s}$.

Additionally, we define teransformation gradient $\nabla \boldsymbol{v}$ (Cell.velGrad) which can be automatically integrated at every step using the Euler scheme

$$\mathbf{T}^{+} = \mathbf{T}^{\circ} + \nabla \boldsymbol{v} \Delta t.$$

Along with the automatic integration of cell transformation, there is an option to homothetically displace all particles so that $\nabla \boldsymbol{v}$ is swept linearly over the whole simulation (enabled via NewtonIntegrator.homotheticCellResize). This avoids all boundary effects coming from change of the transformation.

3.6.1 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.~ref{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.~ref{sect-periodic-insertion-sort}), 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.

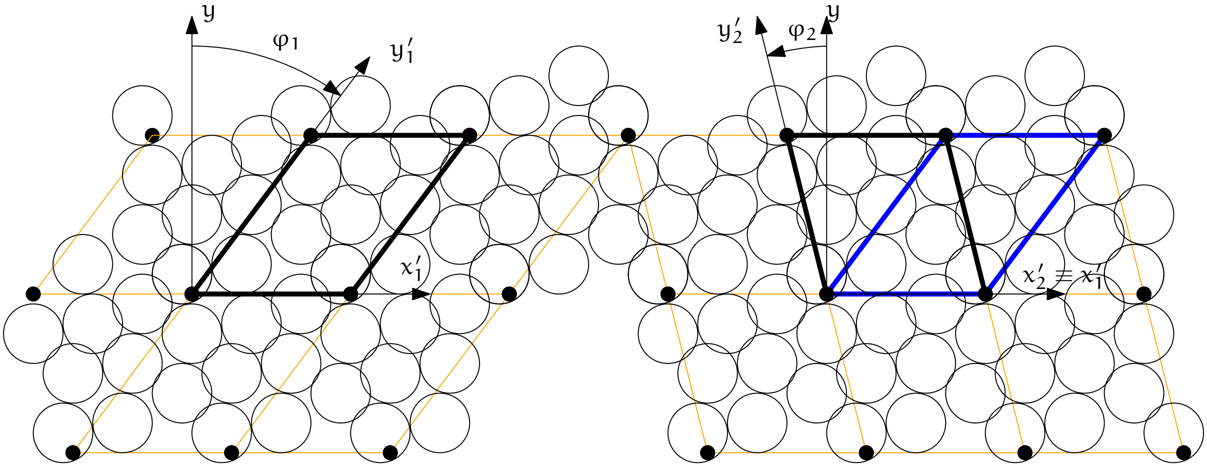


Figure 3.8: Flipping cell (utils.flipCell) to avoid infinite stretch of the bounding boxes' spans with growing φ. Cell flip does not affect interactions from the point of view of the simulation. The periodic arrangement on the left is the same as the one on the right, only the cell is situated differently 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.

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).

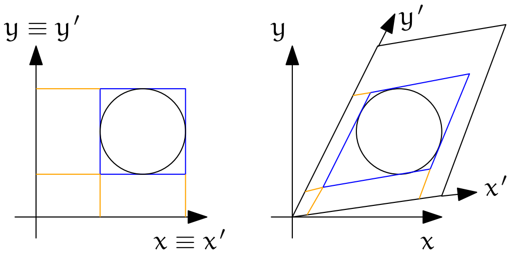


Figure 3.9: Constructing axis-aligned bounding box (Aabb) of a sphere in simulation space coordinates (without periodic cell – left) and transformed cell coordinates (right), where collision detection axes x', y' are not identical with simulation space axes x, y. Bounds' projection to axes is shown by 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 c. Multiplying current cell size $\mathbf{Ts}$ by 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 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 $\langle 0, 10)$, 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;
$(<\!\!\!\|)$	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 $(<\!\!\!\|)$ 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 $(\leq)$:

$$\begin{tikzcd}
4_1 & 9_1 & 12_2 & \| & \boxed{-2_4} & 5_0, \\
4_1 & 9_1 & 12_2 & \boxed{8_3} \arrow[l, "\nleq"] & \| & 5_0, \\
4_1 & 9_1 & \boxed{8_3} \arrow[l, "\nleq"] & 12_2 & \| & 5_0, \\
4_1 & \boxed{8_3} \arrow[l, "\leq"] & 9_1 & 12_2 & \| & 5_0.
\end{tikzcd}$$

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 $(\leq)$:

$$\begin{tikzcd}
4_1 & 8_3 & 9_1 & 12_2 & \| & \boxed{5_0}, \arrow[ll, "\nleq"] \\
4_1 & 8_3 & 9_1 & \boxed{5_{-1}} \arrow[l, "\nleq"] & \| & 2_3, \\
4_1 & 8_3 & \boxed{5_{-1}} \arrow[l, "\nleq"] & 9_1 & \| & 2_3, \\
4_1 & \boxed{5_{-1}} \arrow[l, "\leq"] & 8_3 & 9_1 & \| & 2_3.
\end{tikzcd}$$

We move (wrapping around) to $\boxed{4_1}$, which is ordered:

$$\begin{tikzcd}
\boxed{4_1} \arrow[rrrrr, bend right, "\geq"'] & 5_{-1} & 8_3 & 9_1 & \| & 2_3
\end{tikzcd}$$

and so is the last element

$$\begin{tikzcd}
4_1 & \boxed{5_{-1}} \arrow[l, "\leq"] & 8_3 & 9_1 & \| & 2_3.
\end{tikzcd}$$

3.7 Computational aspects

3.7.1 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.~ref{sect-dt-pwave}) 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 **:ref:‘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

$$Z \propto iN\frac{1}{n_{\mathrm{cpu}}} = \frac{T}{s_t r}\sqrt{\frac{E}{\rho}}p\frac{V}{\frac{4}{3}\pi r^3}\frac{1}{n_{\mathrm{cpu}}} \propto \frac{1}{r}\frac{1}{r^3} = \frac{1}{r^4}.$$

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.

3.7.2 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 (3.9) as

$$\frac{(\Delta F)_{dw}}{F_w} = -\lambda_d \operatorname{sgn} F_w \left(\dot{u}_w^{\ominus} + \frac{\ddot{u}_w^{\circ}\Delta t}{2}\right).$$

If the sgn argument is close to zero then the least significant finite precision artifact can determine whether the equation (relative increment of 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 4

User's manual

4.1 Scene construction

4.1.1 Triangulated surfaces

Yade integrates with the the GNU Triangulated Surface library, exposed in python via the 3rd party 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).

4.1.2 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)
```

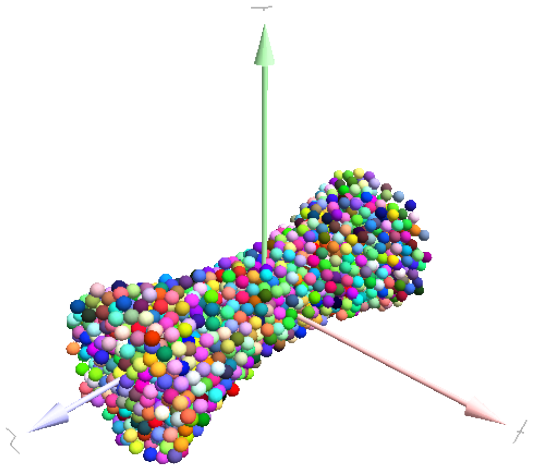


Figure 4.1: Specimen constructed with the pack.inHyperboloid predicate, packed with pack.randomDensePack.

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))
```

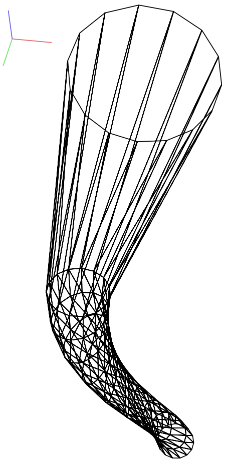


Figure 4.2: Triangulated funnel, constructed with the examples/funnel.py script.

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 scripts/test/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.

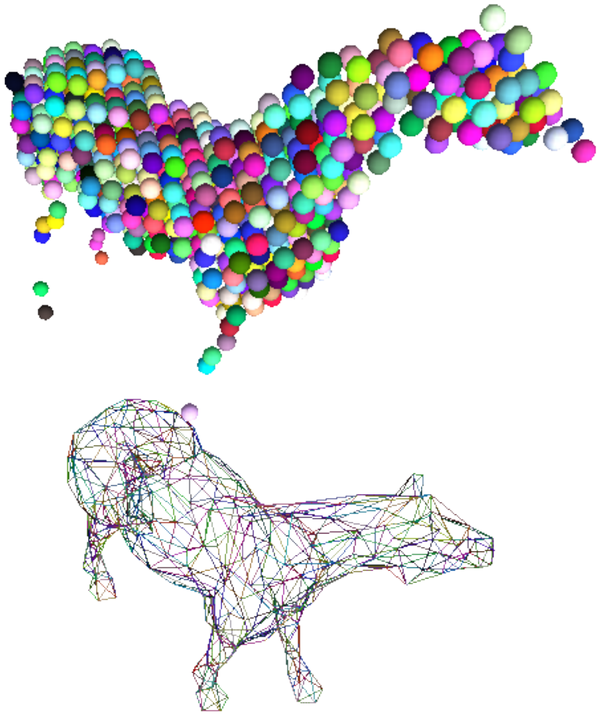


Figure 4.3: Imported GTS surface (horse) used as packing predicate (top) and surface constructed from facets (bottom). See http://www.youtube.com/watch?v=PZVruIlUX1A for movie of this simulation.

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)
```

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. utils.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 utils.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 utils.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

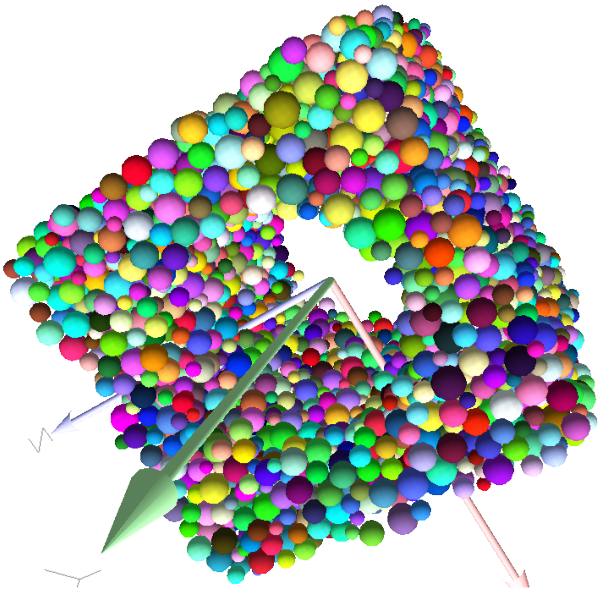


Figure 4.4: Box with cylinder removed from inside, using difference of these two predicates.

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(utils.sphere(c,r))
```

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

```
O.bodies.append([utils.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.

pack.SpherePadder constructs dense sphere packing based on pre-computed tetrahedron mesh; it is documented in pack.SpherePadder documentation; sample script is in scripts/test/SpherePadder.py. pack.SpherePadder does not return Body list as other algorithms, but a pack.SpherePack object; it can be iterated over, adding spheres to the simulation, as shown in its documentation.

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 scripts/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.

4.1.3 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 utils.sphere, utils.facet and utils.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 utils.sphere and other functions forconstructing body. The value returned by `O.materials.append` is an `id` of the material, which can be also passed to utils.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 utils.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 using utils.defaultMaterial; this should not happen for serious simulations, but is handy in simple scripts, where exact material properties are more or less irrelevant.

```
Yade [124]: len(O.materials)
 ->  [124]: 0

Yade [125]: idConcrete=O.materials.append(FrictMat(young=30e9,poisson=.2,frictionAngle=.6,label="concrete"))

Yade [126]: O.materials[idConcrete]
 ->  [126]: <FrictMat instance at 0xa7b3bc0>

# uses the last defined material
Yade [128]: O.bodies.append(utils.sphere(center=(0,0,0),radius=1))
 ->  [128]: 0

# material given by id
Yade [130]: O.bodies.append(utils.sphere((0,0,2),1,material=idConcrete))
 ->  [130]: 1

# material given by label
Yade [132]: O.bodies.append(utils.sphere((0,2,0),1,material="concrete"))
 ->  [132]: 2

Yade [133]: idSteel=O.materials.append(FrictMat(young=210e9,poisson=.25,frictionAngle=.8,label="steel"))

Yade [134]: len(O.materials)
 ->  [134]: 2

# implicitly uses "steel" material, as it is the last one now
Yade [136]: O.bodies.append(utils.facet([(1,0,0),(0,1,0),(-1,-1,0)]))
 ->  [136]: 3
```

Adding multiple particles

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

```
Yade [138]: O.bodies.append(utils.sphere((0,0,0),1))
 ->  [138]: 0

Yade [139]: O.bodies.append(utils.sphere((0,0,2),1))
```

```
 ->  [139]: 1

# this is the same, but in one function call
Yade [141]: O.bodies.append([
     .....:    utils.sphere((0,0,0),1),
     .....:    utils.sphere((0,0,2),1)
     .....: ])
 ->  [144]: [2, 3]
```

Many functions introduced in preceding 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 utils.sphere and utils.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); a special function BodyContainer.appendClumped is designed for this task; for instance, we might add 2 spheres tied together:

```
Yade [146]: O.bodies.appendClumped([
     .....:    utils.sphere([0,0,0],1),
     .....:    utils.sphere([0,0,2],1)
     .....: ])
 ->  [149]: (2, [0, 1])

Yade [150]: len(O.bodies)
 ->  [150]: 3

Yade [151]: O.bodies[1].isClumpMember, O.bodies[2].clumpId
 ->  [151]: (True, 2)

Yade [152]: O.bodies[2].isClump, O.bodies[2].clumpId
 ->  [152]: (True, 2)
```

appendClumped returns a tuple of `(clumpId,[memberId1,memberId2])`: clump is internally represented by a special Body, referenced by clumpId of its members (see also isClump, isClumpMember and isStandalone).

4.1.4 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. (Either Ig2_Sphere_Sphere_Dem3DofGeom.distFactor or Ig2_Sphere_Sphere_ScGeom.interactionDetectionFactor, depending on the functor that is in use.

Note that only Sphere + Sphere interactions are supported; there is no parameter analogous to distFactor in Ig2_Facet_Sphere_Dem3DofGeom. 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_Dem3DofGeom(distFactor=intRadius,label='ig2ss'),
         Ig2_Facet_Sphere_Dem3DofGeom(),
      ],
      [Ip2_CpmMat_CpmMat_CpmPhys()],
      [Law2_Dem3DofGeom_CpmPhys_Cpm(epsSoft=0)], # deactivated
   ),
   NewtonIntegrator(damping=damping,label='damper'),
]

# run one single step
O.step()

# reset interaction radius to the default value
# see documentation of those attributes for the meaning of negative values
bo1s.aabbEnlargeFactor=-1
ig2ss.distFactor=-1

# 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 utils.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 [154]: O.materials.append(FrictMat(young=3e10,poisson=.2,density=1000))
 ->  [154]: 0

Yade [155]: O.bodies.append([
      .....:    utils.sphere([0,0,0],1),
      .....:    utils.sphere([0,0,1000],1)
      .....: ])
 ->  [158]: [0, 1]

# only add InteractionLoop, no other engines are needed now
Yade [159]: O.engines=[
     .....:    InteractionLoop(
     .....:        [Ig2_Sphere_Sphere_Dem3DofGeom(),],
     .....:        [Ip2_FrictMat_FrictMat_FrictPhys()],
     .....:        [] # not needed now
     .....:    )
     .....: ]

Yade [166]: i=utils.createInteraction(0,1)

# created by functors in InteractionLoop
Yade [167]: i.geom, i.phys
 ->  [167]:
(<Dem3DofGeom_SphereSphere instance at 0xbf500a0>,
 <FrictPhys instance at 0xb4030d0>)
```

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 utils.createInteraction function accept list of pairs id's as well.

4.1.5 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:]
```

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_Dem3DofGeom()],[],[])
   ```

   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_Dem3DofGeom(),
      Ig2_Facet_Sphere_Dem3DofGeom()
   ],[],[])
   ```

   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_Dem3DofGeom_CpmPhys_Cpm accepts Dem3DofGeom). This is (for most cases) either Dem3DofGeom (total shear formulation) or ScGeom (incremental shear formulation). For ScGeom, the above example would simply change to:

```
InteractionLoop([
   Ig2_Sphere_Sphere_ScGeom(),
   Ig2_Facet_Sphere_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_Basic 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 Ig2 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.

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_Basic constitutive law. We see that

1. the Ig2 functors most create ScGeom. Since we have spheres and walls in the simulation, we will need functors accepting Sphere + Sphere and Wall + Sphere combinations. We don't want interactions between walls themselves (as a matter of fact, there is no such functor anyway). That gives us Ig2_Sphere_Sphere_ScGeom and Ig2_Wall_Sphere_ScGeom (as a matter of facet, there is no such functor now, although it is planned)
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_Wall_Sphere_ScGeom()],
   [Ip2_FrictMat_FrictMat_FrictPhys()],
   [Law2_ScGeom_FrictPhys_Basic()]
)
```

Concrete model

In this case, our goal is to use the Law2_Dem3DofGeom_CpmPhys_Cpm constitutive law.

- We use spheres and facets in the simulation, which selects `Ig2` functors accepting those types and producing Dem3DofGeom: Ig2_Sphere_Sphere_Dem3DofGeom and Ig2_Facet_Sphere_Dem3DofGeom.
- 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_Dem3DofGeom(),Ig2_Facet_Sphere_Dem3DofGeom()],
   [Ip2_CpmMat_CpmMat_CpmPhys()],
   [Law2_Dem3DofGeom_CpmPhys_Cpm()]
)
```

4.1.6 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 moved by NewtonIntegrator; it is mandatory for bodies with zero mass, where applying non-zero force would result in infinite displacement.

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

  Making sphere non-dynamic is achieved simply by:

  ```
  utils.sphere([x,y,z],radius,dynamic=False)
  ```

  Note: There is an open bug #398089 to define exactly what the `dynamic` flag does. Please read it before writing a new engine relying on this flag.

- 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 [169]: O.bodies.append(utils.sphere((0,0,0),1))
   ->  [169]: 0

  Yade [170]: O.bodies[0].state.blockedDOFs=['z','rx','ry']
  ```

  In contrast to Body.dynamic, blockedDOFs will only block forces (and acceleration) in that direction being effective; if you prescribed linear or angular velocity, they will be applied regardless of blockedDOFs. (This is also related to bug #398089 mentioned above)

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
```

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.

Partial engines

Engines deriving from PartialEngine define the subscribedBodies 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.

If you need an engine applying non-constant value instead, there are several interpolating engines (InterpolatingDirectedForceEngine for applying force with varying magnitude, InterpolatingSpiralEngine 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.

4.1.7 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 [172]: dict(O.tags) # convert to real dictionary
 ->  [172]:
{'author': 'root~(root@zirconium)',
 'd.id': '20110728T022427p2823',
 'id': '20110728T022427p2823',
 'id.d': '20110728T022427p2823',
 'isoTime': '20110728T022427'}
```

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 *batch-output-separate* 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 [173]: O.tags['anythingThat I lik3']='whatever'

Yade [174]: O.tags['anythingThat I lik3']
 ->  [174]: '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 utils.saveVars and utils.loadVars.

utils.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 utils.loadVars with the same identification *mark*:

```
Yade [175]: a=45; b=pi/3

Yade [176]: utils.saveVars('ab',a=a,b=b)

# save simulation (we could save to disk just as well)
Yade [176]: O.saveTmp()

Yade [178]: O.loadTmp()
Yade [179]: utils.loadVars('ab')

Yade [180]: yade.params.ab.a
 ->  [180]: 45

# import like this
Yade [181]: from yade.params import ab

Yade [182]: ab.a, ab.b
 ->  [182]: (45, 1.0471975511965976)

# also possible
Yade [183]: from yade.params import *

Yade [184]: ab.a, ab.b
 ->  [184]: (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)
        #
        utils.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 utils.saveVars.

4.2 Controlling simulation

4.2.1 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 utils.kineticEnergy) every 5 seconds, this engine will be put

PyRunner(command=”print ‘kinetic energy’,utils.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 [186]: from yade import plot

Yade [187]: plot.data
 ->  [187]:
{'eps': [0.0001, 0.001, nan],
 'force': [nan, nan, 1000.0],
 'sigma': [12, nan, nan]}

Yade [188]: 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 [189]: plot.addData(eps=1e-3)

# adds NaNs to already existing sigma and eps columns
Yade [190]: plot.addData(force=1e3)

Yade [191]: plot.data
 ->  [191]:
{'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 [192]: plot.data['eps']
 ->  [192]: [0.0001, 0.001, nan, 0.0001, 0.001, nan]

# get maximum eps
Yade [193]: max(plot.data['eps'])
 ->  [193]: 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.stress,eps=strainer.strain)
```

In that case, naturally, the labeled object must define attributes which are used (UniaxialStrainer.strain and UniaxialStrainer.stress 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')}
```

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

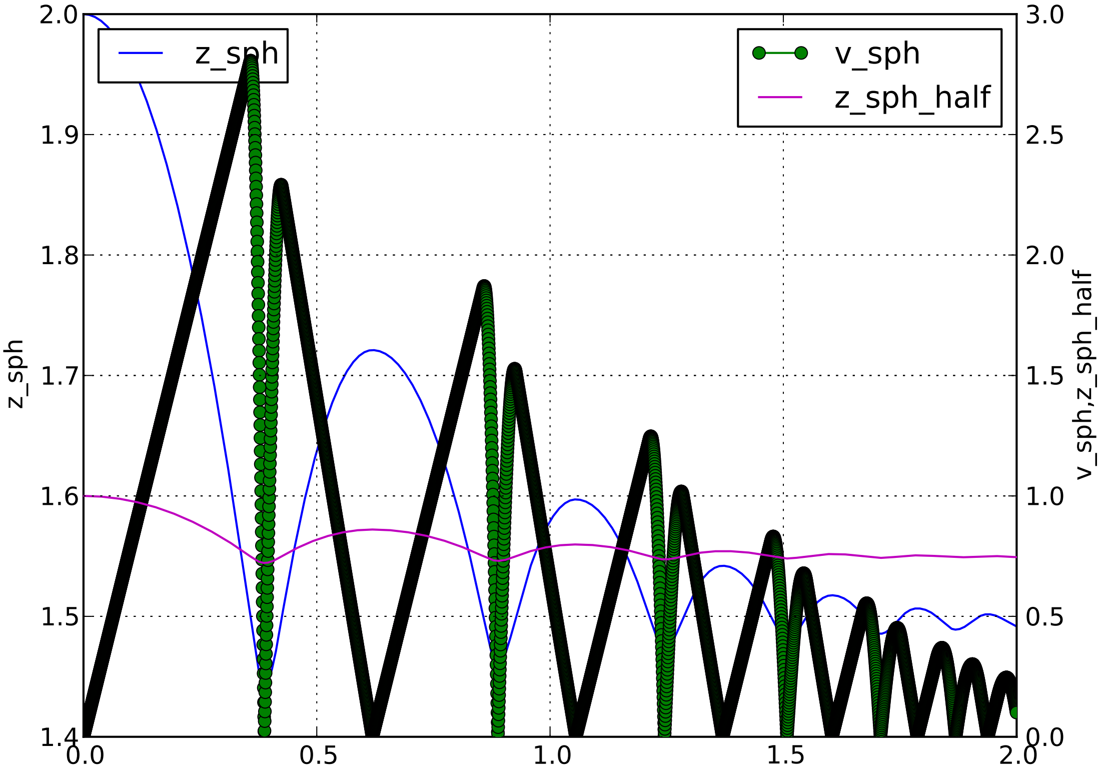


Figure 4.5: Figure generated by examples/simple-scene/simple-scene-plot.py.

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-.'))}
```

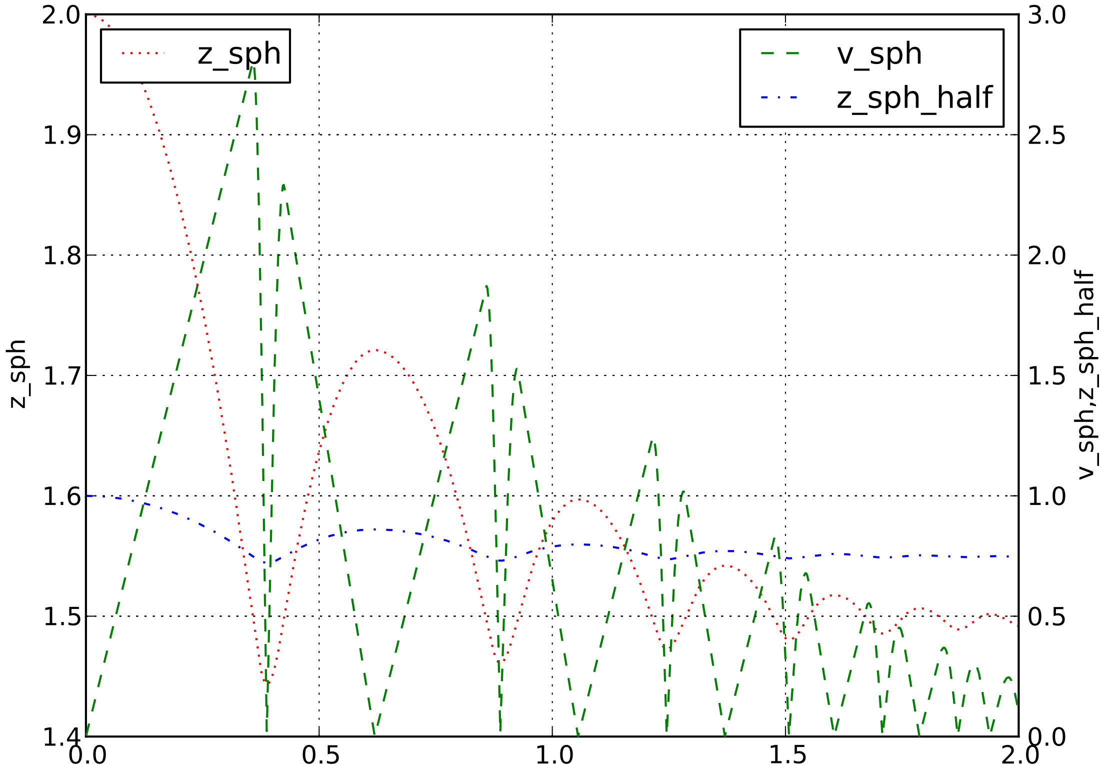


Figure 4.6: Figure generated by changing parameters to plot.plots as above.

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.

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':'$z_{sph}/2$'}
```

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

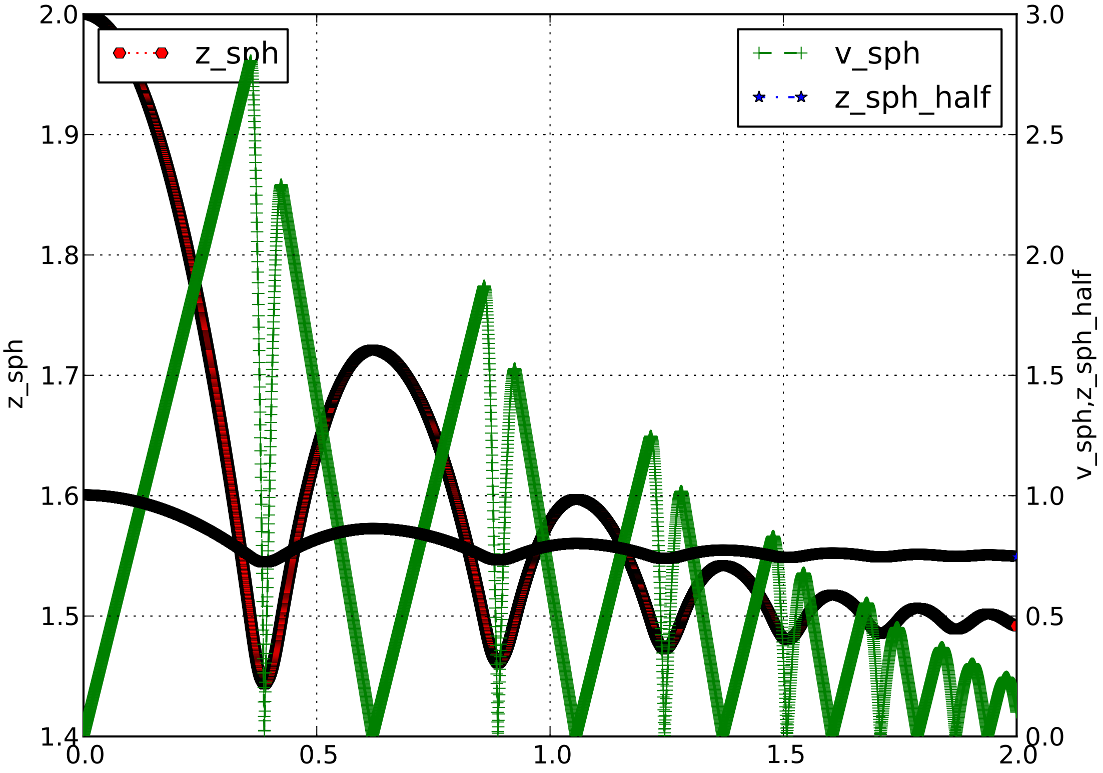


Figure 4.7: Figure generated by changing parameters to plot.plots as above.

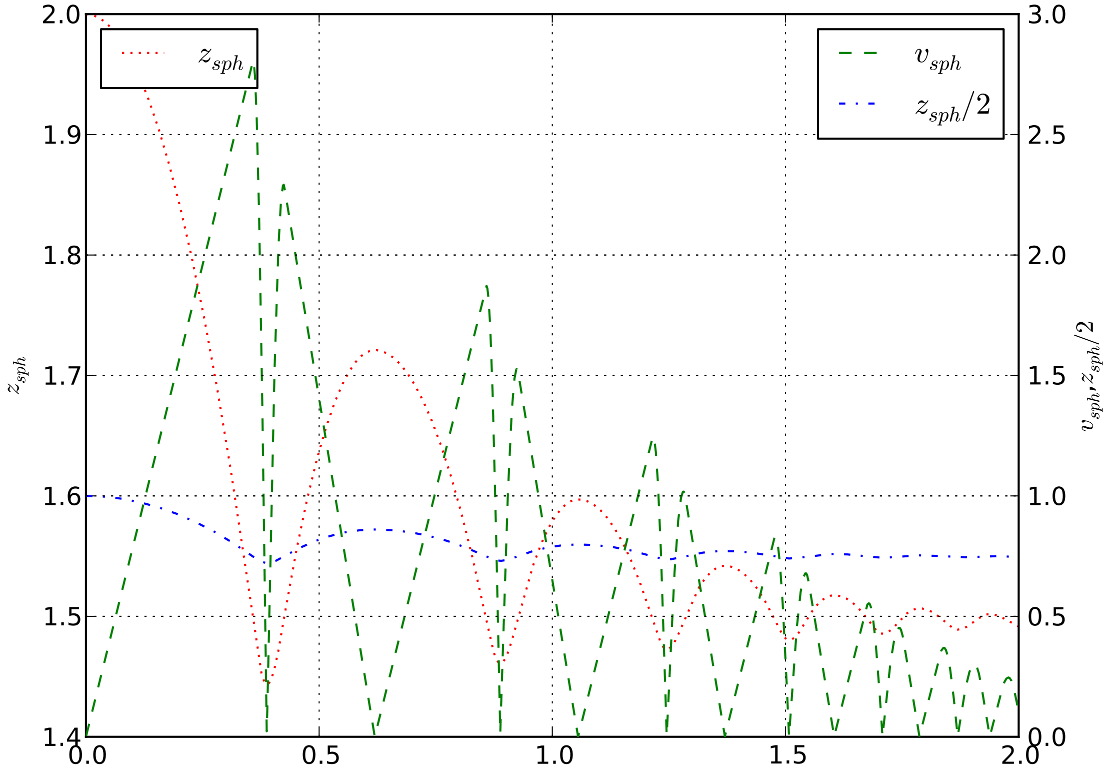


Figure 4.8: Figure generated by examples/simple-scene/simple-scene-plot.py, with TeX labels.

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

Yade [198]: plot.plots
 ->  [198]: {'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 [199]: plot.plots={
     .....:    'i':('t',),
     .....:    'i ':('z1','v1') # note the space in 'i '
     .....: }

Yade [203]: plot.plots
 ->  [203]: {'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.

- 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.

4.2.2 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 _utils.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 utils.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 utils.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()
```

4.2.3 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:

```
$ 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.

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-multi.in for details.

4.2.4 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 utils.readParamsFromTable to obtain parameters from parameter table. In order to make the script runnable outside the batch, utils.readParamsFromTable takes default values of parameters, which might be overridden from the parameter table.

utils.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 utils.readParamsFromTable (using utils.TableParamReader class), called from simulation script, as explained above.

The batch can be run as

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

and it will intelligently run one simulation for each parameter table line.

Example

This example is found in scripts/batch.table and scripts/batch.py.

Suppsoe 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:

```
from yade import utils
utils.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 utils.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 b
```

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.

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

```
utils.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 compuation

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, which prints:

```
gnuplot 20100413T144723p7625.gnuplot
```

to the output (redirected to log file).

This file itself contains single graph:

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

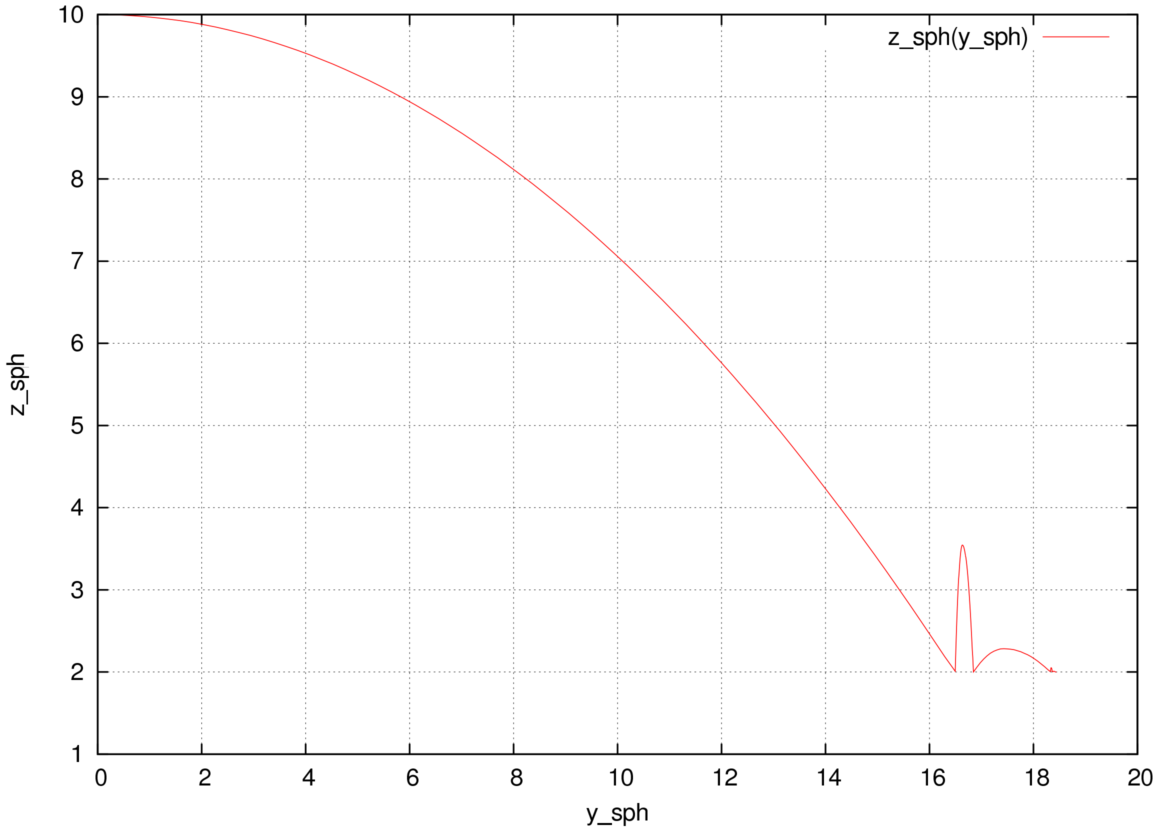


Figure 4.9: Figure from single job in the batch.

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.

4.3 Postprocessing

4.3.1 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.py for a full example):

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

   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 utils.encodeVideoFromFrames:

```
utils.encodeVideoFromFrames(snapshooter.savedSnapshots,out='/tmp/bulldozer.ogg',fps=2)
```

   The video is encoded in the Theora format stored in an ogg container.

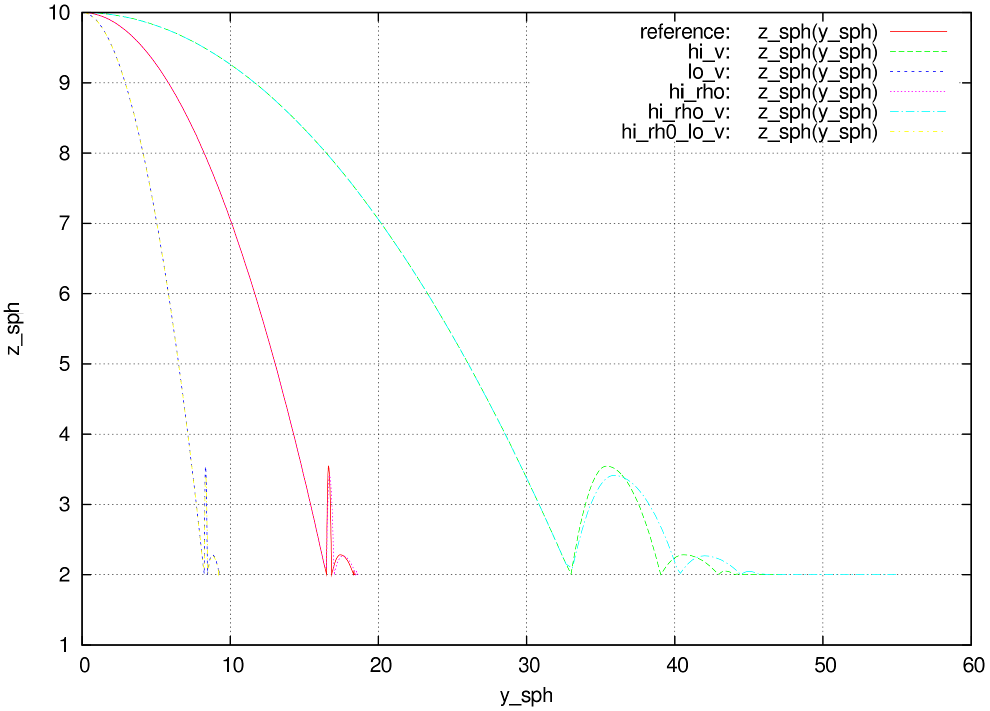
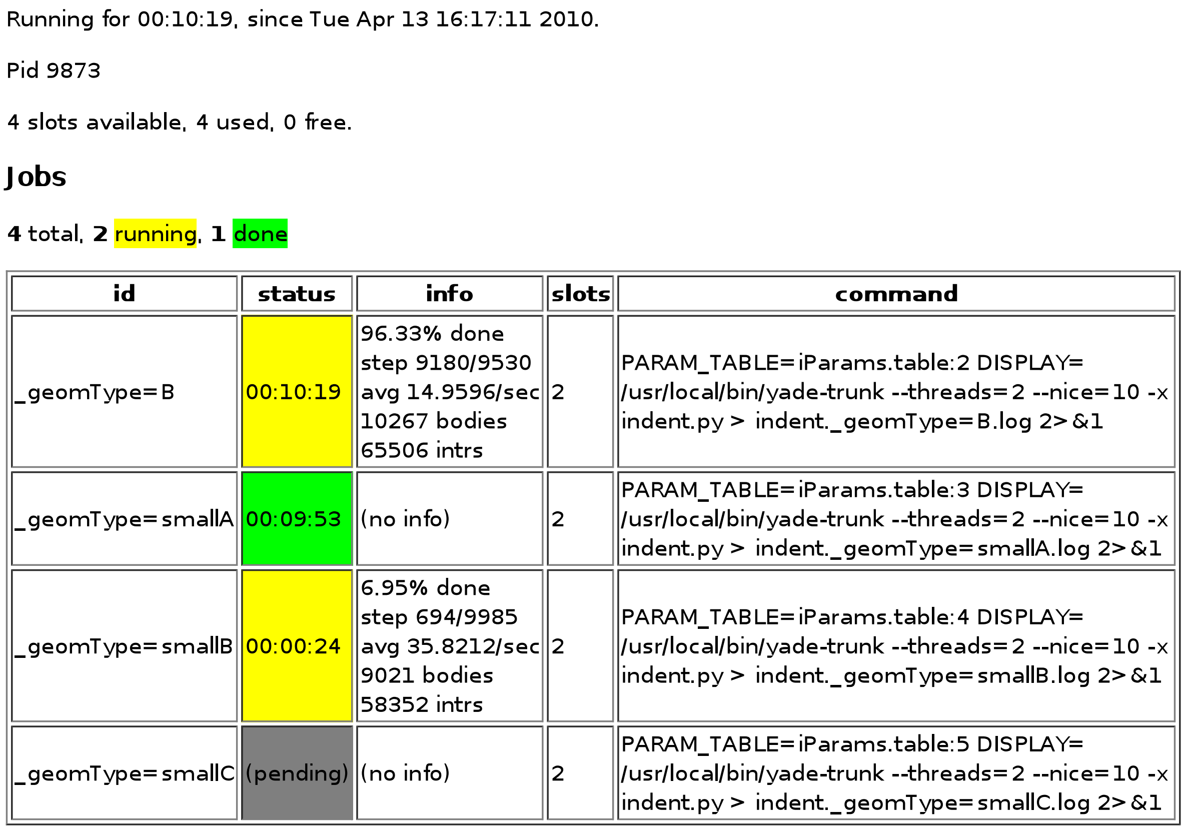


Figure 4.10: Merged figure from all jobs in the batch. Note that labels are prepended by job description to make lines distinguishable.



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

Figure 4.11: Summary page available at port 9080 as batch is processed (updates every 5 seconds automatically). Possible job statuses are pending, running, done, failed.

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

Paraview

Saving data during the simulation

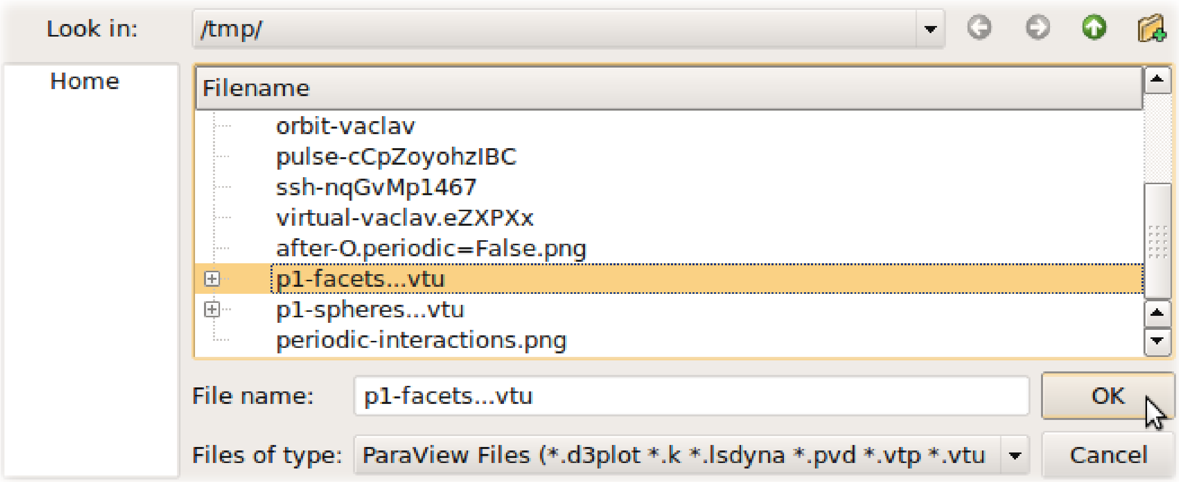
Paraview is based on the Visualization Toolkit, which defines formats for saving various types of data. One of them (with the .vtu extension) can be written by a special engine VTKRecorder. It is added to the simulation loop:

```
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 (see the documentation)

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:



Look in: /tmp/

Filename
orbit-vaclav
pulse-cCpZoyohzIBC
ssh-nqGvMp1467
virtual-vaclav.eZXPXx
after-O.periodic=False.png
p1-facets...vtu
p1-spheres...vtu
periodic-interactions.png

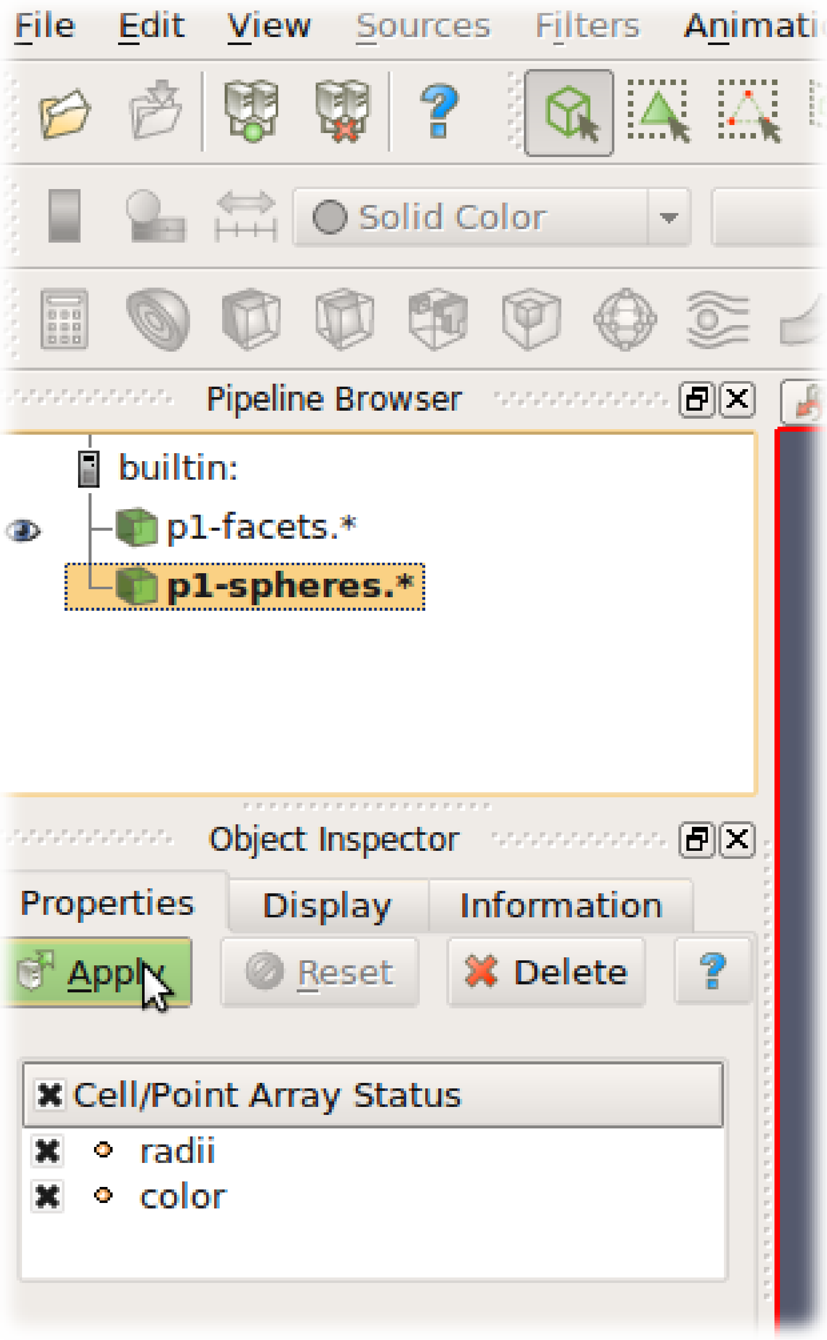
File name: p1-facets...vtu

Files of type: ParaView Files (*.d3plot *.k *.lsdyna *.pvd *.vtp *.vtu

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":

Rendering spherical particles 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*


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

- “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.

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*.

4.4 Python specialties and tricks

4.5 Extending Yade

- new particle shape
- new constitutive law

4.6 Troubleshooting

4.6.1 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 you 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.

4.6.2 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.

4.6.3 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 5

Programmer's manual

5.1 Build system

Yade uses [scons] build system for managing the build process. It takes care of configuration, compilation and installation. SCons is written in python and its build scripts are in python, too. SCons complete documentation can be found in its manual page.

5.1.1 Pre-build configuration

We use `$` to denote build variable in strings in this section; in SCons script, they can be used either by writing `$variable` in strings passed to SCons functions, or obtained as attribute of the `Environment` instance `env`, i.e. `env['variable']`; we use the formed in running text here.

In order to allow parallel installation of multiple yade versions, the installation location follows the pattern `$PREFIX/lib/yade$SUFFIX` for libraries and `$PREFIX/bin/yade$SUFFIX` for executables (in the following, we will refer only to the first one). `$SUFFIX` takes the form `-$version$variant`, which further allows multiple different builds of the same version (typically, optimized and debug builds). For instance, the default debug build of version 0.5 would be installed in `/usr/local/lib/yade-0.5-dbg/`, the executable being `/usr/local/bin/yade-0.5-dbg`.

The build process takes place outside the source tree, in directory reffered to as `$buildDir` within those scripts. By default, this directory is `../build-$SUFFIX`.

Each build depends on a number of configuration parameters, which are stored in mutually independent *profiles*. They are selected according to the `profile` argument to scons (by default, the last profile used, stored in `scons.current-profile`). Each profile remembers its non-default variables in `scons.profile-$profile`.

There is a number of configuration parameters; you can list all of them by `scons -h`. The following table summarizes only a few that are the most used.

`PREFIX` [**default: `/usr/local`**] installation prefix (`PREFIX` preprocessor macro; `yade.config.prefix` in python

`version` [**bzr revision (e.g. bzr1899)**] first part of suffix (`SUFFIX` preprocessor macro; `yade.config.suffix` in python)]

`variant` [***(empty)***] second part of suffix

`buildPrefix` [`..`] where to create `build-$SUFFIX` directory

`debug` [***False*** **(0)**] add debugging symbols to output, enable stack traces on crash

`optimize` [***True*** **(1)**] optimize binaries (`#define NDEBUG`; assertions eliminated; `YADE_CAST` and `YADE_PTR_CAST` are static casts rather than dynamic; LOG_TRACE and LOG_DEBUG are eliminated)

CPPPATH [/usr/include/vtk-5.2:/usr/include/vtk-5.4] additional colon-separated paths for preprocessor (for atypical header locations). Required by some libraries, such as VTK (reflected by the default)

LIBPATH [*(empty)*] additional colon-separated paths for linker

CXX [g++] compiler executable

CXXFLAGS [*(empty)*] additional compiler flags (may are added automatically)

jobs [4] number of concurrent compilations to run

brief [*True* (1)] only show brief notices about what is being done rather than full command-lines during compilation

linkStrategy [monolithic] whether to link all plugins in one shared library (`monolithic`) or in one file per plugin (`per-class`); the first option is faster for overall builds, while the latter one makes recompilation of only part of Yade faster; granularity of monolithic build can be changed with the `chunkSize` parameter, which determines how many files are compiled at once.

features [log4cxx,opengl,gts,openmp] optional comma-separated features to build with (details below; each defines macro `YADE_$FEATURE`; available as lowercased list `yade.config.features` at runtime

Library detection

When the `scons` command is run, it first checks for presence of all required libraries. Some of them are *essential*, other are *optional* and will be required only if features that need them are enabled.

Essentials

compiler Obviously c++ compiler is necessary. Yade relies on several extensions of `g++` from the [gcc] suite and cannot (probably) be built with other compilers.

boost [boost] is a large collection of peer-reviewed c++ libraries. Yade currently uses thread, date_-time, filesystem, iostreams, regex, serialization, program_options, foreach, python; typically the whole boost bundle will be installed. If you need functionality from other modules, you can make presence of that module mandatory. Only be careful about relying on very new features; due to range of systems yade is or might be used on, it is better to be moderately conservative (read: roughly 3 years backwards compatibility).

python [python] is the scripting language used by yade. Besides [boost::python]_, yade further requires

- [ipython] (terminal interaction)
- [matplotlib] (plotting)
- [numpy] (matlab-like numerical functionality and accessing numpy arrays from `c/c++` efficiently)

Optional libraries (features)

The *features* parameter controls optional functionality. Each enabled feature defines preprocessor macro *YADE_FEATURE* (name uppercased) to enable selective exclude/include of parts of code. In some cases, it would be meaningless to compile some file at all (e.g. *VTKRecorder* without the *vtk* feature). This can be controller using the *YADE_REQUIRE_FEATURE* places in the respective implementation file (see the *Linking* section for more details).

log4cxx (YADE_LOG4CXX) Enable flexible logging system ([log4cxx]), which permits to assign logging levels on per-class basis; doesn't change API, only redefines *LOG_INFO* and other macros accordingly; see *log4cxx* for details.

opengl (YADE_OPENGL) Enable 3d rendering as well as the Qt3-based graphical user interface (in addition to python console).

vtk (YADE_VTK) Enable functionality using Visualization Toolkit ([vtk]; e.g. VTKRecorder exporting to files readable with ParaView).

openmp (YADE_OPENMP) Enable parallelization using OpenMP, non-intrusive shared-memory parallelization framework; it is only supported for `g++` > 4.0. Parallel computation leads to significant performance increase and should be enabled unless you have a special reason for not doing so (e.g. single-core machine). See *upyade-parallel* for details.

gts (YADE_GTS) Enable functionality provided by GNU Triangulated Surface library ([gts]) and build PyGTS, its python interface; used for surface import and construction.

cgal (YADE_CGAL) Enable functionality provided by Computation Geometry Algorithms Library ([cgal]); triangulation code in MicroMacroAnalyser and PersistentTriangulationCollider ses its routines.

other There might be more features added in the future. Always refer to `scons -h` output for possible values.

Warning: Due to a long-standing bug in log4cxx, using log4cxx will make yade crash at every exit. We work-around this partially by disabling the crash handler for regular exits, but process exit status will still be non-zero. The batch system (*yade-multi*) detects successful runs by looking at magic line "Yade: normal exit." in the process' standard output.

Before compilation, SCons will check for presence of libraries required by their respective features [1]. Failure will occur if a respective library isn't found. To find out what went wrong, you can inspect `../build-$SUFFIX/config.log` file; it contains exact commands and their output for all performed checks.

Note: Features are not auto-detected on purpose; otherwise problem with library detection might build Yade without expected features, causing specifically problems for automatized builds.

5.1.2 Building

Yade source tree has the following structure (omiting `debian`, `doc`, `examples` and `scripts` which don't participate in the build process); we shall call each top-level component *module*:

```
attic/          ## code that is not currently functional and might be removed unless resurrected
   lattice/        ## lattice and lattice-like models
   snow/           ## snow model (is really a DEM)
core/           ## core simulation building blocks
extra/          ## miscillanea
gui/            ## user interfaces
   qt3/            ## 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
```

Each directory on the top of this hierarchy (except `pkg`, which is treated specially – see below) contains file `SConscript`, determining what files to compile, how to link them together and where should they be installed. Within these script, a scons variable `env` (build `Environment`) contains all the configuration parameters, which are used to influence the build process; they can be either obtained with the `[]` operator, but scons also replaces `$var` strings automatically in arguments to its functions:

```
if 'opengl' in env['features']:
        env.Install('$PREFIX/lib/yade$SUFFIX/',[
```

[1] Library checks are defined inside the `SConstruct` file and you can add your own, should you need it.

```
                # ...
        ])
```

Header installation

To allow flexibility in source layout, SCons 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`	`<yade/core/Scene.hpp>`
`lib/base/Logging.hpp`	`<yade/lib-base/Logging.hpp>`
`lib/serialization/Serializable.hpp`	`<yade/lib-serialization/Serializable.hpp>`
`pkg/dem/DataClass/SpherePack.hpp`	`<yade/pkg-dem/SpherePack.hpp>`
`gui/qt3/QtGUI.hpp`	`<yade/gui-qt3/QtGUI.hpp>`

It is advised to use `#include<yade/module/Class.hpp>` style of inclusion rather than `#include"Class.hpp` even if you are in the same directory.

What files to compile

`SConscript` files in `lib`, `core`, `gui`, `py` and `extra` explicitly determine what files will be built.

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 SCons 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.

Note: The source scanner was written by hand and is not official part of SCons. It is fairly primitive and in particular, it doesn't interpret c preprocessor macros, except for a simple non-nested feature-checks like `#ifdef YADE_*`/`#ifndef YADE_*` #endif.

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` for monolithic builds, `libClass.so` for per-class (per-plugin) builds.	`core`, `yade-support`; may **not** depend on external libraries explcitly (only implicitly, by adding the library to global linker flags in `SConstruct`)
extra	(undefined)	(arbitrary)
gui	`libQtGUI.so`, `libPythonUI.so`	`lib`, `core`, `pkg`
py	(many files)	`lib`, `core`, `pkg`, external

Because `pkg` plugins might be linked differently depending on the `linkStrategy` option, `SConscript` files that need to explicitly declare the dependency should use provided `linkPlugins` function which returns libraries in which given plugins will be defined:

```
env.SharedLibrary('_packSpheres',['_packSpheres.cpp'],
        SHLIBPREFIX='',
        LIBS=env['LIBS']+[linkPlugins(['Shop','SpherePack']),]
),
```

Note: `env['LIBS']` are libraries that all files are linked to and they should always be part of the `LIBS` parameter.

Since plugins in `pkg` are not declared in any `SConscript` file, other plugins they depend on are again found *automatically* by scannig their `#include` directives for the pattern `#include<yade/module/Plugin.hpp>`. Again, this works well in normal circumastances, but is not necessarily robust.

See scons manpage for meaning of parameters passed to build functions, such as `SHLIBPREFIX`.

5.2 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 logging abundantly. Again, `LOG_TRACE` and `LOG_DEBUG` are eliminated from optimized code; unless turned on explicitly, the ouput will be suppressed even in the debug build (see below).
- 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)

5.2.1 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 [2] 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. 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_Dem3DofGeom is binary functor called for Facet and Sphere, creating and instace of Dem3DofGeom.
- Law2_Dem3DofGeom_CpmPhys_Cpm is binary functor (LawFunctor) called for types Dem3Dof (Geom) and CpmPhys.

[2] Not considering OpenGL dispatchers, which might be replaced by regular virtual functions in the future.

5.2.2 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`.

Historically, Yade was using Doxygen for in-source documentation. This documentation is still available (by running `scons doc`), but was rarely written and used by programmers, and had all the disadvantages of auto-generated documentation. Then, as Python became ubiquitous in yade, python was documented using epydoc generator. Finally, hand-written documentation (this one) started to be written using Sphinx, which was developed originally for documenting Python itself. Disadvantages of the original scatter were different syntaxes, impossibility for cross-linking, non-interactivity and frequently not being up-to-date.

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 intergration 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).

Local documentation

Note: At some future point, this documentation will be integrated into yade's sources. This section should be updated accordingly in that case.

To generate Yade's documentation locally, get a copy of the ydoc branch via bzr, then follow instructions in the README file.

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.

5.3 Support framework

Besides the framework provided by the c++ standard library (including STL), boost and other dependencies, yade provides its own specific services.

5.3.1 Pointers

Shared pointers

Yade makes extensive use of shared pointers `shared_ptr`. [3] 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.

[3] 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.

YADE_PTR_CAST expands to static_pointer_cast in optimized builds and to dynamic_pointer_cast in debug builds.

5.3.2 Basic numerics

The floating point type to use in Yade Real, which is by default typedef for double. [4]

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 [61]: Vector3(1,2,3).cross(Vector3(0,0,1))
 ->  [61]: Vector3(2,-1,0)

# construct quaternion from axis and angle
Yade [63]: Quaternion(Vector3(0,0,1),pi/2)
 ->  [63]: 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.

5.3.3 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).

[4] 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.

- `std::string getBaseClassName(unsigned i=0)` returning name of *i*-th base class (here, `Bar` for i=0 and `Baz` for i=1).

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 [65]: yade.system.childClasses('Shape')
 ->  [65]:
set(['Box',
     'ChainedCylinder',
     'Clump',
     'Cylinder',
     'Facet',
     'Sphere',
     'Tetra',
     'Wall'])

Yade [66]: Sphere().name            ## getClassName()
WARN: Sphere.name is deprecated, use:
WARN: * Sphere.__class__.__name__ to get the class name (as string)
WARN: * isinstance(object,Sphere) to test whether object is of type Sphere.

 ->  [66]: 'Sphere'
```

5.3.4 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<yade/pkg-common/GravityEngine.hpp>
#include<yade/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 [67]: O.engines=[GravityEngine(gravity=Vector3(0,0,-9.81))]

Yade [68]: 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="9">
```

```
<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>1e-08</dt>
                <iter>0</iter>
                <subStepping>0</subStepping>
                <subStep>-1</subStep>
                <time>0</time>
                <stopAtIter>0</stopAtIter>
                <isPeriodic>0</isPeriodic>
                <trackEnergy>0</trackEnergy>
                <needsInitializers>1</needsInitializers>
                <selectedBody>-1</selectedBody>
                <tags class_id="3" tracking_level="0" version="0">
                        <count>5</count>
                        <item_version>0</item_version>
                        <item>author=root~(root@zirconium)</item>
                        <item>isoTime=20110728T022424</item>
                        <item>id=20110728T022424p2823</item>
                        <item>d.id=20110728T022424p2823</item>
                        <item>id.d=20110728T022424p2823</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>
                                                                <label></label>
                                                        </Engine>
                                                </GlobalEngine>
                                        </FieldApplier>
                                        <gravity class_id="10" tracking_level="0" version="0">
                                                <x>0</x>
                                                <y>0</y>
                                                <z>-9.8100000000000005</z>
                                        </gravity>
                                </px>
                        </item>
                </engines>
                <initializers>
                        <count>0</count>
                        <item_version>1</item_version>
                </initializers>
                <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>
                        </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>
                                <refSize>
                                        <x>1</x>
                                        <y>1</y>
                                        <z>1</z>
                                </refSize>
                                <trsf class_id="27" tracking_level="0" version="0">
                                        <m00>1</m00>
                                        <m01>0</m01>
                                        <m02>0</m02>
                                        <m10>0</m10>
                                        <m11>1</m11>
                                        <m12>0</m12>
                                        <m20>0</m20>
                                        <m21>0</m21>
                                        <m22>1</m22>
                                </trsf>
                                <velGrad>
                                        <m00>0</m00>
                                        <m01>0</m01>
                                        <m02>0</m02>
                                        <m10>0</m10>
                                        <m11>0</m11>
                                        <m12>0</m12>
                                        <m20>0</m20>
                                        <m21>0</m21>
                                        <m22>0</m22>
                                </velGrad>
                                <prevVelGrad>
                                        <m00>0</m00>
                                        <m01>0</m01>
                                        <m02>0</m02>
                                        <m10>0</m10>
                                        <m11>0</m11>
                                        <m12>0</m12>
                                        <m20>0</m20>
                                        <m21>0</m21>
                                        <m22>0</m22>
                                </prevVelGrad>
                                <Hsize>
                                        <m00>1</m00>
                                        <m01>0</m01>
                                        <m02>0</m02>
                                        <m10>0</m10>
                                        <m11>1</m11>
                                        <m12>0</m12>
                                        <m20>0</m20>
                                        <m21>0</m21>
                                        <m22>1</m22>
                                </Hsize>
                                <homoDeform>3</homoDeform>
                        </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 [69]: s=Sphere()

Yade [70]: s.radius              ## read-access
 ->  [70]: nan

Yade [71]: s.radius=4.           ## write access

Yade [72]: s.keys()              ## show all available keys
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)

/build/buildd/yade-0.60.3/doc/sphinx/<ipython console> in <module>()

AttributeError: 'Sphere' object has no attribute 'keys'

Yade [73]: for k in s.keys(): print s[k]  ## iterate over keys, print their values
     ....:
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
```

```
/build/buildd/yade-0.60.3/doc/sphinx/<ipython console> in <module>()

AttributeError: 'Sphere' object has no attribute 'keys'

Yade [74]: s.has_key('radius')           ## same as: 'radius' in s.keys()
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)

/build/buildd/yade-0.60.3/doc/sphinx/<ipython console> in <module>()

AttributeError: 'Sphere' object has no attribute 'has_key'

Yade [75]: s.dict()                       ## show dictionary of both attributes and values
 ->  [75]: {'color': Vector3(1,1,1), 'highlight': False, 'radius': 4.0, 'wire': False}

## only very rarely needed; calls Serializable::postProcessAttributes(bool deserializing):
Yade [77]: s.postProcessAttributes(False)
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)

/build/buildd/yade-0.60.3/doc/sphinx/<ipython console> in <module>()

AttributeError: 'Sphere' object has no attribute 'postProcessAttributes'
```

5.3.5 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_DEPREC_INIT_CTOR_PY(klass,base,doc,attrs,,init,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 straightforwardsly 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++.

5.3.6 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 currently 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_Dem3DofGeom: 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 Dem3DofGeom classes (for different combination of shapes: Dem3DofGeom_SphereSphere, Dem3DofGeom_FacetSphere, Dem3DofGeom_WallSphere), but only provide a LawFunctor accepting `Dem3DofGeom`, 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. [5] 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.

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 [78]: Sphere().dispIndex, Facet().dispIndex, Wall().dispIndex
 ->  [78]: (1, 5, 7)
```

[5] 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.

```
Yade [79]: Shape().dispIndex                  # top-level indexable
 ->  [79]: -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 [80]: Dem3DofGeom().dispHierarchy()       # parent class of all other Dem3DofGeom_ classes
 ->  [80]: ['Dem3DofGeom', 'IGeom']

Yade [81]: Dem3DofGeom_SphereSphere().dispHierarchy(), Dem3DofGeom_FacetSphere().dispHierarchy(), Dem3DofGeom_Wa
 ->  [81]:
(['Dem3DofGeom_SphereSphere', 'Dem3DofGeom', 'IGeom'],
 ['Dem3DofGeom_FacetSphere', 'Dem3DofGeom', 'IGeom'],
 ['Dem3DofGeom_WallSphere', 'Dem3DofGeom', 'IGeom'])

Yade [82]: Dem3DofGeom_WallSphere().dispHierarchy(names=False)   # show numeric indices instead
 ->  [82]: [6, 3, -1]
```

Dispatchers can also be inspected, using the .dispMatrix() method:

```
Yade [83]: ig=IGeomDispatcher([
     ....:    Ig2_Sphere_Sphere_Dem3DofGeom(),
     ....:    Ig2_Facet_Sphere_Dem3DofGeom(),
     ....:    Ig2_Wall_Sphere_Dem3DofGeom()
     ....: ])

Yade [88]: ig.dispMatrix()
 ->  [88]:
{('Facet', 'Sphere'): 'Ig2_Facet_Sphere_Dem3DofGeom',
 ('Sphere', 'Facet'): 'Ig2_Facet_Sphere_Dem3DofGeom',
 ('Sphere', 'Sphere'): 'Ig2_Sphere_Sphere_Dem3DofGeom',
 ('Sphere', 'Wall'): 'Ig2_Wall_Sphere_Dem3DofGeom',
 ('Wall', 'Sphere'): 'Ig2_Wall_Sphere_Dem3DofGeom'}

Yade [89]: ig.dispMatrix(False)          # don't convert to class names
 ->  [89]:
{(1, 1): 'Ig2_Sphere_Sphere_Dem3DofGeom',
 (1, 5): 'Ig2_Facet_Sphere_Dem3DofGeom',
 (1, 7): 'Ig2_Wall_Sphere_Dem3DofGeom',
 (5, 1): 'Ig2_Facet_Sphere_Dem3DofGeom',
 (7, 1): 'Ig2_Wall_Sphere_Dem3DofGeom'}
```

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 [90]: ld=LawDispatcher([Law2_Dem3DofGeom_CpmPhys_Cpm()])

Yade [91]: ld.dispMatrix()
 ->  [91]: {('Dem3DofGeom', 'CpmPhys'): 'Law2_Dem3DofGeom_CpmPhys_Cpm'}

# see how the entry for Dem3DofGeom_SphereSphere will be filled after this request
Yade [93]: ld.dispFunctor(Dem3DofGeom_SphereSphere(),CpmPhys())
 ->  [93]: <Law2_Dem3DofGeom_CpmPhys_Cpm instance at 0x97dda68>

Yade [94]: ld.dispMatrix()
 ->  [94]:
{('Dem3DofGeom', 'CpmPhys'): 'Law2_Dem3DofGeom_CpmPhys_Cpm',
 ('Dem3DofGeom_SphereSphere', 'CpmPhys'): 'Law2_Dem3DofGeom_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 [96]: yade.system.childClasses('GlShapeFunctor')
 ->  [96]:
set(['Gl1_Box',
     'Gl1_ChainedCylinder',
     'Gl1_Cylinder',
     'Gl1_Facet',
     '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.

5.3.7 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 reponisbility 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.

5.3.8 Logging

Regardless of whether the *optional-libraries* log4cxx is used or not, yade provides logging macros. [6] If log4cxx is enabled, these macros internally operate on the local logger instance (named `logger`, but that is hidden for the user); if log4cxx is disabled, they send their arguments to standard error output (`cerr`).

Log messages are classified by their *severity*, which is one of `TRACE` (tracing variables), `DEBUG` (generally uninsteresting messages useful for debugging), `INFO` (information messages – only use sparingly), `WARN` (warning), `FATAL` (serious error, consider throwing an exception with description instead). Logging level determines which messages will be shown – by default, `INFO` and higher will be shown; if you run yade with `-v` or `-vv`, `DEBUG` and `TRACE` messages will be also enabled (with log4cxx).

Every class using logging should create logger using these 2 macros (they expand to nothing if log4cxx is not used):

`DECLARE_LOGGER;` in class declaration body (in the `.hpp` file); this declares static variable `logger`;

`CREATE_LOGGER(ClassName);` in the implementation file; it creates and initializes that static variable. The logger will be named `yade.ClassName`.

The logging macros are the following:

- `LOG_TRACE`, `LOG_DEBUG`, `LOG_INFO`, `LOG_WARN`, `LOG_ERROR`, `LOG_FATAL` (increasing severity); their argument is fed to the logger stream, hence can contain the `<<` operation:

[6] Because of (seemingly?) no upstream development of log4cxx and a few problems it has, Yade will very likely move to the hypothetical `boost::logging` library once it exists. The logging code will not have to be changed, however, as the log4cxx logic is hidden behind these macros.

```
LOG_WARN("Exceeded "<<maxSteps<<" steps in attempts to converge, the result returned will not be preci
```

Every log message is prepended filename, line number and function name; the final message that will appear will look like this:

```
237763 WARN  yade.ViscosityIterSolver /tmp/yade/trunk/extra/ViscosityIterSolver.cpp:316 newtonRaphsonS
```

The `237763 WARN yade.ViscosityIterSolver` (microseconds from start, severity, logger name) is added by log4cxx and is completely configurable, either programatically, or by using file `~/.yade-$SUFFIX/logging.conf`, which is loaded at startup, if present (FIXME: see more etc user's guide)

- special tracing macros `TRVAR1`, `TRVAR2`, … `TRVAR6`, which show both variable name and its value (there are several more macros defined inside `/lib/base/Logging.hpp`, but they are not generally in use):

```
TRVAR3(var1,var2,var3);
// will be expanded to:
LOG_TRACE("var1="<<var1<<"; var2="<<var2<<"; var3="<<var3);
```

Note: For performance reasons, optimized builds eliminate `LOG_TRACE` and `LOG_DEBUG` from the code at preprocessor level.

Note: Builds without log4cxx (even in debug mode) eliminate `LOG_TRACE` and `LOG_DEBUG`. As there is no way to enable/disable them selectively, the log amount would be huge.

Python provides rudimentary control for the logging system in `yade.log` module (FIXME: ref to docs):

```
Yade [97]: from yade import log

Yade [98]: log.setLevel('InsertionSortCollider',log.DEBUG)  # sets logging level of the yade.InsertionSortColli

Yade [99]: log.setLevel('',log.WARN)                         # sets logging level of all yade.* loggers (they in
```

As of now, there is no python interface for performing logging into log4cxx loggers themselves.

5.3.9 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 [100]: TriaxialTest(numberOfGrains=100).load()

Yade [101]: O.engines[0].label='firstEngine'   ## labeled engines will show by labels in the stats table

Yade [102]: import yade.timing;
Yade [103]: O.timingEnabled=True

Yade [104]: yade.timing.reset()                   ## not necessary if used for the first time

Yade [105]: O.run(50); O.wait()

Yade [106]: yade.timing.stats()
Name                                                    Count                 Time             Rel. time
-------------------------------------------------------------------------------------------------------
"firstEngine"                                        50                 246us                1.49%
InsertionSortCollider                                50                4663us               28.23%
InteractionLoop                                      50                5789us               35.04%
GlobalStiffnessTimeStepper                            2                 945us                5.72%
TriaxialCompressionEngine                            50                2426us               14.69%
TriaxialStateRecorder                                 3                 288us                1.74%
NewtonIntegrator                                     50                2161us               13.08%
TOTAL                                                                 16521us              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_Dem3DofGeom_CpmPhys_Cpm: public LawFunctor {
   /* … */
   YADE_CLASS_BASE_DOC_ATTRS_CTOR(Law2_Dem3DofGeom_CpmPhys_Cpm,LawFunctor,"docstring",
      /* attrs */,
      /* constructor */
      timingDeltas=shared_ptr<TimingDeltas>(new TimingDeltas);
   );
   // ...
};
```

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_Dem3DofGeom_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");
          }
```

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_Dem3DofGeom_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).

5.3.10 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.

5.4 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.

5.4.1 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.

5.4.2 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 [108]: O.bodies.append([Body(),Body(),Body()])
 ->  [108]: [0, 1, 2]

Yade [109]: O.bodies.erase(1)
 ->  [109]: True

Yade [110]: [b.id for b in O.bodies]
 ->  [110]: [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;
   /* … */
}
```

5.4.3 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;
   /* … */
}
```

5.4.4 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 write 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 agains 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 [111]: O.bodies.append(Body())
 ->  [111]: 1

Yade [112]: O.forces.addF(0,Vector3(1,2,3))

Yade [113]: O.forces.f(0)
 ->  [113]: Vector3(1,2,3)

Yade [114]: O.forces.f(100)
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
```

```
/build/buildd/yade-0.60.3/doc/sphinx/<ipython console> in <module>()

IndexError: Body id out of range.
```

5.4.5 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 jsut 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 combintation of IGeom and IPhys of the interaction. Again, it is an error if no functor capable of handling it is found.
6. LawDispatcher can decide that an interaction should be removed (such as if bodies get too far apart for non-cohesive laws; or in case of complete damage for damage models). This is done by calling

```
InteractionContainer::requestErase(id1,id2)
```

Such interaction will not be deleted immediately, but will be reset to potential state. At next step, the collider will call `InteractionContainer::erasePending`, which will only completely erase interactions the collider indicates; 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.

5.4.6 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.

5.5 Runtime structure

5.5.1 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 logging, 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.

5.5.2 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 [7] 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.

5.5.3 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 iteraction number is incremented by 1.

[7] It is understood that instantiating `Omega()` in python only instantiates the wrapper class, not the singleton itself.

Background execution

The engine loop is (normally) executed in background thread (handled by SimulationFlow class), leaving forground 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().

5.6 Python framework

5.6.1 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 [115]: f1=ForceEngine()

Yade [116]: f1.subscribedBodies=[0,4,5]

Yade [117]: f1.force=Vector3(0,-1,-2)
```

and

```
Yade [118]: f2=ForceEngine(subscribedBodies=[0,4,5],force=Vector3(0,-1,-2))

Yade [119]: print f1.dict()
{'force': Vector3(0,-1,-2), 'ids': [0, 4, 5], 'dead': False, 'label': ''}

Yade [120]: print f2.dict()
{'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 [121]: s=Sphere()

Yade [122]: s.name, s.__class__.__name__
WARN: Sphere.name is deprecated, use:
WARN: * Sphere.__class__.__name__ to get the class name (as string)
WARN: * isinstance(object,Sphere) to test whether object is of type Sphere.

 -> [122]: ('Sphere', 'Sphere')
```

5.6.2 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.

5.6.3 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.

5.6.4 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; [8]

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 "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

[8] 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 by hand, to be found in py/mathWrap/miniEigen.cpp; this module is imported at startup.

to `return_by_value` explicitly, using slighly more complicated `add_property` syntax, as explained at the page referenced.

5.7 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.

5.7.1 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.

5.7.2 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.

5.8 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 priporty than lower versions/revisions.

5.8.1 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).

5.8.2 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 6

Class reference (yade.wrapper module)

6.1 Bodies

6.1.1 Body

class `yade.wrapper.Body`(*inherits Serializable*)

A particle, basic element of simulation; interacts with other bodies.

`asphericial`(*=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`)

`clumpId`

Id of clump this body makes part of; invalid number if not part of clump; see Body::isStandalone, Body::isClump, Body::isClumpMember properties.

This property is not meant to be modified directly from Python, use O.bodies.appendClumped instead.

`dynamic`(*=true*)

Whether this body will be moved by forces. (In c++, use `Body::isDynamic`/`Body::setDynamic`)

`flags`(*=FLAG_DYNAMIC|FLAG_BOUNDED*)

Bits of various body-related flags. *Do not access directly.* In c++, use isDynamic/setDynamic, isBounded/setBounded. In python, use Body.dynamic and Body.bounded.

`groupMask`(*=1*)

Bitmask for determining interactions.

`id`(*=Body::ID_NONE*)

Unique id of this body.

`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.

mask
Shorthand for Body::groupMask

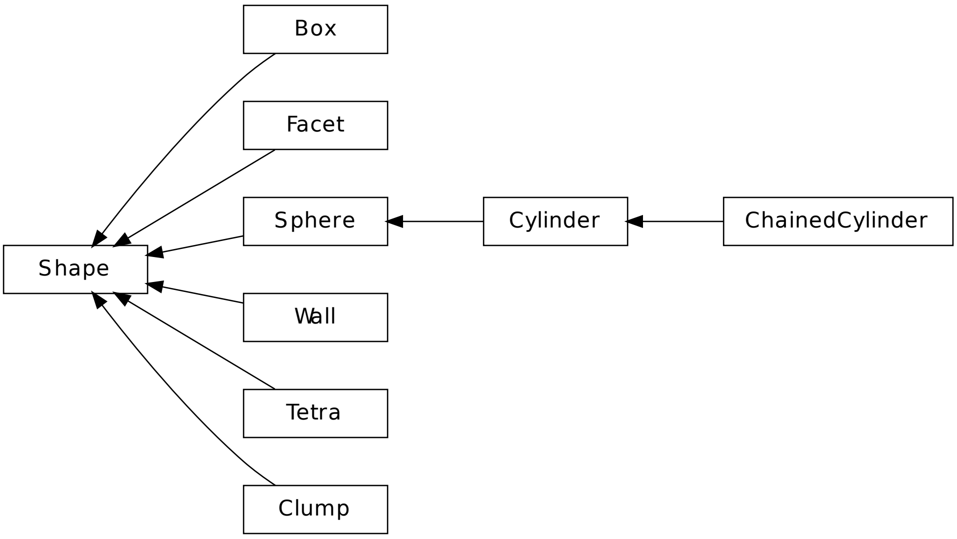
mat
Shorthand for Body::material

material(*=uninitalized*)
Material instance associated with this body.

shape(*=uninitalized*)
Geometrical Shape.

state(*=new State*)
Physical state.

6.1.2 Shape



class yade.wrapper.**Shape**(*inherits Serializable*)
Geometry of a body

color(*=Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

dispHierarchy($\left[(bool)names=True\right]$) → 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.

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.

`extents`(*=uninitalized*)

Half-size of the cuboid

class `yade.wrapper.ChainedCylinder`(*inherits Cylinder → Sphere → Shape → Serializable*)

Geometry of a deformable chained cylinder, using geometry MinkCylinder.

`chainedOrientation`(*=Quaternionr::Identity()*)

Deviation of node1 orientation from node-to-node vector

`initLength`(*=0*)

tensile-free length, used as reference for tensile strain

class `yade.wrapper.Clump`(*inherits Shape → Serializable*)

Rigid aggregate of bodies

class `yade.wrapper.Cylinder`(*inherits Sphere → Shape → Serializable*)

Geometry of a cylinder, as Minkowski sum of line and sphere.

`length`(*=NaN*)

Length [m]

`segment`(*=Vector3r::Zero()*)

Length vector

class `yade.wrapper.Facet`(*inherits Shape → Serializable*)

Facet (triangular particle) geometry.

`edgeAdjHalfAngle`(*=vector<Real>(3, 0)*)

half angle between normals of this facet and the adjacent facet [experimental]

`edgeAdjIds`(*=vector<Body::id_t>(3, Body::ID_NONE)*)

Facet id's that are adjacent to respective edges [experimental]

`vertices`(*=vector<Vector3r>(3)*)

Vertex positions in local coordinates.

class `yade.wrapper.Sphere`(*inherits Shape → Serializable*)

Geometry of spherical particle.

`radius`(*=NaN*)

Radius [m]

class `yade.wrapper.Tetra`(*inherits Shape → Serializable*)

Tetrahedron geometry.

`v`(*=std::vector<Vector3r>(4)*)

Tetrahedron vertices in global coordinate system.

class `yade.wrapper.Wall`(*inherits Shape → Serializable*)

Object representing infinite plane aligned with the coordinate system (axis-aligned wall).

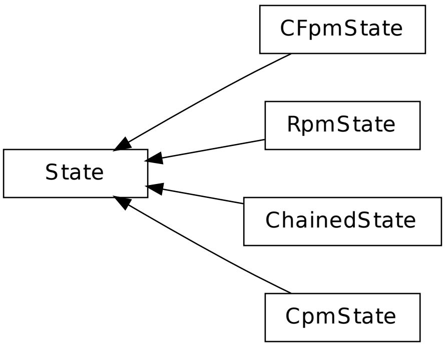
`axis`(*=0*)

Axis of the normal; can be 0,1,2 for +x, +y, +z respectively (Body's orientation is disregarded for walls)

`sense`(*=0*)

Which side of the wall interacts: -1 for negative only, 0 for both, +1 for positive only

6.1.3 State



class `yade.wrapper.State`(*inherits Serializable*)

State of a body (spatial configuration, internal variables).

`accel`(*=Vector3r::Zero()*)

Current acceleration.

`angAccel`(*=Vector3r::Zero()*)

Current angular acceleration

`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. List of any combination of 'x','y','z','rx','ry','rz'.

`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.

`inertia`(*=Vector3r::Zero()*)

Inertia of associated body, in local coordinate system.

`mass`(*=0*)

Mass of this body

`ori`

Current orientation.

`pos`

Current position.

`refOri`(*=Quaternionr::Identity()*)

Reference orientation

`refPos`(*=Vector3r::Zero()*)

Reference position

`se3`(*=Se3r(Vector3r::Zero(), Quaternionr::Identity())*)

Position and orientation as one object.

vel(*=Vector3r::Zero()*)
Current linear velocity.

class yade.wrapper.**CFpmState**(*inherits State → Serializable*)
CFpm state information about each body.

None of that is used for computation (at least not now), only for post-processing.

numBrokenCohesive(*=0*)
Number of broken cohesive links. [-]

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 :yref:rank<ChainedState::rank>‘ and :yref:chainNumber<ChainedState::chainNumber>‘.

addToChain(*(int)bodyId*) → None
Add body to current active chain

chainNumber(*=0*)
chain id

rank(*=0*)
rank in the chain

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.

epsPlBroken(*=0*)
Plastic strain on contacts already deleted (bogus values)

epsVolumetric(*=0*)
Volumetric strain around this body (unused for now)

normDmg(*=0*)
Average damage including already deleted contacts (it is really not damage, but 1-relResidualStrength now)

normEpsPl(*=0*)
Sum of plastic strains normalized by number of contacts (bogus values)

numBrokenCohesive(*=0*)
Number of (cohesive) contacts that damaged completely

numContacts(*=0*)
Number of contacts with this body

sigma(*=Vector3r::Zero()*)
Normal stresses on the particle

tau(*=Vector3r::Zero()*)
Shear stresses on the particle.

class yade.wrapper.**RpmState**(*inherits State → Serializable*)
State information about Rpm body.

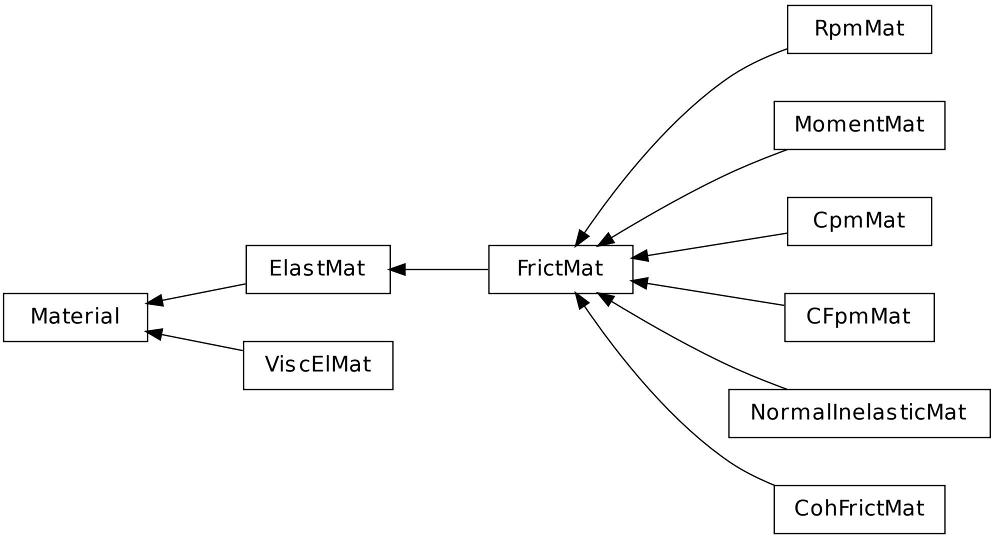
specimenMass(*=0*)
Indicates the mass of the whole stone, which owns the particle.

specimenMaxDiam(*=0*)
Indicates the maximal diametr of the specimen.

specimenNumber(*=0*)
The variable is used for particle size distribution analyze. Indicates, to which part of specimen belongs para of particles.

specimenVol(*=0*)
Indicates the mass of the whole stone, which owns the particle.

6.1.4 Material



class yade.wrapper.**Material**(*inherits Serializable*)

Material properties of a body.

density(*=1000*)

Density of the material [kg/m^3]

dispHierarchy($\left[\textit{(bool)names=True}\right]$) → 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).

class yade.wrapper.**CFpmMat**(*inherits FrictMat → ElastMat → Material → Serializable*)

cohesive frictional material, for use with other CFpm classes

type(*=0*)

Type of the particle. If particles of two different types interact, it will be with friction only (no cohesion).[-]

class yade.wrapper.**CohFrictMat**(*inherits FrictMat → ElastMat → Material → Serializable*)

isBroken(*=true*)

isCohesive(*=true*)

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.

normalCohesion(*=10000000*)

shearCohesion(*=10000000*)

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.

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.

G_over_E(*=NaN*)
Ratio of normal/shear stiffness at interaction level [-]

dmgRateExp(*=0*)
Exponent for normal viscosity function. [-]

dmgTau(*=-1, deactivated if negative*)
Characteristic time for normal viscosity. [s]

epsCrackOnset(*=NaN*)
Limit elastic strain [-]

isoPrestress(*=0*)
Isotropic prestress of the whole specimen. [Pa]

neverDamage(*=false*)
If true, no damage will occur (for testing only).

plRateExp(*=0*)
Exponent for visco-plasticity function. [-]

plTau(*=-1, deactivated if negative*)
Characteristic time for visco-plasticity. [s]

relDuctility(*=NaN*)
Relative ductility, for damage evolution law peak right-tangent. [-]

sigmaT(*=NaN*)
Initial cohesion [Pa]

class yade.wrapper.**ElastMat**(*inherits Material → Serializable*)
Purely elastic material.

poisson(*=.25*)
Poisson's ratio [-]

young(*=1e9*)
Young's modulus [Pa]

class yade.wrapper.**FrictMat**(*inherits ElastMat → Material → Serializable*)
Material with internal friction.

frictionAngle(*=.5*)
Internal friction angle (in radians) [-]

class yade.wrapper.**MomentMat**(*inherits FrictMat → ElastMat → Material → Serializable*)
Material for constitutive law of (Plassiard & al., 2009); see Law2_SCG_MomentPhys_CohesionlessMomentRotation for details.

Users can input eta (constant for plastic moment) to Spheres and Boxes. For more complicated cases, users can modify TriaxialStressController to use different eta values during isotropic compaction.

eta(*=0*)
(has to be stored in this class and not by ContactLaw, because users may want to change its values before/after isotropic compaction.)

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)

class yade.wrapper.**RpmMat**(*inherits FrictMat → ElastMat → Material → Serializable*)
Rock material, for use with other Rpm classes.

Brittleness(*=0*)
One of destruction parameters. [-] //(Needs to be reworked)

G_over_E(*=1*)
Ratio of normal/shear stiffness at interaction level. [-]

exampleNumber(*=0*)
Number of the specimen. This value is equal for all particles of one specimen. [-]

initCohesive(*=false*)
The flag shows, whether particles of this material can be cohesive. [-]

stressCompressMax(*=0*)
Maximal strength for compression. The main destruction parameter. [Pa] //(Needs to be reworked)

class yade.wrapper.**ViscElMat**(*inherits Material → Serializable*)
Material for simple viscoelastic model of contact.

Note: `Shop::getViscoelasticFromSpheresInteraction` (and utils.getViscoelasticFromSpheresInteraction in python) compute kn, cn, ks, cs from analytical solution of a pair spheres interaction problem.

cn(*=NaN*)
Normal viscous constant

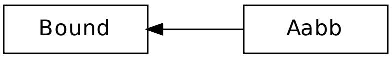
cs(*=NaN*)
Shear viscous constant

frictionAngle(*=NaN*)
Friction angle [rad]

kn(*=NaN*)
Normal elastic stiffness

ks(*=NaN*)
Shear elastic stiffness

6.1.5 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

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.

max(=*Vector3r(NaN, NaN, NaN)*)

Lower 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)

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.)

6.2 Interactions

6.2.1 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 the priod information if the geometric functor again makes it real. Good to know after few days of debugging that :-)

geom(=*uninitalized*)

Geometry part of the interaction.

id1(=*0*)

Id of the first body in this interaction.

id2(=*0*)

Id of the first body in this interaction.

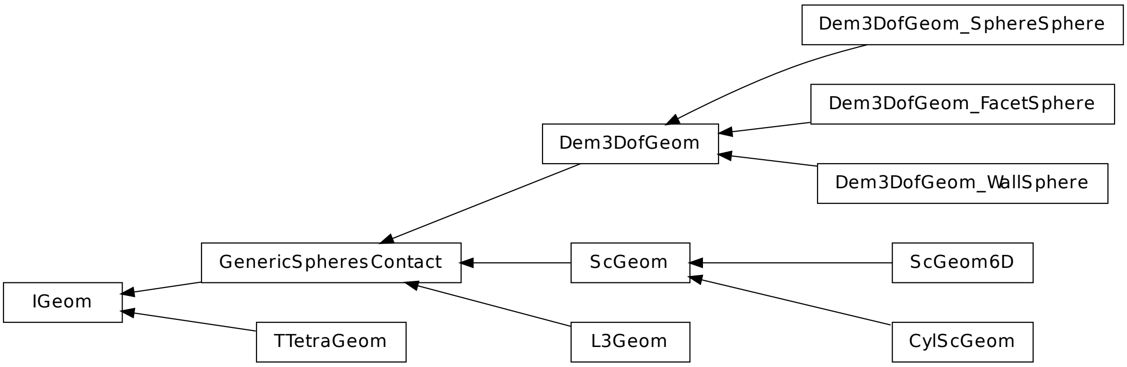
isReal

True if this interaction has both geom and phys; False otherwise.

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.

6.2.2 IGeom



class yade.wrapper.**IGeom**(*inherits Serializable*)
Geometrical configuration of interaction

dispHierarchy($\left[(bool)names=True\right]$) → 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.

class yade.wrapper.**CylScGeom**(*inherits ScGeom → GenericSpheresContact → IGeom → Serializable*)
Geometry of a cylinder-sphere contact.

end(*=Vector3r::Zero()*)
position of 2nd node *(auto-updated)*

id3(*=0*)
id of next chained cylinder *(auto-updated)*

onNode(*=false*)
contact on node?

relPos(*=0*)
position of the contact on the cylinder (0: node-, 1:node+) *(auto-updated)*

start(*=Vector3r::Zero()*)
position of 1st node *(auto-updated)*

class yade.wrapper.**Dem3DofGeom**(*inherits GenericSpheresContact → IGeom → Serializable*)
Abstract base class for representing contact geometry of 2 elements that has 3 degrees of freedom: normal (1 component) and shear (Vector3r, but in plane perpendicular to the normal).

displacementN() → float

displacementT() → Vector3

logCompression(*=false*)
make strain go to -∞ for length going to zero (false by default).

refLength(*=uninitalized*)
some length used to convert displacements to strains. *(auto-computed)*

se31(*=uninitalized*)
Copy of body #1 se3 (needed to compute torque from the contact, strains etc). *(auto-updated)*

se32(*=uninitalized*)
Copy of body #2 se3. *(auto-updated)*

slipToDisplacementTMax(*(float)arg2*) → float

slipToStrainTMax(*(float)arg2*) → float

strainN() → float

strainT() → Vector3

class yade.wrapper.**Dem3DofGeom_FacetSphere**(*inherits Dem3DofGeom → GenericSpheresContact → IGeom → Serializable*)
Class representing facet+sphere in contact which computes 3 degrees of freedom (normal and shear deformation).

cp1pt(*=uninitalized*)
Reference contact point on the facet in facet-local coords.

cp2rel(*=uninitalized*)
Orientation between +x and the reference contact point (on the sphere) in sphere-local coords

effR2(*=uninitalized*)
Effective radius of sphere

localFacetNormal(*=uninitalized*)
Unit normal of the facet plane in facet-local coordinates

class yade.wrapper.**Dem3DofGeom_SphereSphere**(*inherits Dem3DofGeom → GenericSpheresContact → IGeom → Serializable*)
Class representing 2 spheres in contact which computes 3 degrees of freedom (normal and shear deformation).

cp1rel(*=uninitalized*)
Sphere's #1 relative orientation of the contact point with regards to sphere-local +x axis (quasi-constant)

cp2rel(*=uninitalized*)
Same as cp1rel, but for sphere #2.

effR1(*=uninitalized*)
Effective radius of sphere #1; can be smaller/larger than refR1 (the actual radius), but quasi-constant throughout interaction life

effR2(*=uninitalized*)
Same as effR1, but for sphere #2.

class yade.wrapper.**Dem3DofGeom_WallSphere**(*inherits Dem3DofGeom → GenericSpheresContact → IGeom → Serializable*)
Representation of contact between wall and sphere, based on Dem3DofGeom.

cp1pt(*=uninitalized*)
initial contact point on the wall, relative to the current contact point

cp2rel(*=uninitalized*)
orientation between +x and the reference contact point (on the sphere) in sphere-local coords

effR2(*=uninitalized*)
effective radius of sphere

class yade.wrapper.**GenericSpheresContact**(*inherits IGeom → Serializable*)
Class uniting ScGeom and Dem3DofGeom, for the purposes of GlobalStiffnessTimeStepper. (It might be removed inthe future). Do not use this class directly.

`contactPoint`(*=uninitalized*)
: some reference point for the interaction (usually in the middle). *(auto-computed)*

`normal`(*=uninitalized*)
: Unit vector oriented along the interaction. *(auto-updated)*

`refR1`(*=uninitalized*)
: Reference radius of particle #1. *(auto-computed)*

`refR2`(*=uninitalized*)
: Reference radius of particle #2. *(auto-computed)*

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]

`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.

class `yade.wrapper.ScGeom`(*inherits GenericSpheresContact → IGeom → Serializable*)

Class representing geometry of two bodies in contact. The contact has 3 DOFs (normal and 2×shear) and uses incremental algorithm for updating shear. (For shear formulated in total displacements and rotations, see Dem3DofGeom and related classes).

We use symbols $\mathbf{x}$, $\mathbf{v}$, $\boldsymbol{\omega}$ 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 compute unit contact normal

$$\mathbf{n} = \frac{\mathbf{x}_2 - \mathbf{x}_1}{||\mathbf{x}_2 - \mathbf{x}_1||}$$

Relative velocity of spheres is then

$$\mathbf{v}_{12} = (\mathbf{v}_2 + \boldsymbol{\omega}_2 \times (-r_2\mathbf{n})) - (\mathbf{v}_1 + \boldsymbol{\omega}_1 \times (r_1\mathbf{n}))$$

and its shear component

$$\Delta\mathbf{v}_{12}^s = \mathbf{v}_{12} - (\mathbf{n} \cdot \mathbf{v}_{12})\mathbf{n}.$$

Tangential displacement increment over last step then reads

$$\mathbf{x}_{12}^s = \Delta t \mathbf{v}_{12}^s.$$

incidentVel(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3
Return incident velocity of the interaction.

penetrationDepth(*=NaN*)
Penetration distance of spheres (positive if overlapping)

shearInc(*=Vector3r::Zero()*)
Shear displacement increment in the last step

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).

currentContactOrientation(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)

initialContactOrientation(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)

initialOrientation1(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)

initialOrientation2(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)

orientationToContact1(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)

orientationToContact2(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)

twist(*=0*)
Elastic twist angle of the contact.

twistCreep(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)

class yade.wrapper.**TTetraGeom**(*inherits IGeom → Serializable*)
Geometry of interaction between 2 tetrahedra, including volumetric characteristics

contactPoint(*=uninitalized*)
Contact point (global coords)

equivalentCrossSection(*=NaN*)
Cross-section of the overlap (perpendicular to the axis of least inertia

equivalentPenetrationDepth(*=NaN*)
??

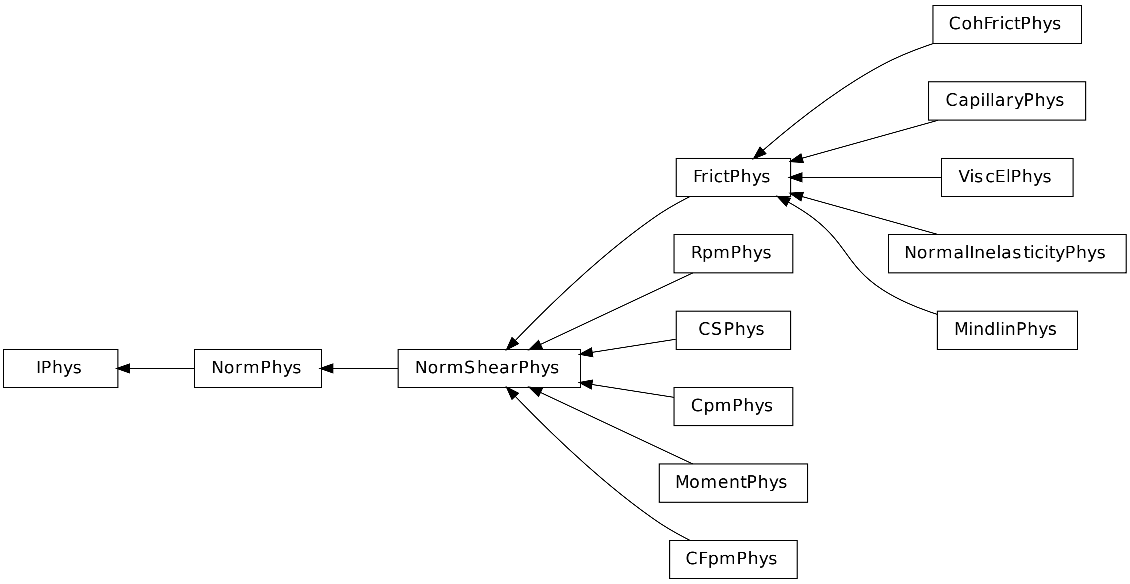
maxPenetrationDepthA(*=NaN*)
??

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]

6.2.3 IPhys



class `yade.wrapper.IPhys`(*inherits Serializable*)
Physical (material) properties of interaction.

`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.

class `yade.wrapper.CFpmPhys`(*inherits NormShearPhys → NormPhys → IPhys → Serializable*)
Representation of a single interaction of the CFpm type, storage for relevant parameters

`FnMax`(*=0*)
Defines the maximum admissible normal force in traction FnMax=tensileStrength*crossSection, with crossSection=pi*Rmin^2. [Pa]

`FsMax`(*=0*)
Defines the maximum admissible tangential force in shear FsMax=cohesion*FnMax, with crossSection=pi*Rmin^2. [Pa]

`cumulativeRotation`(*=0*)
Cumulated rotation... [-]

`frictionAngle`(*=0*)
defines Coulomb friction. [deg]

`initD`(*=0*)
equilibrium distance for particles. Computed as the initial interparticular distance when bonded particle interact. initD=0 for non cohesive interactions.

`initialOrientation1`(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Used for moment computation.

`initialOrientation2`(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Used for moment computation.

`isCohesive`(*=false*)
If false, particles interact in a frictional way. If true, particles are bonded regarding the given cohesion and tensileStrength.

kr(*=0*)
Defines the stiffness to compute the resistive moment in rotation. [-]

maxBend(*=0*)
Defines the maximum admissible resistive moment in rotation Mtmax=maxBend*Fn, maxBend=eta*meanRadius. [m]

moment_bending(*=Vector3r::Zero()*)
[N.m]

moment_twist(*=Vector3r::Zero()*)
[N.m]

prevNormal(*=Vector3r::Zero()*)
Normal to the contact at previous time step.

strengthSoftening(*=0*)
Defines the softening when Dtensile is reached to avoid explosion. Typically, when D > Dtensile, Fn=FnMax - (kn/strengthSoftening)*(Dtensile-D). [-]

tanFrictionAngle(*=0*)
Tangent of frictionAngle. [-]

class yade.wrapper.**CSPhys**(*inherits NormShearPhys → NormPhys → IPhys → Serializable*)
Physical properties for Cundall&Strack constitutive law, created by Ip2_2xFrictMat_CSPhys.

frictionAngle(*=NaN*)
Friction angle of the interaction. *(auto-computed)*

tanFrictionAngle(*=NaN*)
Precomputed tangent of CSPhys::frictionAngle. *(auto-computed)*

class yade.wrapper.**CapillaryPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
Physics (of interaction) for Law2_ScGeom_CapillaryPhys_Capillarity. Rk: deprecated -> needs some work to be conform with the new formalism!

CapillaryPressure(*=0.*)
Value of the capillary pressure Uc defines as Ugas-Uliquid

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)

Fcap(*=Vector3r::Zero()*)
Capillary Force produces by the presence of the meniscus

Vmeniscus(*=0.*)
Volume of the menicus

fusionNumber(*=0.*)
Indicates the number of meniscii that overlap with this one

meniscus(*=false*)
Presence of a meniscus if true

class yade.wrapper.**CohFrictPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)

cohesionBroken(*=true*)
is cohesion active? will be set false 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].

fragile(*=true*)
do cohesion disapear when contact strength is exceeded?

kr(*=0*)
rotational stiffness [N.m/rad]

momentRotationLaw(*=false*)
use bending/twisting moment at contacts. See CohFrictPhys::cohesionDisablesFriction for details.

moment_bending(*=Vector3r(0, 0, 0)*)
Bending moment

moment_twist(*=Vector3r(0, 0, 0)*)
Twist moment

normalAdhesion(*=0*)
tensile strength

shearAdhesion(*=0*)
cohesive part of the shear strength (a frictional term might be added depending on Law2_ScGeom_CohFrictPhys_CohesionMoment::always_use_moment_law)

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_Dem3DofGeom_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.

Fs
Magnitude of shear force

G(*=NaN*)
shear modulus [Pa]

crossSection(*=NaN*)
equivalent cross-section associated with this contact [m^2]

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 where the damage-evolution law tangent from the top (epsCrackOnset) touches the axis; since the softening law is exponential, this doesn't mean that the contact is fully damaged at this point, that happens only asymptotically

epsN
Current normal strain

epsNPl(*=0*)
normal plastic strain (initially zero)

epsPlSum(*=0*)
: cummulative shear plastic strain measure (scalar) on this contact

epsT(*=Vector3r::Zero()*)
: Total shear strain (either computed from increments with ScGeom or simple copied with Dem3DofGeom) *(auto-updated)*

epsTrans(*=0*)
: Transversal strain (perpendicular to the contact axis)

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(*=0*)
: Up to now maximum normal strain (semi-norm), non-decreasing in time.

neverDamage(*=false*)
: the damage evolution function will always return virgin state

omega
: Damage internal variable

plRateExp(*=0*)
: exponent in the rate-dependent viscoplasticity

plTau(*=-1*)
: characteristic time for viscoplasticity (if non-positive, no rate-dependence for shear)

relResidualStrength
: Relative residual strength

sigmaN
: Current normal stress

sigmaT
: Current shear stress

tanFrictionAngle(*=NaN*)
: tangens of internal friction angle [-]

undamagedCohesion(*=NaN*)
: virgin material cohesion [Pa]

class yade.wrapper.**FrictPhys**(*inherits NormShearPhys → NormPhys → IPhys → Serializable*)
: Interaction with friction

prevNormal(*=Vector3r::Zero()*)
: unit normal of the contact plane in previous step

tangensOfFrictionAngle(*=NaN*)
: tan of angle of friction

class yade.wrapper.**MindlinPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
: Representation of an interaction of the Hertz-Mindlin type.

adhesionForce(*=0.0*)
: Force of adhesion as predicted by DMT

betan(*=0.0*)
: Fraction of the viscous damping coefficient (normal direction) equal to $\frac{c_n}{C_{n,crit}}$.

betas(*=0.0*)
: Fraction of the viscous damping coefficient (shear direction) equal to $\frac{c_s}{C_{s,crit}}$.

isAdhesive(*=false*)
: bool to identify if the contact is adhesive, that is to say if the contact force is attractive

kno(*=0.0*)
Constant value in the formulation of the normal stiffness

kso(*=0.0*)
Constant value in the formulation of the tangential stiffness

normalViscous(*=Vector3r::Zero()*)
Normal viscous component

radius(*=NaN*)
Contact radius (only computed with Law2_ScGeom_MindlinPhys_Mindlin::calcEnergy)

shearElastic(*=Vector3r::Zero()*)
Total elastic shear force

shearViscous(*=Vector3r::Zero()*)
Shear viscous component

usElastic(*=Vector3r::Zero()*)
Total elastic shear displacement (only elastic part)

usTotal(*=Vector3r::Zero()*)
Total elastic shear displacement (elastic+plastic part)

class yade.wrapper.**MomentPhys**(*inherits NormShearPhys → NormPhys → IPhys → Serializable*)
Physical interaction properties for use with Law2_SCG_MomentPhys_CohesionlessMomentRotation, created by Ip2_MomentMat_MomentMat_MomentPhys.

Eta(*=0*)
??

cumulativeRotation(*=0*)
??

frictionAngle(*=0*)
Friction angle [rad]

initialOrientation1(*=Quaternionr::Identity()*)
??

initialOrientation2(*=Quaternionr::Identity()*)
??

kr(*=0*)
rolling stiffness

moment_bending(*=Vector3r::Zero()*)
??

moment_twist(*=Vector3r::Zero()*)
??

prevNormal(*=Vector3r::Zero()*)
Normal in the previous step.

shear(*=Vector3r::Zero()*)
??

tanFrictionAngle(*=0*)
Tangent of friction angle

class yade.wrapper.**NormPhys**(*inherits IPhys → Serializable*)
Abstract class for interactions that have normal stiffness.

kn(*=NaN*)
Normal stiffness

normalForce(*=Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

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.

ks(*=NaN*)
Shear stiffness

shearForce(*=Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

class `yade.wrapper.NormalInelasticityPhys`(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)

Physics (of interaction) for using Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity : with inelastic unloadings

forMaxMoment(*=1.0*)
parameter stored for each interaction, and allowing to compute the maximum value of the exchanged torque : TorqueMax= forMaxMoment * NormalForce

knLower(*=0.0*)
the stifness corresponding to a virgin load for example

kr(*=0.0*)
the rolling stiffness of the interaction

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

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

unMax(*=0.0*)
the maximum value of penetration depth of the history of this interaction

class `yade.wrapper.RpmPhys`(*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_Dem3DofGeom_CpmPhys_Cpm, that includes damage effects and chages of parameters inside CpmPhys

E(*=NaN*)
normal modulus (stiffness / crossSection) [Pa]

G(*=NaN*)
shear modulus [Pa]

crossSection(*=0*)
equivalent cross-section associated with this contact [m^2]

isCohesive(*=false*)
if not cohesive, interaction is deleted when distance is greater than lengthMaxTension or less than lengthMaxCompression.

lengthMaxCompression(*=0*)
Maximal penetration of particles during compression. If it is more, the interaction is deleted [m]

lengthMaxTension(*=0*)
Maximal distance between particles during tension. If it is more, the interaction is deleted [m]

tanFrictionAngle(*=NaN*)
tangens of internal friction angle [-]

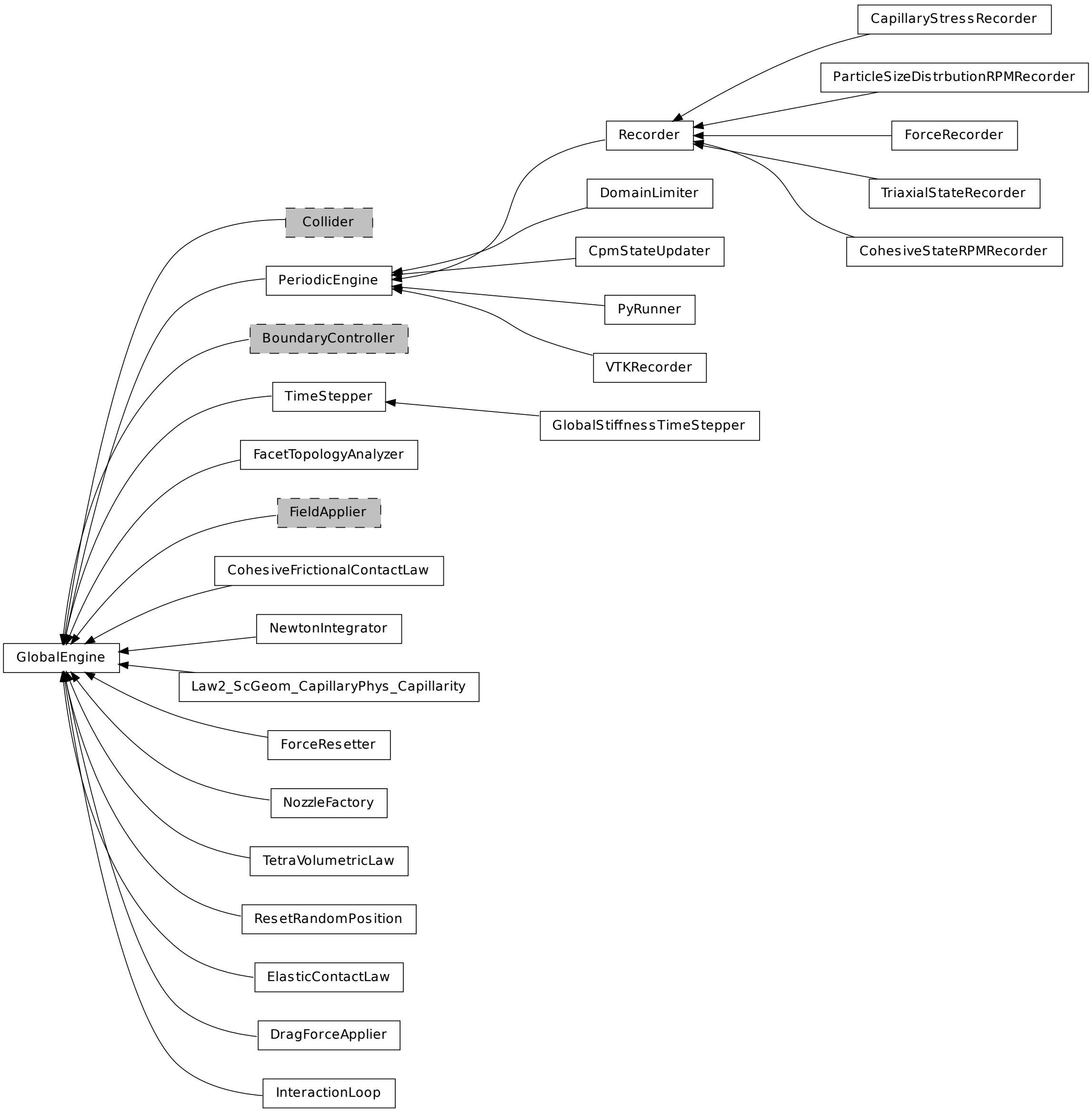
class yade.wrapper.**ViscElPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
IPhys created from ViscElMat, for use with Law2_ScGeom_ViscElPhys_Basic.

cn(*=NaN*)
Normal viscous constant

cs(*=NaN*)
Shear viscous constant

6.3 Global engines

6.3.1 GlobalEngine



class yade.wrapper.**GlobalEngine**(*inherits Engine → Serializable*)
Engine that will generally affect the whole simulation (contrary to PartialEngine).

class `yade.wrapper.CapillaryStressRecorder`(*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable*)

Records information from capillary meniscii on samples submitted to triaxial compressions. -> New formalism needs to be tested!!!

class `yade.wrapper.CohesiveFrictionalContactLaw`(*inherits GlobalEngine → Engine → Serializable*)

[DEPRECATED] Loop over interactions applying Law2_ScGeom_CohFrictPhys_CohesionMoment on all interactions.

Note: Use InteractionLoop and Law2_ScGeom_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_2xCohFrictMat_CohFrictPhys...

`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.

`twist_creep`(*=false*)

activate creep on the twisting moment, using CohesiveFrictionalContactLaw::creep_viscosity.

class `yade.wrapper.CohesiveStateRPMRecorder`(*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable*)

Store number of cohesive contacts in RPM model to file.

`numberCohesiveContacts`(*=0*)

Number of cohesive contacts found at last run. [-]

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.

`maxOmega`(*=NaN*)

Globally maximum damage parameter at last run.

class `yade.wrapper.DomainLimiter`(*inherits PeriodicEngine → GlobalEngine → Engine → Serializable*)

Delete particles that are out of axis-aligned box given by *lo* and *hi*.

`hi`(*=Vector3r(0, 0, 0)*)

Upper corner of the domain.

`lo`(*=Vector3r(0, 0, 0)*)

Lower corner of the domain.

`nDeleted`(*=0*)

Cummulative number of particles deleted.

class `yade.wrapper.DragForceApplier`(*inherits GlobalEngine → Engine → Serializable*)

Apply drag force on particles, decelerating them proportionally to their linear velocities. The applied force reads

$$F_d = -\frac{\mathbf{v}}{|\mathbf{v}|}\frac{1}{2}\rho|\mathbf{v}|^2 C_d A$$

where ρ is the medium density (density), v 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.

density(*=0*)
Density of the medium.

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.

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)

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)*

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

class **yade.wrapper.ForceRecorder**(*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable*)
Engine saves the resulting force affecting to Subscribed bodies. For instance, can be useful for defining the forces, which affect to _buldozer_ during its work.

ids(*=uninitalized*)
Lists of bodies whose state will be measured

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.

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

defaultDt(*=1*)
used as default AND as max value of the timestep

previousDt(*=1*)
last computed dt *(auto-updated)*

timestepSafetyCoefficient(*=0.8*)
safety factor between the minimum eigen-period and the final assigned dt (less than 1))

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. (NOT YET DONE: Optionally, list of IntrCallbacks can be provided as fourth argument.)

callbacks(*=uninitalized*)
Callbacks which will be called for every Interaction, if activated.

geomDispatcher(*=new IGeomDispatcher*)
IGeomDispatcher object that is used for dispatch.

lawDispatcher(*=new LawDispatcher*)
LawDispatcher object used for dispatch.

physDispatcher(*=new IPhysDispatcher*)
IPhysDispatcher object used for dispatch.

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).

refs:

1.in french [Scholtes2009d] (lot of documentation)

2.in english [Scholtes2009b] (less documentation), pg. 64-75.

The law needs ascii files M(r=i) with i=R1/R2 to work (see https://yade-dem.org/index.php/CapillaryTriaxialTest). These ASCII files contain a set of results from the resolution of the Laplace-Young equation for different configurations of the interacting geometry.

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).

CapillaryPressure(*=0.*)
Value of the capillary pressure Uc defines as Uc=Ugas-Uliquid

binaryFusion(*=true*)
If true, capillary forces are set to zero as soon as, at least, 1 overlap (menisci fusion) is detected

fusionDetection(*=false*)
If true potential menisci overlaps are checked

class yade.wrapper.**NewtonIntegrator**(*inherits GlobalEngine → Engine → Serializable*)
Engine integrating newtonian motion equations.

callbacks(*=uninitalized*)
List (std::vector in c++) of BodyCallbacks which will be called for each body as it is being processed.

damping(*=0.2*)
damping coefficient for Cundall's non viscous damping (see [Chareyre2005]) [-]

exactAsphericalRot(*=true*)
Enable more exact body rotation integrator for aspherical bodies *only*, using formulation from [Allen1989], pg. 89.

maxVelocitySq(*=NaN*)
store square of max. velocity, for informative purposes; computed again at every step. *(auto-updated)*

prevVelGrad(*=Matrix3r::Zero()*)
Store previous velocity gradient (Cell::velGrad) to track acceleration. *(auto-updated)*

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.**NozzleFactory**(*inherits GlobalEngine → Engine → Serializable*)
Engine for spitting spheres based on mass flow rate, particle size distribution etc. The area where spehres are generated should be circular, given by radius, center and normal. For now, axis-aligned cube-shape corresponding is used instead, centered at *center* and with size $\frac{2\sqrt{3}}{3}$. 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.

A sample script for this engine is in scripts/shots.py.

center(*=Vector3r(NaN, NaN, NaN)*)
Center of the nozzle

goalMass(*=0*)
Total mass that should be attained at the end of the current step. *(auto-updated)*

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.

maxParticles(*=100*)
The number of particles at which to stop generating new ones (regardless of massFlowRate

normal(*=Vector3r(NaN, NaN, NaN)*)
Spitting direction, i.e. normal of the circle where spheres are generated.

numParticles(*=0*)
Cummulative number of particles produces so far *(auto-updated)*

rMax(*=NaN*)
Maximum radius of generated spheres (uniform distribution)

rMin(*=NaN*)
Minimum radius of generated spheres (uniform distribution)

radius(*=NaN*)
Radius of the nozzle

totalMass(*=0*)
Mass of spheres that was produced so far. *(auto-updated)*

vAngle(*=NaN*)
Maximum angle by which the initial sphere velocity deviates from the nozzle normal.

vMax(*=NaN*)
Maximum velocity norm of generated spheres (uniform distribution)

vMin(*=NaN*)
Minimum velocity norm of generated spheres (uniform distribution)

class yade.wrapper.**ParticleSizeDistrbutionRPMRecorder**(*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable*)
Store number of PSD in RPM model to file.

numberCohesiveContacts(*=0*)
Number of cohesive contacts found at last run. [-]

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 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.

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)

nDo(*=-1, deactivated*)
Limit number of executions by this number (deactivated if negative)

nDone(*=0*)
Track number of executions (cummulative) *(auto-updated)*.

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)

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.

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)

file(*=uninitalized*)
Name of file to save to; must not be empty.

truncate(*=false*)
Whether to delete current file contents, if any, when opening (false by default)

class yade.wrapper.**ResetRandomPosition**(*inherits 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.

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.

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.

normal(*=Vector3r(0, 1, 0)*)
??

point(*=Vector3r::Zero()*)
??

subscribedBodies(*=uninitalized*)
Affected bodies.

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.

volumeSection(*=false, define factory by facets.*)
Create new spheres inside factory volume rather than on its surface.

class yade.wrapper.**TetraVolumetricLaw**(*inherits GlobalEngine → Engine → Serializable*)
Calculate physical response of 2 tetrahedra in interaction, based on penetration configuration given by TTetraGeom.

class yade.wrapper.**TimeStepper**(*inherits GlobalEngine → Engine → Serializable*)
Engine defining time-step (fundamental class)

active(*=true*)
is the engine active?

timeStepUpdateInterval(*=1*)
dt update interval

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).

porosity(*=1*)
porosity of the packing [-]

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.

`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].

`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.

`mask`(*=0*)

If mask defined, only bodies with corresponding groupMask will be exported. If 0, all bodies will be exported.

`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).

`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.

`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.

`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.

Specific recorders

The following should only be activated in appropriate cases, otherwise crashes can occur due to violation of type presuppositions.

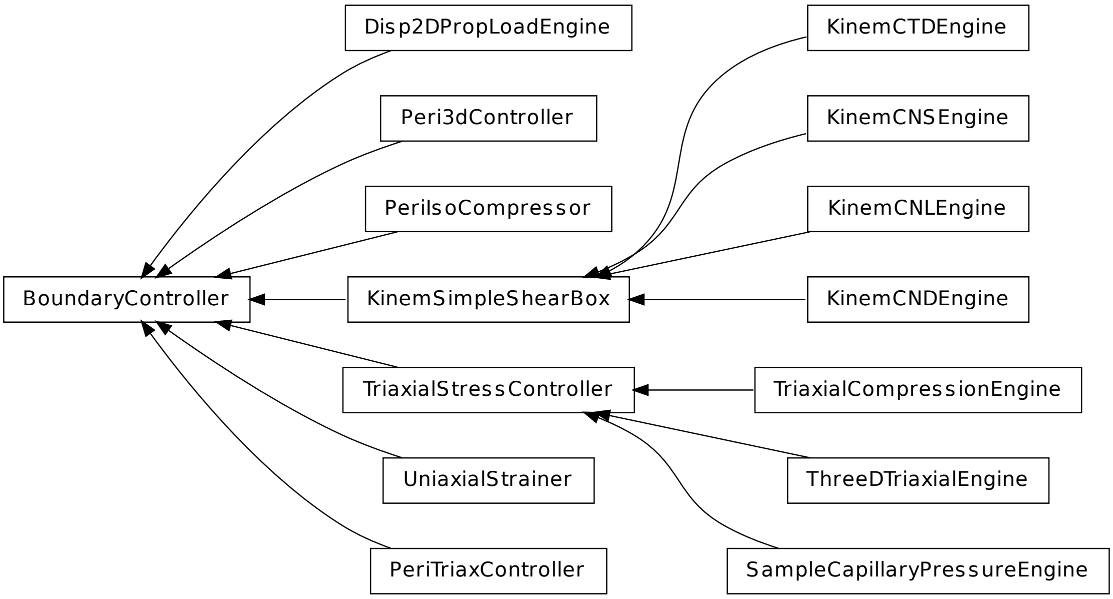
cpm Saves data pertaining to the concrete model: `cpmDamage` (normalized residual strength averaged on particle), `cpmSigma` (stress on particle, normal components), `cpmTau` (shear components of stress on particle), `cpmSigmaM` (mean stress around particle); `intr` is activated automatically by `cpm`

rpm Saves data pertaining to the rock particle model: `rpmSpecNum` shows different pieces of separated stones, only ids. `rpmSpecMass` shows masses of separated stones.

skipFacetIntr(*=true*)
Skip interactions with facets, when saving interactions

skipNondynamic(*=false*)
Skip non-dynamic spheres (but not facets).

6.3.2 BoundaryController



class `yade.wrapper.BoundaryController`(*inherits GlobalEngine → Engine → Serializable*)
Base for engines controlling boundary conditions of simulations. Not to be used directly.

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

`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

`nbre_iter`(=*0*)
: the number of iterations of loading to perform

`theta`(=*0.0*)
: the angle, in a (gamma,h=-u) plane from the gamma - axis to the perturbation vector (trigo wise) [degrees]

`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.

`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]

`shearSpeed`(=*0.0*)
: the speed at which the shear is performed : speed of the upper plate [m/s]

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.

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]

shearSpeed(*=0.0*)
the speed at wich the shearing is performed : speed of the upper plate [m/s]

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

KnC(*=10.0e6*)
the normal rigidity chosen by the user [MPa/mm] - the conversion in Pa/m will be made

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]

shearSpeed(*=0.0*)
: the speed at wich the shearing is performed : speed of the upper plate [m/s]

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*

compSpeed(*=0.0*)
: (vertical) speed of the upper box : >0 for real compression, <0 for unloading [m/s]

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]

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, not to be changed by the user.

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]. 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). 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

max_vel(*=1.0*)
: to limit the speed of the vertical displacements done to control σ (CNL or CNS cases) [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). Not to be changed by the user.

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 $[m]$. 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 :yref:"Cell'. goal are the goal values, while stressMask determines which components prescribe stress and which prescribe strain.

If the strain is prescribed, appropeiate 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

$$d\sigma_{i+1} \begin{cases} > 0 \rightarrow \dot{\varepsilon}_{i+1} = \dot{\varepsilon}_i - \max(\text{abs}(\dot{\boldsymbol{\varepsilon}}_i)) \cdot \text{mod} \\ < 0 \rightarrow \dot{\varepsilon}_{i+1} = \dot{\varepsilon}_i + \max(\text{abs}(\dot{\boldsymbol{\varepsilon}}_i)) \cdot \text{mod} \end{cases}$$

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 $\mathcal{T} = \mathbf{UP}$, 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 $\mathbf{P}$

$$\mathbf{P}_{i+1} = \mathbf{P} + \mathbf{U}^{\mathsf{T}} dt \cdot \dot{\boldsymbol{\varepsilon}} \mathbf{U}$$

The new value of trsf is computed at $\mathbf{T}_{i+1} = \mathbf{UP}_{i+1}$. From current and next trsf the cell's velocity gradient velGrad is computed (according to its definition) as

$$\mathbf{V} = (\mathbf{T}_{i+1}\mathbf{T}^{-1} - \mathbf{I})/dt$$

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 scripts/test/peri3dController_example1 and scripts/test/peri3dController_triaxialCompression explain usage and inputs of Peri3dController, scripts/test/peri3dController_shear is an example of using shear components and also simulation on rotatd cell.

doneHook(*=uninitalized*)
: Python command (as string) to run when nSteps is achieved. If empty, the engine will be set dead.

goal(*=Vector6r::Zero()*)
: Goal state; only the upper triangular matrix is considered; each component is either prescribed stress or strain, depending on stressMask.

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.

poissonEstimation(*=.25*)
: Estimation of macroscopic Poisson's ratio, used used for the first simulation step

progress(*=0.*)
: Actual progress of the simulation with Controller.

strain(*=Vector6r::Zero()*)
: Current strain (deformation) vector (ε_x,ε_y,ε_z,γ_{yz},γ_{zx},γ_{xy}) *(auto-updated)*.

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|.

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).

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

doneHook(*=""*)
Python command to be run when reaching the last specified stress

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

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,

sigma
Current stress value

state(*=0*)
Where are we at in the stress series

stresses(*=uninitalized*)
Stresses that should be reached, one after another

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)*

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 (PeriTriaxController->mass).

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.

mass(*=NaN*)
mass of the cell (user set)

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.

prevGrow(*=Vector3r::Zero()*)
previous cell grow

relStressTol(*=3e-5*)
Relative stress tolerance

reversedForces(*=false*)
For broken constitutive laws, normalForce and shearForce on interactions are in the reverse sense. see bugreport

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)

class yade.wrapper.**SampleCapillaryPressureEngine**(*inherits TriaxialStressController → BoundaryController → GlobalEngine → Engine → Serializable*)
Rk: this engine has to be tested withthe new formalism. It produces the isotropic compaction of an assembly and allows one to controlled the capillary pressure inside (uses Law2_ScGeom_CapillaryPhys_Capillarity).

Pressure(*=0*)
Value of the capillary pressure Uc=Ugas-Uliquid (see Law2_ScGeom_CapillaryPhys_Capillarity). [Pa]

`PressureVariation`(*=0*)
Variation of the capillary pressure (each iteration). [Pa]

`SigmaPrecision`(*=0.001*)
tolerance in terms of mean stress to consider the packing as stable

`StabilityCriterion`(*=0.01*)
tolerance in terms of :yref:'TriaxialCompressionEngine::UnbalancedForce' to consider the packing as stable

`UnbalancedForce`(*=1*)
mean resultant forces divided by mean contact force

`binaryFusion`(*=1*)
If yes, capillary force are set to 0 when, at least, 1 overlap is detected for a meniscus. If no, capillary force is divided by the number of overlaps.

`fusionDetection`(*=1*)
Is the detection of menisci overlapping activated?

`pressureVariationActivated`(*=1*)
Is the capillary pressure varying?

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.

`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

`currentStrainRate1`(*=0*)
current strain rate in direction 1 - converging to :yref:'ThreeDTriaxialEngine::strainRate1' (./s)

`currentStrainRate2`(*=0*)
current strain rate in direction 2 - converging to :yref:'ThreeDTriaxialEngine::strainRate2' (./s)

`currentStrainRate3`(*=0*)
current strain rate in direction 3 - converging to :yref:'ThreeDTriaxialEngine::strainRate3' (./s)

`frictionAngleDegree`(*=-1*)
Value of friction used in the simulation if (updateFrictionAngle)

`setContactProperties`(*(float)arg2*) → None
Assign a new friction angle (degrees) to dynamic bodies and relative interactions

`strainRate1`(*=0*)
target strain rate in direction 1 (./s)

`strainRate2`(*=0*)
target strain rate in direction 2 (./s)

`strainRate3`(*=0*)
target strain rate in direction 3 (./s)

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

updateFrictionAngle(*=false*)
Switch to activate the update of the intergranular frictionto the value :yref:'ThreeDTriaxialEngine::frictionAngleDegree

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: This engine handles many different manipulations, including some save/reload with attributes modified manually in between. Please don't modify the algorithms, even if they look strange (especially test sequences) without notifying me and getting explicit approval. A typical situation is somebody generates a sample with !autoCompressionActivation and run : he wants a saved simulation at the end. He then reload the saved state, modify some parameters, set autoCompressionActivation=true, and run. He should get the compression test done.

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(*=true*)
Stop the simulation when the sample reach STATE_LIMBO, or keep running

autoUnload(*=true*)
Auto-switch from isotropic compaction to unloading

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)

epsilonMax(*=0.5*)
Value of axial deformation for which the loading must stop

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

maxStress(*=0*)
Max value of stress during the simulation (for post-processing)

noFiles(*=false*)
If true, no files will be generated (*.xml, *.spheres,...)

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)

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

sigmaLateralConfinement(*=1*)
Prescribed confining pressure in the deviatoric loading; might be different from TriaxialCompressionEngine::sigmaIsoCompaction

strainRate(*=0*)
target strain rate (./s)

testEquilibriumInterval(*=20*)
interval of checks for transition between phases, higher than 1 saves computation time.

translationAxis(*=TriaxialStressController::normal[wall_bottom_id]*)
compression axis

uniaxialEpsilonCurr(*=1*)
Current value of axial deformation during confined loading (is reference to strain[1])

class yade.wrapper.**TriaxialStressController**(*inherits BoundaryController → GlobalEngine → Engine → Serializable*)
An engine maintaining constant stresses on some boundaries of a parallepipedic packing.

boxVolume
Total packing volume.

computeStressStrainInterval(*=10*)

depth(*=0*)

depth0(*=0*)

externalWork(*=0*)
Energy provided by boundaries.

finalMaxMultiplier(*=1.00001*)
max multiplier of diameters during internal compaction (secondary precise adjustment - TriaxialStressController::maxMultiplier is used in the initial stage)

height(*=0*)

height0(*=0*)

internalCompaction(*=true*)
Switch between 'external' (walls) and 'internal' (growth of particles) compaction.

isAxisymetric(*=true*)
if true, sigma_iso is assigned to sigma1, 2 and 3 (applies at each iteration and overrides user-set values of s1,2,3)

maxMultiplier(*=1.001*)
max multiplier of diameters during internal compaction (initial fast increase - TriaxialStressController::finalMaxMultiplier is used in a second stage)

max_vel(*=0.001*)
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.

porosity
Porosity of the packing.

previousMultiplier(*=1*)

previousStress(*=0*)

radiusControlInterval(*=10*)

sigma1(*=0*)
prescribed stress on axis 1 (see TriaxialStressController::isAxisymetric)

sigma2(*=0*)
prescribed stress on axis 2 (see TriaxialStressController::isAxisymetric)

sigma3(*=0*)
prescribed stress on axis 3 (see TriaxialStressController::isAxisymetric)

sigma_iso(*=0*)
prescribed confining stress (see TriaxialStressController::isAxisymetric)

spheresVolume
Total volume pf spheres.

stiffnessUpdateInterval(*=10*)
target strain rate (./s)

strain
Current strain (logarithmic).

stress(*(int)id*) → Vector3
Return the mean stress vector acting on boundary 'id', with 'id' between 0 and 5.

thickness(*=-1*)

volumetricStrain(*=0*)
Volumetric strain (see TriaxialStressController::strain).

wallDamping(*=0.25*)
wallDamping coefficient - wallDamping=0 implies a (theoretical) perfect control, wallDamping=1 means no movement

wall_back_activated(*=true*)

wall_back_id(*=0*)
id of boundary ; coordinate 2-

wall_bottom_activated(*=true*)

wall_bottom_id(*=0*)
id of boundary ; coordinate 1-

wall_front_activated(*=true*)

wall_front_id(*=0*)
id of boundary ; coordinate 2+

wall_left_activated(*=true*)

wall_left_id(*=0*)
id of boundary ; coordinate 0-

wall_right_activated(*=true*)

wall_right_id(*=0*)
id of boundary ; coordinate 0+

wall_top_activated(*=true*)

wall_top_id(*=0*)
id of boundary ; coordinate 1+

width(*=0*)

width0(*=0*)

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 of 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)*

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]

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).

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?

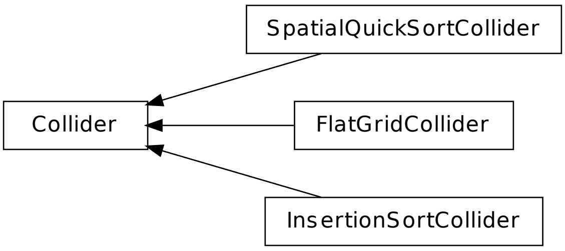
stopStrain(*=NaN*)
Strain at which we will pause simulation; inactive (nan) by default; must be reached from below (in absolute value)

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.

6.3.3 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.

`boundDispatcher`(*=new BoundDispatcher*)

BoundDispatcher object that is used for creating bounds on collider's request as necessary.

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:** Body::bound objects 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.

`step`(*=0*)

Step in the grid (cell size)

`verletDist`(*=0*)

Length by which enlarge space occupied by each particle; avoids running collision detection at every step.

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 one axis (sortAxis), 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 it it does and gets in interaction.
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 velocities 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 `sweepLength` (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 you additionally set `nBins` to >=1, not all particles will have their bound enlarged by `sweepLength`; instead, they will be put to bins (in the statistical sense) based on magnitude of their velocity; `sweepLength` will only be used for particles in the fastest bin, whereas only proportionally smaller length will be used for slower particles; The coefficient between bin's velocities is given by `binCoeff`.

`binCoeff`(*=5*)
Coefficient of bins for velocities, i.e. if `binCoeff==5`, successive bins have 5 × smaller velocity peak than the previous one. (Passed to VelocityBins)

`binOverlap`(*=0.8*)
Relative bins hysteresis, to avoid moving body back and forth if its velocity is around the border value. (Passed to VelocityBins)

`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 flowing items:

- coordinate (float)
- body id (int), but negated for negative bounds
- period numer (int), if the collider is in the periodic regime.

`fastestBodyMaxDist`(*=-1*)
Maximum displacement of the fastest body since last run; if >= sweepLength, we could get out of bboxes and will trigger full run. DEPRECATED, was only used without bins. *(auto-updated)*

`histInterval`(*=100*)
How often to show velocity bins graphically, if debug logging is enabled for VelocityBins.

`maxRefRelStep`(*=.3*)
(Passed to VelocityBins)

`nBins`(*=0*)
Number of velocity bins for striding. If <=0, bin-less strigin is used (this is however DEPRECATED).

`numReinit`(*=0*)
Cummulative number of bound array re-initialization.

`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)*

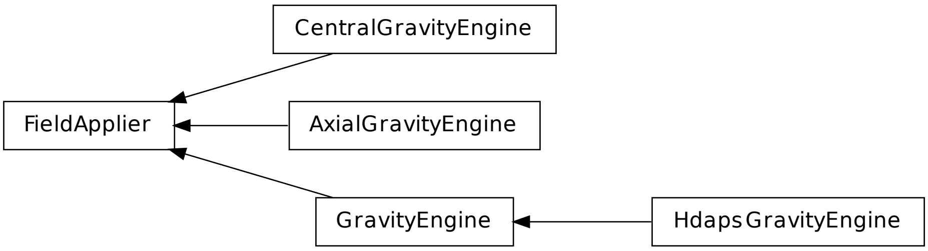
sweepFactor(*=1.05*)
Overestimation factor for the sweep velocity; must be >=1.0. Has no influence on sweepLength, only on the computed stride. [DEPRECATED, is used only when bins are not used].

sweepLength(*=-1, Stride deactivated*)
Length by which to enlarge particle bounds, to avoid running collider at every step. Stride disabled if negative.

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.

6.3.4 FieldApplier



class yade.wrapper.FieldApplier(*inherits GlobalEngine → Engine → Serializable*)
Base for engines applying force files on particles. Not to be used directly.

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.

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.

reciprocal(*=false*)
If true, acceleration will be applied on the central body as well.

class yade.wrapper.GravityEngine(*inherits FieldApplier → GlobalEngine → Engine → Serializable*)
Engine applying constant acceleration to all bodies.

gravity(*=Vector3r::Zero()*)
: Acceleration [kgms ^{2}]

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*.

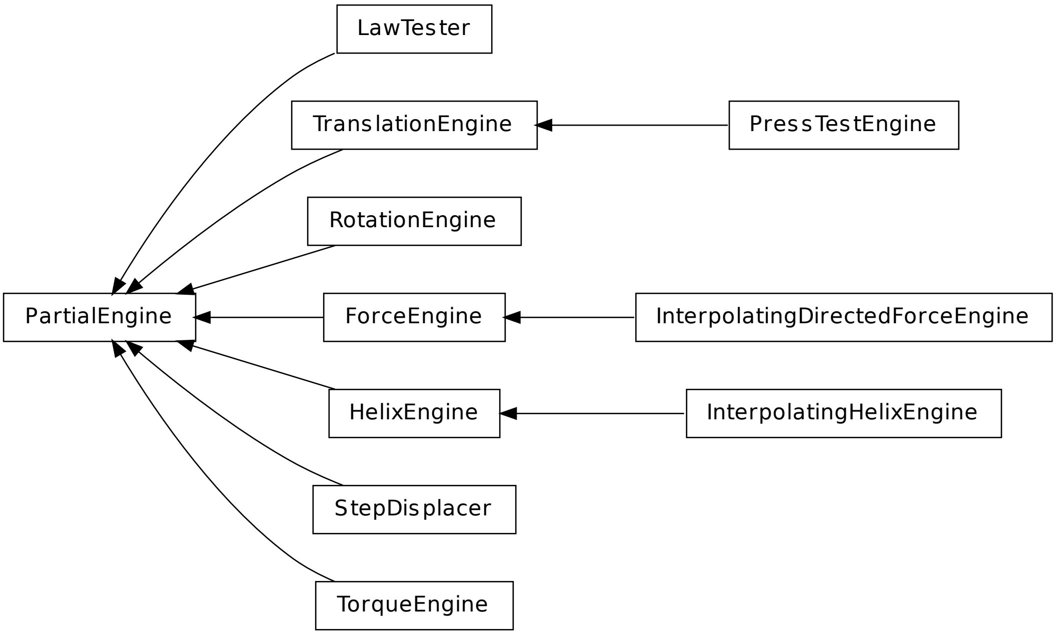
hdapsDir(*="/sys/devices/platform/hdaps"*)
: Hdaps directory; contains `position` (with accelerometer readings) and `calibration` (zero acceleration).

msecUpdate(*=50*)
: How often to update the reading.

updateThreshold(*=4*)
: Minimum difference of reading from the file before updating gravity, to avoid jitter.

zeroGravity(*=Vector3r(0, 0, -1)*)
: Gravity if the accelerometer is in flat (zero) position.

6.4 Partial engines



class yade.wrapper.**PartialEngine**(*inherits Engine → Serializable*)
: Engine affecting only particular bodies in the simulation, defined by *ids*.

ids(*=uninitalized*)
: Ids of bodies affected by this PartialEngine.

class `yade.wrapper.`**`ForceEngine`**(*inherits PartialEngine → Engine → Serializable*)

Apply contact force on some particles at each step.

`force`(*=Vector3r::Zero()*)

Force to apply.

class `yade.wrapper.`**`HelixEngine`**(*inherits 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]

`axis`(*=Vector3r::UnitX()*)

Axis of translation and rotation; will be normalized by the engine.

`axisPt`(*=Vector3r::Zero()*)

A point on the axis, to position it in space properly.

`linearVelocity`(*=0*)

Linear velocity [m/s]

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.

`direction`(*=Vector3r::UnitX()*)

Contact force direction (normalized automatically)

`magnitudes`(*=uninitalized*)

Force magnitudes readings [N]

`times`(*=uninitalized*)

Time readings [s]

`wrap`(*=false*)

wrap to the beginning of the sequence if beyond the last time point

class `yade.wrapper.`**`InterpolatingHelixEngine`**(*inherits HelixEngine → 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).

`angularVelocities`(*=uninitalized*)

List of angular velocities; manadatorily of same length as times. [rad/s]

`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]

`wrap`(*=false*)

Wrap t if t>times_n, i.e. t_wrapped=t-N*(times_n-times_0)

class `yade.wrapper.LawTester`(*inherits PartialEngine → Engine → Serializable*)

Prescribe and apply deformations of an interaction in terms of normal and shear displacements. See scripts/test/law-test.py.

`axX`(*=uninitalized*)
: Local x-axis in global coordinates (normal of the contact) *(auto-updated)*

`axY`(*=uninitalized*)
: Local y-axis in global coordinates; perpendicular to axX; initialized arbitrarily, but tracked to be consistent. *(auto-updated)*

`axZ`(*=uninitalized*)
: Local z-axis in global coordinates; computed from axX and axY. *(auto-updated)*

`contPt`(*=Vector3r::Zero()*)
: Contact point (for rendering only)

`displIsRel`(*=true*)
: Whether displacement values in *path* 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.

`forceControl`(*=Vector3i::Zero()*)
: Select which components of path (non-zero value) have force (stress) rather than displacement (strain) meaning.

`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 ⟨0,1⟩ determining on which particle are displacements applied (0 for id1, 1 for id2); intermediate values will apply respective part to each of them.

`path`(*=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.

`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.

`phiPrev`(*=Vector3r::Zero()*)
: Rotation value reached in the previous step.

`ptGeom`(*=Vector3r::Zero()*)
: Current displacement, as computed by the geometry functor

`ptOurs`(*=Vector3r::Zero()*)
: Current displacement, computed by ourselves from applied increments; should correspond to ptGeom.

`refLength`(*=0*)
: Reference contact length, for rendering only.

`renderLength`(*=0*)
: Characteristic length for the purposes of rendering, set equal to the smaller radius.

`rotGeom`(*=Vector3r::Zero()*)
: Current rotation, as computed by the geometry functor

`rotOurs`(*=Vector3r::Zero()*)
: Current rotation, computed by ourselves from applied increments; should correspond to rotGeom.

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.

rotTot(*=Vector3r::Zero()*)
Current rotation in global coordinates.

rotWeight(*=1*)
Float ⟨0,1⟩ determining whether shear displacement is applied as rotation or displacement on arc (0 is displacemetn-only, 1 is rotation-only).

shearTot(*=Vector3r::Zero()*)
Current displacement in global coordinates.

step(*=0*)
Step number in which this engine is active; determines position in path, using pathSteps.

trsf(*=uninitalized*)
Transformation matrix for the local coordinate system. *(auto-updated)*

trsfQ(*=uninitalized*)
Transformation quaterion for the local coordinate system. *(auto-updated)*

uPrev(*=Vector3r::Zero()*)
Displacement value reached in the previous step.

class yade.wrapper.**PressTestEngine**(*inherits TranslationEngine → PartialEngine → Engine → Serializable*)
This class simulates the simple press work When the press cracks the solid brittle material, it returns back to the initial position and stops the simulation loop.

numberIterationAfterDestruction(*=0*)
The number of iterations, which will be carry out after destruction [-]

predictedForce(*=0*)
The minimal force, after what the engine will look for a destruction [N]

riseUpPressHigher(*=1*)
After destruction press rises up. This is the relationship between initial press velocity and velocity for going *back* [-]

class yade.wrapper.**RotationEngine**(*inherits 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]

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.

zeroPoint(*=Vector3r::Zero()*)
Point around which bodies will rotate if `rotateAroundZero` is True

class yade.wrapper.**StepDisplacer**(*inherits PartialEngine → Engine → Serializable*)
Apply generalized displacement (displacement or rotation) stepwise on subscribed bodies.

deltaSe3(*=Se3r(Vector3r(NaN, NaN, NaN), Quaternionr::Identity())*)
|ydeprec| Set mov/rot directly instead; kept only for backwards compat. If the 0th component of the vector is not NaN, then it was updated by the user, warning is issued and mov and rot are updated automatically.

mov(*=Vector3r::Zero()*)
Linear displacement step to be applied per iteration, by addition to State.pos.

rot(*=Quaternionr::Identity()*)
Rotation step to be applied per iteration (via rotation composition with State.ori).

setVelocities(*=false*)
If true, velocity and angularVelocity are modified in such a way that over one iteration (dt), the body will have the prescribed jump. In this case, position and orientation itself is not updated for dynamic bodies, since they would have the delta applied twice (here and in the integrator). For non-dynamic bodies however, they *are* still updated.

class yade.wrapper.**TorqueEngine**(*inherits PartialEngine → Engine → Serializable*)
Apply given torque (momentum) value at every subscribed particle, at every step.

moment(*=Vector3r::Zero()*)
Torque value to be applied.

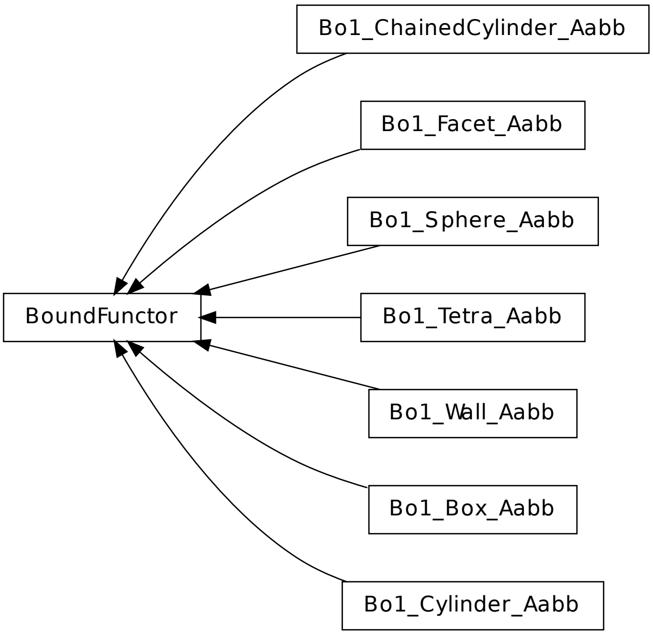
class yade.wrapper.**TranslationEngine**(*inherits PartialEngine → Engine → Serializable*)
This engine is the base class for different engines, which require any kind of motion.

translationAxis(*=uninitalized*)
Direction [Vector3]

velocity(*=uninitalized*)
Velocity [m/s]

6.5 Bounding volume creation

6.5.1 BoundFunctor



class yade.wrapper.**BoundFunctor**(*inherits Functor → Serializable*)
Functor for creating/updating Body::bound.

class yade.wrapper.**Bo1_Box_Aabb**(*inherits BoundFunctor → Functor → Serializable*)
Create/update an Aabb of a Box.

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_Dem3DofGeom::distFactor / Ig2_Cylinder_Cylinder_ScGeom::interactionDetectionFactor should have the same value as aabbEnlargeFactor.

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_Dem3DofGeom::distFactor / Ig2_Cylinder_Cylinder_ScGeom::interactionDetectionFactor should have the same value as aabbEnlargeFactor.

class `yade.wrapper.Bo1_Facet_Aabb`(*inherits BoundFunctor → Functor → Serializable*)

Creates/updates an Aabb of a Facet.

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_Dem3DofGeom::distFactor / Ig2_Sphere_Sphere_ScGeom::interactionDetectionFactor should have the same value as aabbEnlargeFactor.

class `yade.wrapper.Bo1_Tetra_Aabb`(*inherits BoundFunctor → Functor → Serializable*)

Create/update Aabb of a Tetra

class `yade.wrapper.Bo1_Wall_Aabb`(*inherits BoundFunctor → Functor → Serializable*)

Creates/updates an Aabb of a Wall

6.5.2 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)

`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.

`functors`

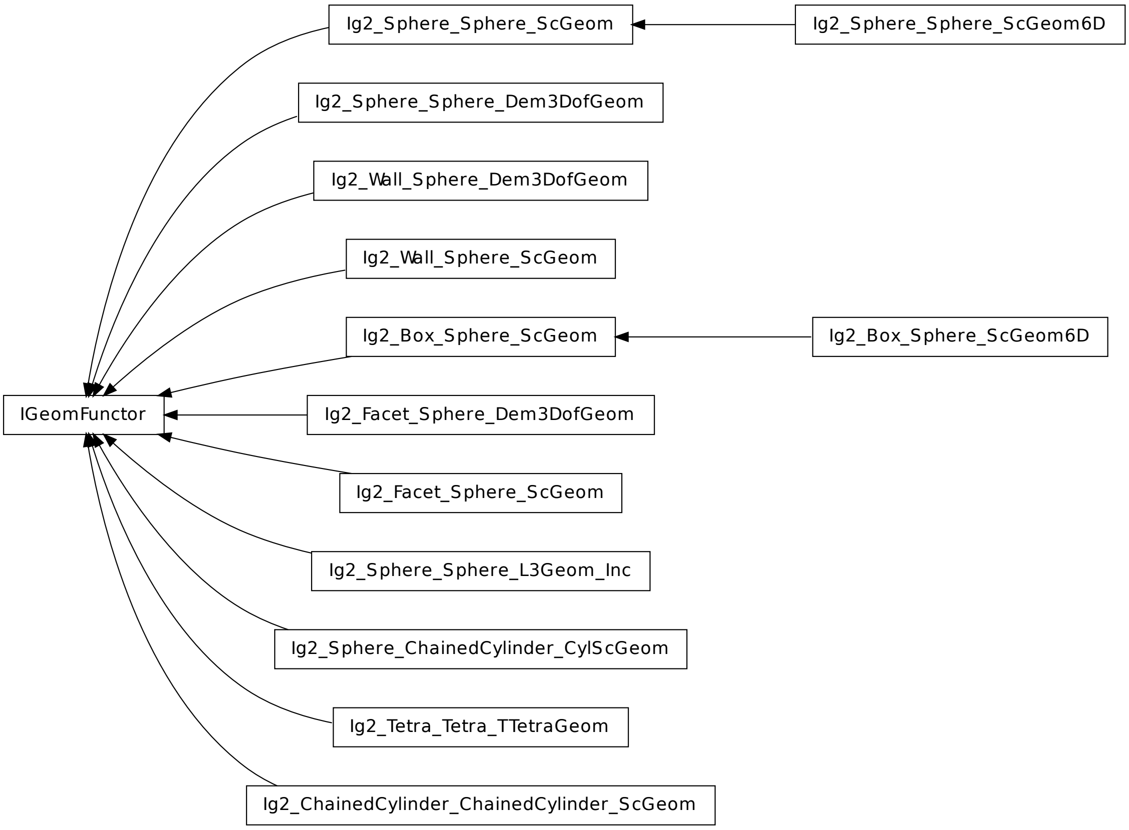
Functors associated with this dispatcher.

`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).

6.6 Interaction Geometry creation

6.6.1 IGeomFunctor


Ig2_Sphere_Sphere_ScGeom
Ig2_Sphere_Sphere_ScGeom6D
Ig2_Sphere_Sphere_Dem3DofGeom
Ig2_Wall_Sphere_Dem3DofGeom
Ig2_Wall_Sphere_ScGeom
Ig2_Box_Sphere_ScGeom
Ig2_Box_Sphere_ScGeom6D
IGeomFunctor
Ig2_Facet_Sphere_Dem3DofGeom
Ig2_Facet_Sphere_ScGeom
Ig2_Sphere_Sphere_L3Geom_Inc
Ig2_Sphere_ChainedCylinder_CylScGeom
Ig2_Tetra_Tetra_TTetraGeom
Ig2_ChainedCylinder_ChainedCylinder_ScGeom

class `yade.wrapper.`**`IGeomFunctor`**(*inherits Functor → Serializable*)
Functor for creating/updating Interaction::geom objects.

class `yade.wrapper.`**`Ig2_Box_Sphere_ScGeom`**(*inherits IGeomFunctor → Functor → Serializable*)
Create an interaction geometry ScGeom from Box and Sphere

class `yade.wrapper.`**`Ig2_Box_Sphere_ScGeom6D`**(*inherits Ig2_Box_Sphere_ScGeom → IGeomFunctor → Functor → Serializable*)
Create an interaction geometry ScGeom6D from Box and Sphere

class `yade.wrapper.`**`Ig2_ChainedCylinder_ChainedCylinder_ScGeom`**(*inherits IGeomFunctor → Functor → Serializable*)
Create/update a ScGeom instance representing connexion between chained cylinders.

`interactionDetectionFactor`(*=1*)
Enlarge both radii by this factor (if >1), to permit creation of distant interactions.

class `yade.wrapper.`**`Ig2_Facet_Sphere_Dem3DofGeom`**(*inherits IGeomFunctor → Functor → Serializable*)
Compute geometry of facet-sphere contact with normal and shear DOFs. As in all other Dem3DofGeom-related classes, total formulation of both shear and normal deformations is used. See Dem3DofGeom_FacetSphere for more information.

class `yade.wrapper.`**`Ig2_Facet_Sphere_ScGeom`**(*inherits IGeomFunctor → Functor → Serializable*)
Create/update a ScGeom instance representing intersection of Facet and Sphere.

`shrinkFactor`(*=0, no shrinking*)
The radius of the inscribed circle of the facet is decreased by the value of the sphere's ra-

dius 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.

class `yade.wrapper.Ig2_Sphere_ChainedCylinder_CylScGeom`(*inherits IGeomFunctor → Functor → Serializable*)

Create/update a ScGeom instance representing intersection of two Spheres.

`interactionDetectionFactor`(*=1*)

Enlarge both radii by this factor (if >1), to permit creation of distant interactions.

class `yade.wrapper.Ig2_Sphere_Sphere_Dem3DofGeom`(*inherits IGeomFunctor → Functor → Serializable*)

Functor handling contact of 2 spheres, producing Dem3DofGeom instance

`distFactor`(*=-1*)

Factor of sphere radius such that sphere "touch" if their centers are not further than distFactor*(r1+r2); if negative, equilibrium distance is the sum of the sphere's radii.

class `yade.wrapper.Ig2_Sphere_Sphere_L3Geom_Inc`(*inherits IGeomFunctor → Functor → Serializable*)

Incrementally compute L3Geom for contact of 2 spheres.

Note: The initial value of *u[0]* (normal displacement) might be non-zero, with or without *distFactor*, since it is given purely by sphere's geometry. If you want to set "equilibrium distance", do it in the contact law as explained in L3Geom.u0.

`approxMask`

Selectively enable geometrical approximations (bitmask); add the values for approximations to be enabled.

1: do not renormalize transformation matrix at every step 2: 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. 4: do not take average (mid-step) normal when computing relative shear displacement, use previous value instead 8: do not re-normalize average (mid-step) normal, if used.… By default, the mask is zero and neither of these approximations is used.

`distFactor`(*=1*)

Create interaction if spheres are not futher than distFactor*(r1+r2).

`noRatch`(*=true*)

See ScGeom.avoidGranularRatcheting.

class `yade.wrapper.Ig2_Sphere_Sphere_ScGeom`(*inherits IGeomFunctor → Functor → Serializable*)

Create/update a ScGeom instance representing intersection of two Spheres.

`avoidGranularRatcheting`

Granular ratcheting is mentioned in [GarciaRojo2004], [Alonso2004], [McNamara2008].

Unfortunately, published papers tend to focus on the "good" ratcheting, i.e. irreversible deformations due to the intrinsic nature of frictional granular materials, while a 2004 talk of McNamara in Paris clearly mentioned a possible "bad" ratcheting, purely linked with the formulation of the contact laws in what he called "molecular dynamics" (i.e. Cundall's model, as opposed to "contact dynamics" from Moreau and Jean).

The bad ratcheting is best understood considering this 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 to avoid that is quite simple in the case of linear-elastic laws: use a constant branch vector, which is what this functor does by default.

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.

Warning: Functionally equal class Ig2_Sphere_Sphere_Dem3DofGeom (which creates Dem3DofGeom rather than ScGeom) calls this parameter distFactor, but its semantics is *different* in some aspects.

class `yade.wrapper.Ig2_Sphere_Sphere_ScGeom6D`(*inherits Ig2_Sphere_Sphere_ScGeom → IGeomFunctor → Functor → Serializable*)

Create/update a ScGeom6D instance representing intersection of two Spheres.

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_ScGeom_CohFrictPhys_CohesionMoment.

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)

class `yade.wrapper.Ig2_Wall_Sphere_Dem3DofGeom`(*inherits IGeomFunctor → Functor → Serializable*)

Create/update contact of Wall and Sphere (Dem3DofGeom_WallSphere instance)

class `yade.wrapper.Ig2_Wall_Sphere_ScGeom`(*inherits IGeomFunctor → Functor → Serializable*)

Create/update a ScGeom instance representing intersection of Wall and Sphere.

noRatch(*=true*)

Avoid granular ratcheting

6.6.2 IGeomDispatcher

class `yade.wrapper.IGeomDispatcher`(*inherits Dispatcher → Engine → Serializable*)

Dispatcher calling functors based on received argument type(s).

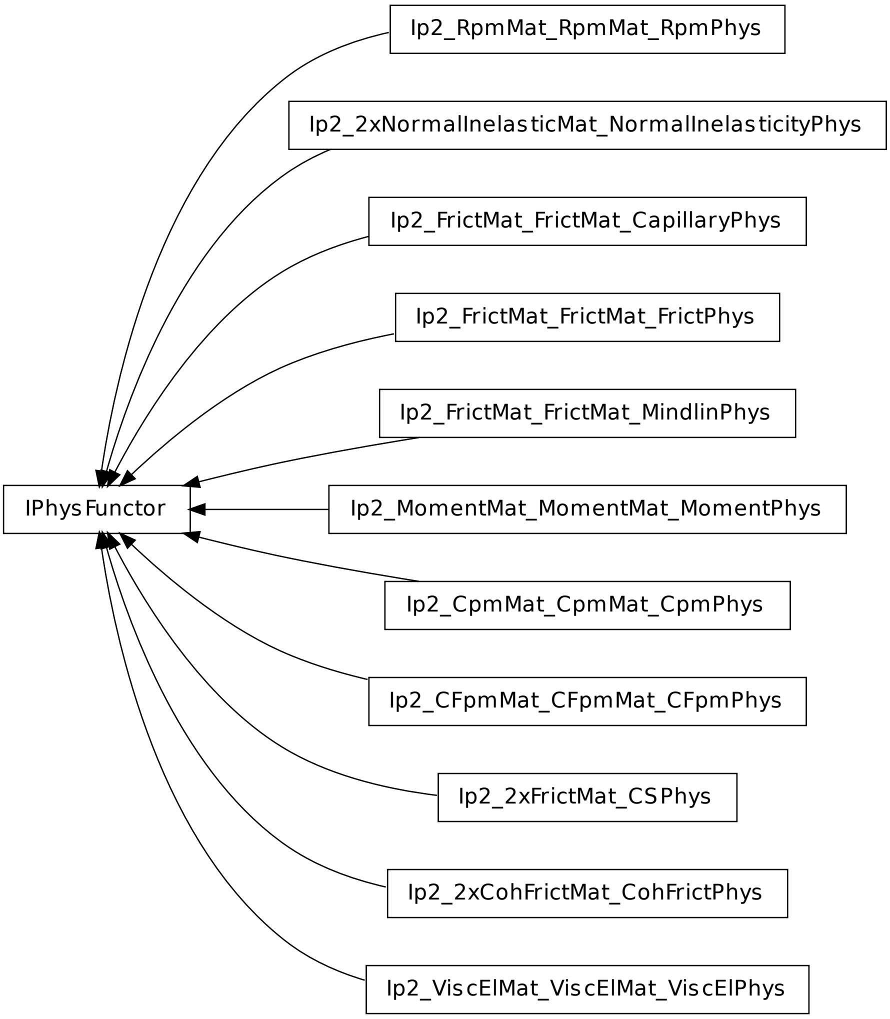
dispFunctor(*(Shape)arg2*, *(Shape)arg3*) → IGeomFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix([*(bool)names=True*]) → dict
Return dictionary with contents of the dispatch matrix.

functors
Functors associated with this dispatcher.

6.7 Interaction Physics creation

6.7.1 IPhysFunctor


Ip2_RpmMat_RpmMat_RpmPhys
Ip2_2xNormalInelasticMat_NormalInelasticityPhys
Ip2_FrictMat_FrictMat_CapillaryPhys
Ip2_FrictMat_FrictMat_FrictPhys
Ip2_FrictMat_FrictMat_MindlinPhys
IPhysFunctor
Ip2_MomentMat_MomentMat_MomentPhys
Ip2_CpmMat_CpmMat_CpmPhys
Ip2_CFpmMat_CFpmMat_CFpmPhys
Ip2_2xFrictMat_CSPhys
Ip2_2xCohFrictMat_CohFrictPhys
Ip2_ViscElMat_ViscElMat_ViscElPhys

class yade.wrapper.**IPhysFunctor**(*inherits Functor → Serializable*)
Functor for creating/updating Interaction::phys objects.

class yade.wrapper.**Ip2_2xCohFrictMat_CohFrictPhys**(*inherits IPhysFunctor → Functor → Serializable*)
Generates cohesive-frictional interactions with moments. Used in the contact law Law2_ScGeom_CohFrictPhys_CohesionMoment.

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_2xCohFrictMat_CohFrictPhys::setCohesionNow), and new contacts are only frictional.

class yade.wrapper.Ip2_2xFrictMat_CSPhys(*inherits IPhysFunctor → Functor → Serializable*)
Functor creating CSPhys from two FrictMat. See Law2_Dem3Dof_CSPhys_CundallStrack for details.

class yade.wrapper.Ip2_2xNormalInelasticMat_NormalInelasticityPhys(*inherits IPhysFunctor → Functor → Serializable*)
The RelationShips for using Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity

In these RelationShips all the attributes of the interactions (which are of NormalInelasticityPhys type) are computed.

> **Warning:** as in the others Ip2 functors, most of the attributes are computed only once, when the interaction is new.

betaR(*=0.12*)
Parameter for computing the torque-stifness : T-stifness = betaR * Rmoy^2

class yade.wrapper.Ip2_CFpmMat_CFpmMat_CFpmPhys(*inherits IPhysFunctor → Functor → Serializable*)
Converts 2 CFpmmat instances to CFpmPhys with corresponding parameters.

Alpha(*=0*)
Defines the ratio ks/kn.

Beta(*=0*)
Defines the ratio kr/(ks*meanRadius^2) to compute the resistive moment in rotation. [-]

cohesion(*=0*)
Defines the maximum admissible tangential force in shear FsMax=cohesion*crossSection. [Pa]

cohesiveTresholdIteration(*=1*)
Should new contacts be cohesive? They will before this iter, they won't afterward.

eta(*=0*)
Defines the maximum admissible resistive moment in rotation MtMax=eta*meanRadius*Fn. [-]

strengthSoftening(*=0*)
Defines the softening when Dtensile is reached to avoid explosion of the contact. Typically, when D > Dtensile, Fn=FnMax - (kn/strengthSoftening)*(Dtensile-D). [-]

tensileStrength(*=0*)
Defines the maximum admissible normal force in traction FnMax=tensileStrength*crossSection. [Pa]

useAlphaBeta(*=false*)
If true, stiffnesses are computed based on Alpha and Beta.

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.

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).

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.

class `yade.wrapper.Ip2_FrictMat_FrictMat_FrictPhys`(*inherits 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.
>
> The friction angle of the contact is defined as the minimum angle of the two materials in contact.

Only interactions with ScGeom or Dem3DofGeom geometry are meaningfully accepted; run-time typecheck can make this functor unnecessarily slow in general. Such design is problematic in itself, though – from http://www.mail-archive.com/yade-dev@lists.launchpad.net/msg02603.html:

> You have to suppose some exact type of IGeom in the Ip2 functor, but you don't know anything about it (Ip2 only guarantees you get certain IPhys types, via the dispatch mechanism).
>
> That means, unless you use Ig2 functor producing the desired type, the code will break (crash or whatever). The right behavior would be either to accept any type (what we have now, at least in principle), or really enforce IGeom type of the interation passed to that particular Ip2 functor.

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's formulation (as implemented in PFC).

Viscous parameters can be specified either using coefficients of restitution (e_n, e_s) or viscous damping coefficient (β_n, β_s). The following rules apply: #. If the β_n (β_s) coefficient 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/shots.py for an example of specifying e_n based on combination of parameters.

`betan`(*=uninitalized*)
: Normal viscous damping coefficient β_n.

`betas`(*=uninitalized*)
: Shear viscous damping coefficient β_s.

`en`(*=uninitalized*)
: Normal coefficient of restitution e_n.

es(*=uninitalized*)
Shear coefficient of restitution e_s.

gamma(*=0.0*)
Surface energy parameter [J/m^2] per each unit contact surface, to derive DMT formulation from HM

class yade.wrapper.**Ip2_MomentMat_MomentMat_MomentPhys**(*inherits IPhysFunctor → Functor → Serializable*)
Create MomentPhys from 2 instances of MomentMat.

1. If boolean userInputStiffness=true & useAlphaBeta=false, users can input Knormal, Kshear and Krotate directly. Then, kn,ks and kr will be equal to these values, rather than calculated E and v.
2. If boolean userInputStiffness=true & useAlphaBeta=true, users input Knormal, Alpha and Beta. Then ks and kr are calculated from alpha & beta respectively.
3. If both are false, it calculates kn and ks are calculated from E and v, whilst kr = 0.

Alpha(*=0*)
Ratio of Ks/Kn

Beta(*=0*)
Ratio to calculate Kr

Knormal(*=0*)
Allows user to input stiffness properties from triaxial test. These will be passed to MomentPhys or NormShearPhys

Krotate(*=0*)
Allows user to input stiffness properties from triaxial test. These will be passed to MomentPhys or NormShearPhys

Kshear(*=0*)
Allows user to input stiffness properties from triaxial test. These will be passed to MomentPhys or NormShearPhys

useAlphaBeta(*=false*)
for users to choose whether to input stiffness directly or use ratios to calculate Ks/Kn

userInputStiffness(*=false*)
for users to choose whether to input stiffness directly or use ratios to calculate Ks/Kn

class yade.wrapper.**Ip2_RpmMat_RpmMat_RpmPhys**(*inherits IPhysFunctor → Functor → Serializable*)
Convert 2 RpmMat instances to RpmPhys with corresponding parameters.

initDistance(*=0*)
Initial distance between spheres at the first step.

class yade.wrapper.**Ip2_ViscElMat_ViscElMat_ViscElPhys**(*inherits IPhysFunctor → Functor → Serializable*)
Convert 2 instances of ViscElMat to ViscElPhys using the rule of consecutive connection.

6.7.2 IPhysDispatcher

class yade.wrapper.**IPhysDispatcher**(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

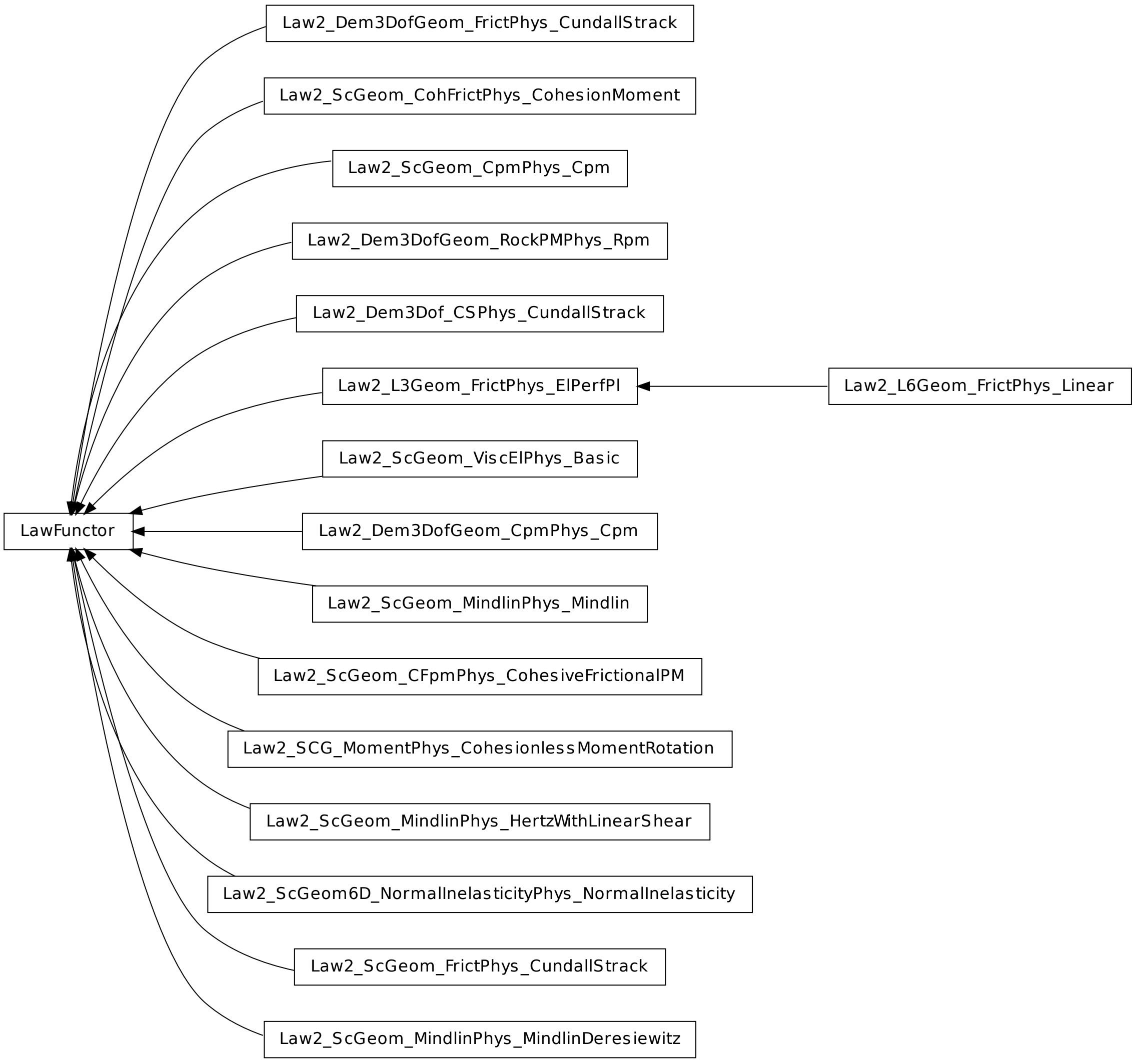
dispFunctor(*(Material)arg2, (Material)arg3*) → IPhysFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix([*(bool)names=True*]) → dict
Return dictionary with contents of the dispatch matrix.

functors
Functors associated with this dispatcher.

6.8 Constitutive laws

6.8.1 LawFunctor


Law2_Dem3DofGeom_FrictPhys_CundallStrack
Law2_ScGeom_CohFrictPhys_CohesionMoment
Law2_ScGeom_CpmPhys_Cpm
Law2_Dem3DofGeom_RockPMPhys_Rpm
Law2_Dem3Dof_CSPhys_CundallStrack
Law2_L3Geom_FrictPhys_ElPerfPl
Law2_L6Geom_FrictPhys_Linear
Law2_ScGeom_ViscElPhys_Basic
LawFunctor
Law2_Dem3DofGeom_CpmPhys_Cpm
Law2_ScGeom_MindlinPhys_Mindlin
Law2_ScGeom_CFpmPhys_CohesiveFrictionalPM
Law2_SCG_MomentPhys_CohesionlessMomentRotation
Law2_ScGeom_MindlinPhys_HertzWithLinearShear
Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity
Law2_ScGeom_FrictPhys_CundallStrack
Law2_ScGeom_MindlinPhys_MindlinDeresiewitz

class `yade.wrapper.LawFunctor`(*inherits Functor → Serializable*)

Functor for applying constitutive laws on interactions.

class `yade.wrapper.Law2_Dem3DofGeom_CpmPhys_Cpm`(*inherits LawFunctor → Functor → Serializable*)

Constitutive law for the *cpm-model.*

`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)

`funcG`(*(float)epsCrackOnset, (float)epsFracture*$\Big[$, *(bool)neverDamage=False*$\Big]$) → 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).

`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 $\to +\infty$

`relKnSoft`(*=.3*)

Relative rigidity of the softening branch in compression (0=perfect elastic-plastic, <0 softening, >0 hardening)

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_Dem3DofGeom_FrictPhys_CundallStrack**(*inherits LawFunctor → Functor → Serializable*)
Constitutive law for linear compression, no tension, and linear plasticity surface.

This class serves also as tutorial and is documented in detail at https://yade-dem.org/index.php/ConstitutiveLawHowto.

class yade.wrapper.**Law2_Dem3DofGeom_RockPMPhys_Rpm**(*inherits LawFunctor → Functor → Serializable*)
Constitutive law for the Rpm model

class yade.wrapper.**Law2_Dem3Dof_CSPhys_CundallStrack**(*inherits LawFunctor → Functor → Serializable*)
Basic constitutive law published originally by Cundall&Strack; it has normal and shear stiffnesses (Kn, Kn) and dry Coulomb friction. Operates on associated Dem3DofGeom and CSPhys instances.

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`).

noBreak(*=false*)
Do not break contacts when particles separate.

noSlip(*=false*)
No plastic slipping.

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.

charLen(*=1*)
Characteristic length with the meaning of the stiffness ratios bending/shear and torsion/normal.

class yade.wrapper.**Law2_SCG_MomentPhys_CohesionlessMomentRotation**(*inherits LawFunctor → Functor → Serializable*)
Contact law based on Plassiard et al. (2009) : A spherical discrete element model: calibration procedure and incremental response. The functionality has been verified with results in the paper.

The contribution of stiffnesses are scaled according to the radius of the particle, as implemented in that paper.

See also associated classes MomentMat, Ip2_MomentMat_MomentMat_MomentPhys, MomentPhys.

Note: This constitutive law can be used with triaxial test, but the following significant changes in code have to be made: Ip2_MomentMat_MomentMat_MomentPhys and Law2_SCG_MomentPhys_CohesionlessMomentRotation have to be added. Since it uses ScGeom, it uses boxes rather than facets. Spheres and boxes have to be changed to MomentMat rather than FrictMat.

preventGranularRatcheting(*=false*)

??

class yade.wrapper.**Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity**(*inherits LawFunctor → Functor → Serializable*)

Contact law used to simulate granulate filler in rock joints [Duriez2009a], [Duriez2010]. 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 scripts/normalInelasticityTest.py

momentAlwaysElastic(*=false*)

boolean, true=> the torque (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

class yade.wrapper.**Law2_ScGeom_CFpmPhys_CohesiveFrictionalPM**(*inherits LawFunctor → Functor → Serializable*)

Constitutive law for the CFpm model.

preventGranularRatcheting(*=true*)

If true rotations are computed such as granular ratcheting is prevented. See article [Alonso2004], pg. 3-10 – and a lot more papers from the same authors).

class yade.wrapper.**Law2_ScGeom_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_n 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 (in course of implementation), the behaviour is perfectly plastic, and the shear strength is kept constant.

If Law2_ScGeom_CohFrictPhys_CohesionMoment::momentRotationLaw =True, bending and twisting moments are computed using a linear law with moduli respectively k_t and k_r (the two values are the same currently), 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. There is no maximum value of moments in the current implementation, though they could be added 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 $d\Theta_{t,e}/dt = -M_t/\nu_t$.

Note: Periodicity is not handled yet in this law.

always_use_moment_law(*=false*)
If true, use bending/twisting moments at all contacts. If false, compute moments only for cohesive contacts.

creep_viscosity(*=1*)
creep viscosity [Pa.s/m]. probably should be moved to Ip2_2xCohFrictMat_CohFrictPhys...

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.

twist_creep(*=false*)
activate creep on the twisting moment, using CohesiveFrictionalContactLaw::creep_viscosity.

class yade.wrapper.**Law2_ScGeom_CpmPhys_Cpm**(*inherits LawFunctor → Functor → Serializable*)
An experimental version of Law2_Dem3DofGeom_CpmPhys_Cpm which uses ScGeom instead of Dem3DofGeom.

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 →+∞

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.

Note: This law uses ScGeom; there is also functionally equivalent Law2_Dem3DofGeom_FrictPhys_CundallStrack, which uses Dem3DofGeom (sphere-box interactions are not implemented for the latest).

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_ScGeom_CohFrictPhys_CohesionMoment, which adds cohesion and moments at contact.

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)

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.

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.

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.

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.

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.

normDampDissip(*=uninitalized*)
Energy dissipation due to sliding

normElastEnergy() → float
Compute normal elastic potential energy. It handle the DMT formulation if Law2_ScGeom_MindlinPhys_Mindlin::includeAdhesion is set to true.

preventGranularRatcheting(*=true*)
bool to avoid granular ratcheting

shearDampDissip(*=uninitalized*)
Energy dissipation due to sliding

shearEnergy(*=uninitalized*)
Shear elastic potential energy

class yade.wrapper.**Law2_ScGeom_MindlinPhys_MindlinDeresiewitz**(*inherits LawFunctor → Functor → Serializable*)
Hertz-Mindlin contact law with partial slip solution, as described in [Thornton1991].

class yade.wrapper.**Law2_ScGeom_ViscElPhys_Basic**(*inherits LawFunctor → Functor → Serializable*)
Linear viscoelastic model operating on ScGeom and ViscElPhys.

6.8.2 LawDispatcher

class yade.wrapper.**LawDispatcher**(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

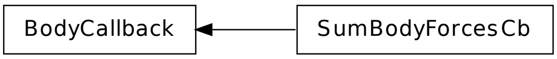
dispFunctor(*(IGeom)arg2, (IPhys)arg3*) → LawFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix($\left[\textit{(bool)names=True}\right]$) → dict
Return dictionary with contents of the dispatch matrix.

functors
Functors associated with this dispatcher.

6.9 Callbacks

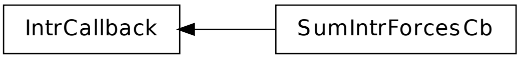
6.9.1 BodyCallback



class yade.wrapper.**BodyCallback**(*inherits Serializable*)
Abstract callback object which will be called for every Body after being processed by NewtonIntegrator. See IntrCallback for details.

class yade.wrapper.**SumBodyForcesCb**(*inherits BodyCallback → Serializable*)
Callback summing magnitudes of resultant forces over dynamic bodies.

6.9.2 IntrCallback



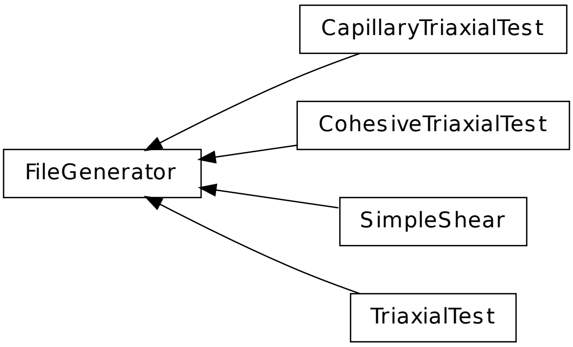
class yade.wrapper.**IntrCallback**(*inherits Serializable*)
Abstract callback object which will be called for every (real) Interaction after the interaction has been processed by InteractionLoop.

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.

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).

6.10 Preprocessors



class `yade.wrapper.FileGenerator`(*inherits Serializable*)

Base class for scene generators, preprocessors.

`generate`(*(str)out*) → None

Generate scene, save to given file

`load`() → None

Generate scene, save to temporary file and load immediately

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.'

`CapillaryPressure`(*=0*)

Define succion in the packing [Pa]. This is the value used in the capillary model.

`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.

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

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

importFilename(*=”“*)
: File with positions and sizes of spheres.

internalCompaction(*=false*)
: flag for choosing between moving boundaries or increasing particles sizes during the compaction stage.

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.

sigmaLateralConfinement(*=50000*)
Lateral stress during triaxial loading. 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

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_2xCohFrictMat_CohFrictPhys, then turn Ip2_2xCohFrictMat_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

`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?

importFilename(*=""*)
File with positions and sizes of spheres.

internalCompaction(*=false*)
flag for choosing between moving boundaries or increasing particles sizes during the compaction stage.

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.

sigmaLateralConfinement(*=50000*)
Lateral stress during triaxial loading. 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

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 could be generated via the same method used in TriaxialTest Preprocesor (=> see GenerateCloud) or by reading a text file containing positions and radii of a sample (=> see ImportCloud). This last one is the one by default used by this PreProcessor as it is written here => you need to have such a file.

  Thanks to the Engines (in pkg/common/Engine/PartialEngine) KinemCNDEngine, KinemCNSEngine and KinemCNLEngine, respectively constant normal displacement, constant normal rigidity and constant normal stress are possible to execute over such samples.

NB about micro-parameters : their values correspond to those used in [Duriez2009a].

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]

filename(*="../porosite0_44.txt"*)
file with the list of spheres centers and radii

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 [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]

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

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 TrixaialCompressionEngine 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 inc**
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

`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.

`importFilename`(=*"“*)
File with positions and sizes of spheres.

`internalCompaction`(=*false*)
flag for choosing between moving boundaries or increasing particles sizes during the compaction stage.

`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.

sigmaLateralConfinement(*=50000*)
Lateral stress during triaxial loading. 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

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)

6.11 Rendering

6.11.1 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

`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

`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).

`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.

`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.

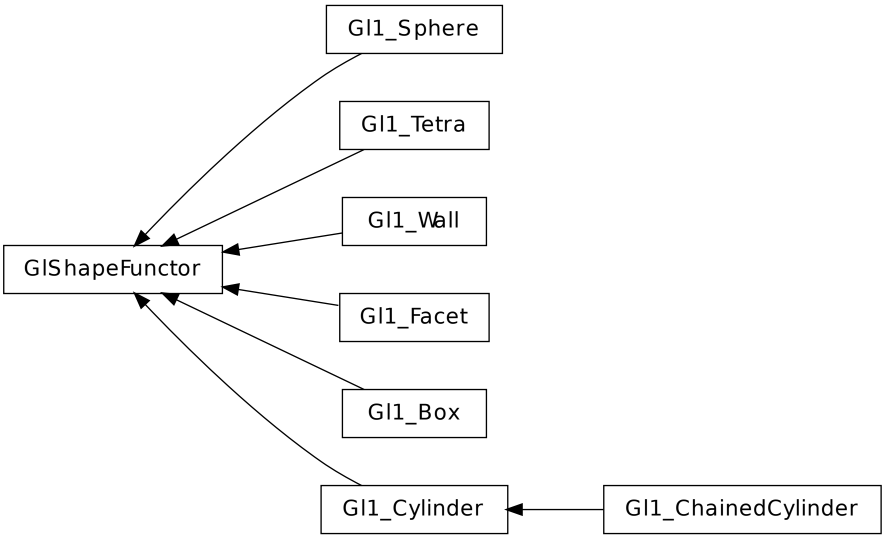
`selId`(*=Body::ID_NONE*)
Id of particle that was selected by the user.

`setRefSe3`() → None
Make current positions and orientation reference for scaleDisplacements and scaleRotations.

`shape`(*=true*)
Render body Shape

`wire`(*=false*)
Render all bodies with wire only (faster)

6.11.2 GlShapeFunctor



class `yade.wrapper.`**`GlShapeFunctor`**(*inherits Functor → Serializable*)
  Abstract functor for rendering Shape objects.

class `yade.wrapper.`**`Gl1_Box`**(*inherits GlShapeFunctor → Functor → Serializable*)
  Renders Box object

class `yade.wrapper.`**`Gl1_ChainedCylinder`**(*inherits Gl1_Cylinder → GlShapeFunctor → Functor → Serializable*)
  Renders ChainedCylinder object including a shift for compensating flexion.

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.

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.

class `yade.wrapper.`**`Gl1_Sphere`**(*inherits GlShapeFunctor → Functor → Serializable*)
  Renders Sphere object

  `wire`(*=false* **[static]**)
    Only show wireframe (controlled by `glutSlices` and `glutStacks`.

  `stripes`(*=false* **[static]**)
    In non-wire rendering, show stripes clearly showing particle rotation.

  `glutNormalize`(*=true* **[static]**)
    Fix normals for non-wire rendering; see http://lists.apple.com/archives/Mac-opengl/2002/Jul/msg00085.html

glutSlices(=*12* [**static**])
Number of sphere slices; not used with **stripes** (see glut{Solid,Wire}Sphere reference)

glutStacks(=*6* [**static**])
Number of sphere stacks; not used with **stripes** (see glut{Solid,Wire}Sphere reference)

class yade.wrapper.**Gl1_Tetra**(*inherits GlShapeFunctor → Functor → Serializable*)
Renders Tetra object

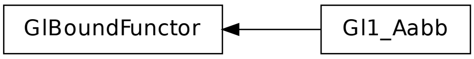
class yade.wrapper.**Gl1_Wall**(*inherits GlShapeFunctor → Functor → Serializable*)
Renders Wall object

div(=*20* [**static**])
Number of divisions of the wall inside visible scene part.

6.11.3 GlStateFunctor

class yade.wrapper.**GlStateFunctor**(*inherits Functor → Serializable*)
Abstract functor for rendering State objects.

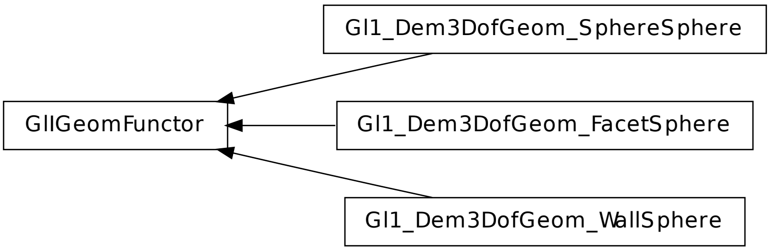
6.11.4 GlBoundFunctor



class yade.wrapper.**GlBoundFunctor**(*inherits Functor → Serializable*)
Abstract functor for rendering Bound objects.

class yade.wrapper.**Gl1_Aabb**(*inherits GlBoundFunctor → Functor → Serializable*)
Render Axis-aligned bounding box (Aabb).

6.11.5 GlIGeomFunctor


Gl1_Dem3DofGeom_SphereSphere
GlIGeomFunctor
Gl1_Dem3DofGeom_FacetSphere
Gl1_Dem3DofGeom_WallSphere

class yade.wrapper.**GlIGeomFunctor**(*inherits Functor → Serializable*)
Abstract functor for rendering IGeom objects.

class yade.wrapper.**Gl1_Dem3DofGeom_FacetSphere**(*inherits GlIGeomFunctor → Functor → Serializable*)
Render interaction of facet and sphere (represented by Dem3DofGeom_FacetSphere)

normal(=*false* [**static**])
Render interaction normal

rolledPoints(=*false* [**static**])
Render points rolled on the sphere & facet (original contact point)

unrolledPoints(=*false* [**static**])
Render original contact points unrolled to the contact plane

shear(=*false* [**static**])
Render shear line in the contact plane

shearLabel(=*false* [**static**])
Render shear magnitude as number

class yade.wrapper.**Gl1_Dem3DofGeom_SphereSphere**(*inherits GlIGeomFunctor → Functor → Serializable*)
Render interaction of 2 spheres (represented by Dem3DofGeom_SphereSphere)

normal(=*false* [**static**])
Render interaction normal

rolledPoints(=*false* [**static**])
Render points rolled on the spheres (tracks the original contact point)

unrolledPoints(=*false* [**static**])
Render original contact points unrolled to the contact plane

shear(=*false* [**static**])
Render shear line in the contact plane

shearLabel(=*false* [**static**])
Render shear magnitude as number

class yade.wrapper.**Gl1_Dem3DofGeom_WallSphere**(*inherits GlIGeomFunctor → Functor → Serializable*)
Render interaction of wall and sphere (represented by Dem3DofGeom_WallSphere)

normal(=*false* [**static**])
Render interaction normal

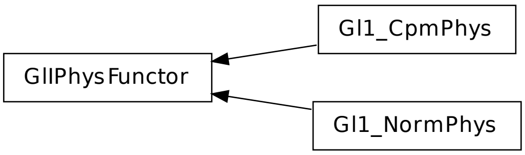
rolledPoints(=*false* [**static**])
Render points rolled on the spheres (tracks the original contact point)

unrolledPoints(=*false* [**static**])
Render original contact points unrolled to the contact plane

shear(=*false* [**static**])
Render shear line in the contact plane

shearLabel(=*false* [**static**])
Render shear magnitude as number

6.11.6 GlIPhysFunctor



class yade.wrapper.**GlIPhysFunctor**(*inherits Functor → Serializable*)
Abstract functor for rendering IPhys objects.

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 ε_0 × *colorStrainRatio* (ε_0=:yref:*CpmPhys.epsCrackOnset*). Otherwise, color based on the residual strength.

epsNLabel(*=false* **[static]**)
Numerically show normal strain

class `yade.wrapper.Gl1_NormPhys`(*inherits GlIPhysFunctor → Functor → Serializable*)
Renders NormPhys objects as cylinders of which diameter and color depends on NormPhys:normForce magnitude.

maxFn(*=0* **[static]**)
Value of NormPhys.normalForce corresponding to maxDiameter. 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)

6.12 Simulation data

6.12.1 Omega

class `yade.wrapper.Omega`

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.

- assigning negative value enables dynamic Δt (by looking for a TimeStepper in O.engine) and sets positive timestep `O.dt=|Δt|` (will be used until the timestepper is run and updates it)
- assigning positive value sets Δt to that value and disables dynamic Δt (via TimeStepper, if there is one).

dynDt can be used to query whether dynamic Δt is in use.

`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 (Scene::engines).

`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.

`initializers`
List of initializers (Scene::initializers).

`interactions`
Interactions in the current simulation (container supporting index acces by either (id1,id2) or interactionNumber and iteration)

`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)arg2*) → None
Load simulation from file.

`loadTmp`([*(str)mark=''*]) → 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() → None
Reload current simulation

reset() → None
Reset simulations completely (including another 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)arg2*) → None
Save current simulation to file (should be .xml or .xml.bz2)

saveTmp([*(str)mark='‘*]) → None
Save simulation to memory (disappears at shutdown), can be loaded later with loadTmp. *mark* optionally distinguishes different memory-saved simulations.

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.

subStep
Get the current subStep number (only meaningful if 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])

tags
Tags (string=string dictionary) of the current simulation (container supporting string-index access/assignment)

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).

6.12.2 BodyContainer

class yade.wrapper.**BodyContainer**

__init__(*(BodyContainer)arg2*) → None

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*) → tuple
Append given list of bodies as a clump (rigid aggregate); return list of ids.

clear() → None
Remove all bodies (interactions not checked)

clump(*(object)arg2*) → int
Clump given bodies together (creating a rigid aggregate); returns clump id.

erase(*(int)arg2*) → bool
Erase body with the given id; all interaction will be deleted by InteractionLoop in the next step.

replace(*(object)arg2*) → object

6.12.3 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.iteraction 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

`clear()` → None
Remove all interactions

`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 .

`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.

6.12.4 ForceContainer

class `yade.wrapper.ForceContainer`

`__init__(`*(ForceContainer)arg2*`)` → None

`addF(`*(int)id*, *(Vector3)f*`)` → None
Apply force on body (accumulates).

`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*`)` → None
Apply torque on body (accumulates).

`f(`*(int)id*`)` → Vector3
Force applied on body.

`m(`*(int)id*`)` → Vector3
Deprecated alias for t (torque).

`move(`*(int)id*`)` → Vector3
Displacement applied on body.

`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*`)` → Vector3
Torque applied on body.

6.12.5 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.

6.12.6 Scene

class `yade.wrapper.Scene`(*inherits Serializable*)

Object comprising the whole simulation.

`dt`(*=1e-8*)

Current timestep for integration.

`isPeriodic`(*=false*)

Whether periodic boundary conditions are active.

`iter`(*=0*)

Current iteration (computational step) number

`needsInitializers`(*=true*)

Whether initializers will be run before the first step.

`selectedBody`(*=-1*)

Id of body that is selected by the user

`stopAtIter`(*=0*)

Iteration 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.

6.12.7 Cell

class yade.wrapper.**Cell**(*inherits Serializable*)

Parameters of periodic boundary conditions. Only applies if O.isPeriodic==True.

Hsize(*=Matrix3r::Zero()*)

The current cell size (one column per box edge), computed from *refSize* and *trsf (auto-updated)*

homoDeform(*=3*)

Deform (velGrad) the cell homothetically, by adjusting positions or velocities of particles. The values have the following meaning: 0: no homothetic deformation, 1: set absolute particle positions directly (when **velGrad** is non-zero), but without changing their velocity, 2: adjust particle velocity (only when **velGrad** changed) with Δv_i=Δ v x_i. 3: as 2, but include a 2nd order term in addition – the derivative of 1 (convective term in the velocity update).

prevVelGrad(*=Matrix3r::Zero()*)

Velocity gradient in the previous step.

refSize(*=Vector3r(1, 1, 1)*)

Reference size of the cell.

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 3 cell lateral vectors after applying current trsf. Update automatically at every step.

trsf(*=Matrix3r::Identity()*)

Current transformation matrix of the cell.

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)

velGrad(*=Matrix3r::Zero()*)

Velocity gradient of the transformation; used in NewtonIntegrator.

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.

6.13 Other classes

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.

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.

`timingDeltas`
Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

class `yade.wrapper.Cell`(*inherits Serializable*)
Parameters of periodic boundary conditions. Only applies if O.isPeriodic==True.

`Hsize`(*=Matrix3r::Zero()*)
The current cell size (one column per box edge), computed from *refSize* and *trsf (auto-updated)*

`homoDeform`(*=3*)
Deform (velGrad) the cell homothetically, by adjusting positions or velocities of particles. The values have the following meaning: 0: no homothetic deformation, 1: set absolute particle positions directly (when `velGrad` is non-zero), but without changing their velocity, 2: adjust particle velocity (only when `velGrad` changed) with Δv_i=Δ v x_i. 3: as 2, but include a 2nd order term in addition – the derivative of 1 (convective term in the velocity update).

`prevVelGrad`(*=Matrix3r::Zero()*)
Velocity gradient in the previous step.

`refSize`(*=Vector3r(1, 1, 1)*)
Reference size of the cell.

`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 3 cell lateral vectors after applying current trsf. Update automatically at every step.

`trsf`(*=Matrix3r::Identity()*)
Current transformation matrix of the cell.

`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)

`velGrad`(*=Matrix3r::Zero()*)
Velocity gradient of the transformation; used in NewtonIntegrator.

`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.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.GlIGeomDispatcher`(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

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.

functors
: Functors associated with this dispatcher.

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.

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.

class yade.wrapper.**GlShapeDispatcher**(*inherits Dispatcher → Engine → Serializable*)
: Dispatcher calling functors based on received argument type(s).

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.

functors
: Functors associated with this dispatcher.

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.

label(*=uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly fron python (must be a valid python identifier).

timingDeltas
: Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

class yade.wrapper.**Serializable**

dict() → dict
: Return dictionary of attributes.

name

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.

tester(*=uninitalized*)
: Associated LawTester object.

class yade.wrapper.**GlStateDispatcher**(*inherits Dispatcher → Engine → Serializable*)
: Dispatcher calling functors based on received argument type(s).

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.

functors
Functors associated with this dispatcher.

class `yade.wrapper.`**MatchMaker**(*inherits Serializable*)
Class matching pair of ids to return pre-defined or derived value of a scalar parameter.

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.

---

computeFallback(*(float)val1*, *(float)val2*) → float
Compute fallback value for *val1* and *val2*, using algorithm specified by fallback.

fallback
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 fallback algorithms do *not* require meaningful input values in order to work: * 'val' (return value specified by val) * 'zero' (return 0.)

matches(*=uninitalized*)
Array of `(id1,id2,value)` items; queries matching `id1``+``id2` or `id2``+``id1` will return `value`

val(*=NaN*)
Constant value returned if there is no match and fallback is `val`

class `yade.wrapper.`**GlBoundDispatcher**(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

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.

functors
Functors associated with this dispatcher.

class `yade.wrapper.`**GlIPhysDispatcher**(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

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.

functors
Functors associated with this dispatcher.

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).

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.

class yade.wrapper.**EnergyTracker**(*inherits Serializable*)
Storage for tracing energies. Only to be used if O.traceEnergy is True.

clear() → None
Clear all stored values.

energies(*=uninitalized*)
Energy values, in linear array

keys() → list
Return defined energies.

Chapter 7

Yade modules

7.1 yade.eudoxos module

Miscillaneous functions that are not believed to be generally usable, therefore kept in my "private" module here.

They comprise notably oofem export and various CPM-related functions.

class yade.eudoxos.**IntrSmooth3d**

Return spatially weigted gaussian average of arbitrary quantity defined on interactions.

At construction time, all real interactions are put inside spatial grid, permitting fast search for points in neighbourhood defined by distance.

Parameters for the distribution are standard deviation σ and relative cutoff distance *relThreshold* (3 by default) which will discard points farther than *relThreshold* ×σ.

Given central point p_0, points are weighted by gaussian function

$$\rho(p_0, p) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-||p_0 - p||^2}{2\sigma^2}\right)$$

To get the averaged value, simply call the instance, passing central point and callable object which received interaction object and returns the desired quantity:

```
>>> O.reset()
>>> from yade import utils
>>> O.bodies.append([utils.sphere((0,0,0),1),utils.sphere((0,0,1.9),1)])
[0, 1]
>>> O.engines=[InteractionLoop([Ig2_Sphere_Sphere_Dem3DofGeom(),],[Ip2_FrictMat_FrictMat_FrictPhys()]
>>> utils.createInteraction(0,1)
<Interaction instance at 0x...>
```

>> is3d=IntrSmooth3d(0.003) >> is3d((0,0,0),lambda i: i.phys.normalForce) Vector3(0,0,0)

bounds()

count()

yade.eudoxos.**displacementsInteractionsExport**(*fName*)

yade.eudoxos.**eliminateJumps**(*eps*, *sigma*, *numSteep=10*, *gapWidth=5*, *movWd=40*)

yade.eudoxos.**estimatePoissonYoung**(*principalAxis*, *stress=0*, *plot=False*, *cutoff=0.0*)

Estimate Poisson's ration given the "principal" axis of straining. For every base direction, homogenized strain is computed (slope in linear regression on discrete function particle coordinate → → particle displacement in the same direction as returned by utils.coordsAndDisplacements) and, (if axis '0' is the strained axis) the poisson's ratio is given as -½(ε1+ε2)/ε .

Young's modulus is computed as σ/ε ; if stress σ is not given (default 0), the result is 0.

cutoff, if > 0., will take only smaller part (centered) or the specimen into account

`yade.eudoxos.estimateStress`(*strain*, *cutoff=0.0*)

Use summed stored energy in contacts to compute macroscopic stress over the same volume, provided known strain.

`yade.eudoxos.oofemDirectExport`(*fileBase*, *title=None*, *negIds=[]*, *posIds=[]*)

`yade.eudoxos.oofemPrescribedDisplacementsExport`(*fileName*)

`yade.eudoxos.oofemTextExport`(*fName*)

Export simulation data in text format

The format is line-oriented as follows:

```
E G                                                # elastic material parameters
epsCrackOnset relDuctility xiShear transStrainCoeff # tensile parameters; epsFr=epsCrackOnset*relDuctility
cohesionT tanPhi                                   # shear parameters
number_of_spheres number_of_links
id x y z r boundary                                # spheres; boundary: -1 negative, 0 none, 1 positive
…
id1 id2 cp_x cp_y cp_z A                           # interactions; cp = contact point; A = cross-section
```

class `yade._eudoxos.HelixInteractionLocator2d`

Locate all real interactions in 2d plane (reduced by spiral projection from 3d, using `Shop::spiralProject`, which is the same as utils.spiralProject) using their contact points.

Note: Do not run simulation while using this object.

`__init__`(*(float)dH_dTheta*[, *(int)axis=0*[, *(float)periodStart=nan*[, *(float)theta0=0*[, *(float)thetaMin=nan*[, *(float)thetaMax=nan*]]]]]) → None

Parameters

- **dH_dTheta** (*float*) – Spiral inclination, i.e. height increase per 1 radian turn;
- **axis** (*int*) – axis of rotation (0=x,1=y,2=z)
- **theta** (*float*) – spiral angle at zero height (theta intercept)
- **thetaMin** (*float*) – only interactions with ϑ *thetaMin* will be considered (NaN to deactivate)
- **thetaMax** (*float*) – only interactions with ϑ *thetaMax* will be considered (NaN to deactivate)

See utils.spiralProject.

`hi`

Return upper corner of the rectangle containing all interactions.

`intrsAroundPt`(*(Vector2)pt2d*, *(float)radius*) → list

Return list of interaction objects that are not further from *pt2d* than *radius* in the projection plane

`lo`

Return lower corner of the rectangle containing all interactions.

`macroAroundPt`(*(Vector2)pt2d*, *(float)radius*) → tuple

Compute macroscopic stress around given point; the interaction (n and σ^T are rotated to the projection plane by ϑ (as given by utils.spiralProject) first, but no skew is applied). The formula used is

$$\sigma_{ij} = \frac{1}{V}\sum_{IJ} d^{IJ} A^{IJ}\left[\sigma^{N,IJ} n_i^{IJ} n_j^{IJ} + \frac{1}{2}\left(\sigma_i^{T,IJ} n_j^{IJ} + \sigma_j^{T,IJ} n_i^{IJ}\right)\right]$$

where the sum is taken over volume V containing interactions IJ between spheres I and J;

- i, j indices denote Cartesian components of vectors and tensors,

- d^{IJ} is current distance between spheres I and J,
- A^{IJ} is area of contact IJ,
- n is (ϑ-rotated) interaction normal (unit vector pointing from center of I to the center of J)
- $\sigma^{N,IJ}$ is normal stress (as scalar) in contact IJ,
- $\sigma^{T,IJ}$ is shear stress in contact IJ in global coordinates and ϑ-rotated.

Additionally, computes average of CpmPhys.omega ($\bar{\omega}$) and CpmPhys.kappaD ($\bar{\kappa}_D$). N is the number of interactions in the volume given.

Returns tuple of (N, $\boldsymbol{\sigma}$, $\bar{\omega}$, $\bar{\kappa}_D$).

class `yade._eudoxos.`**`InteractionLocator`**

Locate all (real) interactions in space by their contact point. When constructed, all real interactions are spatially indexed (uses vtkPointLocator internally). Use instance methods to use those data.

Note: Data might become inconsistent with real simulation state if simulation is being run between creation of this object and spatial queries.

`bounds`

Return coordinates of lower and uppoer corner of axis-aligned abounding box of all interactions

`count`

Number of interactions held

`intrsAroundPt``(`*(Vector3)point*, *(float)maxDist*`)` → list

Return list of real interactions that are not further than *maxDist* from *point*.

`macroAroundPt``(`*(Vector3)point*, *(float)maxDist*[, *(float)forceVolume=-1*]`)` → tuple

Return tuple of averaged stress tensor (as Matrix3), average omega and average kappa values. *forceVolume* can be used (if positive) rather than the sphere (with *maxDist* radius) volume for the computation. (This is useful if *point* and *maxDist* encompass empty space that you want to avoid.)

`yade._eudoxos.`**`particleConfinement`**`()` → None

`yade._eudoxos.`**`velocityTowardsAxis`**`(`*(Vector3)axisPoint*, *(Vector3)axisDirection*, *(float)timeToAxis*[, *(float)subtractDist*[, *(float)perturbation*]]`)` → None

7.2 yade.export module

Export geometry to various formats.

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.`**`text`**`(`*filename*, *consider=<function <lambda> at 0xa6eadbc>*`)`

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.

***consider*:** anonymous function(optional)

Returns number of spheres which were written.

`yade.export.textExt`(*filename*, *format='x_y_z_r'*, *consider=<function <lambda> at 0xa656f7c>*, *comment=''*)

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*:** the name of output format. Supported *x_y_z_r'(default), 'x_y_z_r_matId*

***comment*:** 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 '

#' for next lines.

***consider*:** anonymous function(optional)

Returns number of spheres which were written.

7.3 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

7.4 yade.log module

Access and manipulation of log4cxx loggers.

`yade.log.loadConfig`(*(str)fileName*) → None
Load configuration from file (log4cxx::PropertyConfigurator::configure)

`yade.log.setLevel`(*(str)logger*, *(int)level*) → None
Set minimum severity *level* (constants `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`) for given logger. Leading 'yade.' will be appended automatically to the logger name; if logger is '', the root logger 'yade' will be operated on.

7.5 yade.pack module

Creating packings and filling volumes defined by boundary representation or constructive solid geometry.

For examples, see

- scripts/test/gts-horse.py
- 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/regular-sphere-pack/regular-sphere-pack.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:

```
>>> O.periodic
True
>>> O.cell.refSize
Vector3(5,5,5)
>>> O.cell.trsf
Matrix3(0.707107,-0.707107,0, 0.707107,0.707107,0, 0,0,1)
```

Parameters

- **rot** (*Quaternion/Matrix3*) – rotation of the packing, which will be applied on spheres and will be used to set Cell.trsf as well.
- ****kw** – passed to utils.sphere

Returns list of body ids added (like O.bodies.append) `yade.pack.filterSpherePack`(*predicate*, *spherePack*, ***kw*)

Using given SpherePack instance, return spheres the satisfy predicate. 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.

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()`

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=True*, *memoDbg=False*, *color=None*)

Generator of random dense packing with given geometry properties, using TriaxialTest (aperiodic) or PeriIsoCompressor (periodic). The priodicity 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)). 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 (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 scripts/test/gts-triax-pack-obb.py.
- **memoDbg** – show packigns that are considered and reasons why they are rejected/accepted

Returns SpherePack object with spheres, filtered by the predicate.

`yade.pack.`**`randomPeriPack`**`(`*radius*, *initSize*, *rRelFuzz=0.0*, *memoizeDb=None*`)`

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

`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.

`load(`*(str)fileName*`)` → None

Load packing from external text file (current data will be discarded).

`makeCloud(`*(Vector3)minCorner*, *(Vector3)maxCorner*[, *(float)rMean=-1*[, *(float)rRelFuzz=0*[, *(int)num=-1*[, *(bool)periodic=False*[, *(float)porosity=-1*[, *(object)psdSizes=[]*[, *(object)psdCumm=[]*[, *(bool)distributeMass=False*]]]]]]]]`)` → int

Create random loose packing enclosed in box. Sphere radius distribution can be specified using one of the following ways (they are mutually exclusive):

1. *rMean* and *rRelFuzz* gives uniform radius distribution between *rMean*(1* ± **rRelFuzz*).
2. *porosity*, *num* and *rRelFuzz* which estimates mean radius so that *porosity* is attained at the end
3. *psdSizes* and *psdCumm*, two arrays specifying points of the particle size distribution function.

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`.

Parameters

- **minCorner** (*Vector3*) – lower corner of the box
- **maxCorner** (*Vector3*) – upper corner of the box
- **rMean** (*float*) – mean radius or spheres
- **rRelFuzz** (*float*) – dispersion of radius relative to rMean
- **num** (*int*) – number of spheres to be generated (if negavite, generate as many as possible, ending after a fixed number of tries to place the sphere in space)
- **periodic** (*bool*) – whether the packing to be generated should be periodic
- **porosity** (*float*) – if specified, estimate mean radius r_m (*rMean* must not be given) using *rRelFuzz* (z) and *num* (N) 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 value of ρ=0.65 is recommended.
- **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.

Returns number of created spheres, which can be lower than *num* is the packing is too tight.

`makeClumpCloud(`*(Vector3)minCorner*, *(Vector3)maxCorner*, *(object)clumps*[, *(bool)periodic=False*[, *(int)num=-1*]]`)` → 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.

`particleSD(`*(Vector3)minCorner*, *(Vector3)maxCorner*, *(float)rMean*, *(bool)periodic=False*, *(str)name*, *(int)numSph*[, *(object)radii=[]*[, *(object)passing=[]*[, *(bool)passingIsNotPercentageButCount=False*]]]`)` → int

Create random packing enclosed in box given by minCorner and maxCorner, containing numSph spheres. Returns number of created spheres, which can be < num if the packing is too tight. The computation is done according to the given psd.

psd([*(int)bins=10*[, *(bool)mass=False*]]) → 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`.

psdScaleExponent

Exponent used to scale cummulative distribution function values so that standard uniform distribution is mapped to uniform mass distribution. Theoretically, this value is 3, but that usually overfavors small particles. The default value is 2.5.

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:

>>> O.periodic True >>> O.cell.refSize Vector3(5,5,5) >>> O.cell.trsf Matrix3(0.707107,-0.707107,0, 0.707107,0.707107,0, 0,0,1)

Parameters

- **rot** (*Quaternion/Matrix3*) – rotation of the packing, which will be applied on spheres and will be used to set Cell.trsf as well.
- ****kw** – passed to utils.sphere

Returns list of body ids added (like O.bodies.append)

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

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`.

__init__(*(object)arg2*, *(object)arg3*) → None

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`.

__init__(*(object)arg2*, *(object)arg3*) → None

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`.

__init__(*(object)arg2*, *(object)arg3*) → None

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`.

__init__(*(object)arg2*, *(object)arg3*) → None

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).

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.

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).

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.

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).

class `yade._packPredicates.`**`inSphere`**(*inherits Predicate*)
Sphere predicate.

`__init__`(*(Vector3)center*, *(float)radius*) → None
Ctor taking center (as a 3-tuple) and radius

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.

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).

class `yade._packSpherePadder.`**`SpherePadder`**
Geometrical algorithm for filling tetrahedral mesh with spheres; the algorithm was designed by Jean-François Jerier and is described in [Jerier2009].

`__init__`(*(str)fileName*[, *(str)meshType=''*]) → None
Initialize using tetrahedral mesh stored in *fileName*. Type of file is determined by extension: .gmsh implies *meshType*='GMSH', .inp implies *meshType*='INP'. If the extension is different, specify *meshType* explicitly. Possible values are 'GMSH' and 'INP'.

`asSpherePack`() → SpherePack

`densify`() → None

`insert_sphere`(*(float)arg2*, *(float)arg3*, *(float)arg4*, *(float)arg5*) → None

`maxNumberOfSpheres`

`maxOverlapRate`

`maxSolidFractioninProbe`

`meanSolidFraction`

`numberOfSpheres`

`pad_5`() → None

`place_virtual_spheres`() → None

`radiusRange`

`radiusRatio`

`save_mgpost`(*(str)arg2*) → None

`setRadiusRatio`(*(float)arg2*, *(float)arg3*) → None
Like radiusRatio, but taking 2nd parameter.

`virtualRadiusFactor`

7.6 yade.plot module

Module containing utility functions for plotting inside yade. See scripts/simple-scene-plot.py or examples/concrete/uniax.py for example of usage.

`yade.plot.data`
: 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`
: dictionary x-name -> (yspec,...), where yspec is either y-name or (y-name,'line-specification')

`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`
: Enable/disable live plot updating. Disabled by default for now, since it has a few rough edges.

`yade.plot.liveInterval`
: Interval for the live plot updating, in seconds.

`yade.plot.autozoom`
: Enable/disable automatic plot rezooming after data update.

`yade.plot.plot(`*noShow=False*, *subPlots=False*`)`
: Do the actual plot, which is either shown on screen (and nothing is returned: if *noShow* is `False`) or, if *noShow* is `True`, returned as matplotlib's Figure object or list of them.

You can use

```
>>> from yade import plot
>>> plot.plots={'foo':('bar',)}
>>> plot.plot(noShow=True).savefig('someFile.pdf')
>>> import os
>>> os.path.exists('someFile.pdf')
True
```

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.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 left-padded with nan's, unspecified data will be nan. This way, equal length of all data is assured so that they can be plotted one against any other.

Nan's don't appear in graphs.

`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*`)`

7.7 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.

7.7.1 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`).

7.7.2 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.

7.7.3 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.

7.8 yade.qt module

7.9 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                        400               9449µs            0.01%
BoundingVolumeMetaEngine             400            1171770µs            1.15%
PersistentSAPCollider                400            9433093µs            9.24%
InteractionGeometryMetaEngine        400           15177607µs           14.87%
InteractionPhysicsMetaEngine         400            9518738µs            9.33%
ConstitutiveLawDispatcher            400           64810867µs           63.49%
  ef2_Spheres_Brefcom_BrefcomLaw
         setup                            4926145            7649131µs           15.25%
         geom                             4926145           23216292µs           46.28%
         material                         4926145            8595686µs           17.14%
         rest                             4926145           10700007µs           21.33%
         TOTAL                                              50161117µs          100.00%
"damper"                             400            1866816µs            1.83%
"strainer"                           400              21589µs            0.02%
"plotDataCollector"                  160              64284µs            0.06%
"damageChecker"                        9               3272µs            0.00%
TOTAL                                             102077490µs          100.00%
```

7.10 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).

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=None*, *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*)

Return average numer of interactions per particle, also known as *coordination number* Z. This number is defined as

$$Z = 2C/N$$

where C is number of contacts and N is number of particles.

With *skipFree*, particles not contributing to stable state of the packing are skipped, following equation (8) given in [Thornton2000]:

$$Z_m = \frac{2C - N_1}{N - N_0 - N_1}$$

Parameters

- **cutoff** – cut some relative part of the sample's bounding box away.
- **skipFree** –

`yade.utils.box`(*center, extents, orientation=[1, 0, 0, 0], dynamic=True, 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=True, wire=False, color=None, highlight=False, material=-1, mask=1*)

Create and chain a MinkCylinder with given parameters. This shape is the Minkowski sum of line 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.

`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:

FrictMat(density=1e3,young=1e7,poisson=.3,frictionAngle=.5,label='defaultMat')

`yade.utils.downCast`(*obj, newClassName*)

Cast given object to class deriving from the same yade root class and copy all parameters from given object. Obj should be up in the inheritance tree, otherwise some attributes may not be defined in the new class.

`yade.utils.facet`(*vertices, dynamic=False, wire=True, color=None, highlight=False, noBound=False, material=-1, mask=1*)

Create facet with given parameters.

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.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 created box

***extents*: (eX,eY,eZ)** 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.utils.facetCylinder`(*center*, *radius*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=7*, *angleRange=None*, *closeGap=False*, ***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
- **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 cylinder 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.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.loadVars`(*mark=None*)

Load variables from 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*, *bps=2400*)

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=…`)
- **bps** (*int*) – Bitrate (`-lavcopts vbitrate=…`)

`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*)
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.

> **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.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=False*, *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:

- *description* column is assigned to Omega().tags['description']; this column is synthesized if absent (see utils.TableParamReader);
- `Omega().tags['params']="name1=val1,name2=val2,…"`
- `Omega().tags['defaultParams']="unassignedName1=defaultValue1,…"`
- `Omega().tags['d.id']=O.tags['id']+'.'+O.tags['description']`
- `Omega().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*:** number of line where to get the values from.

***noTableOk*: bool** do not raise exception if the file cannot be open; use default values

***unknownOk*: bool** do not raise exception if unknown column name is found in the file; 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:

```
>>> from yade import utils
>>> utils.saveVars('mark',a=1,b=2,c=3)
>>> from yade.params.mark 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

```
>>> utils.loadVars('mark')
```

and they will be defined in the yade.params.*mark* module

m*==True, variables will be loaded immediately after saving. That effectively makes ***kw* available in builtin namespace.

`yade.utils.sphere`(*center*, *radius*, *dynamic=True*, *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
- **Vector3-or-None** – body's color, as normalized RGB; random color will be assigned if ‘‘None‘.
- **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

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.`**`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*, ***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: number of section that will be used to estimate cross-section

axis: 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.`**`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

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.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.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.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.

This function will very likely behave incorrectly for periodic simulations (though it could be extended it to handle it farily easily).

`yade._utils.elasticEnergy(`*(tuple)arg1*`)` → float

`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`.

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.getSpheresVolume()` → float

Compute the total volume of spheres in the simulation (might crash for now if dynamic bodies are not spheres)

`yade._utils.getViscoelasticFromSpheresInteraction(`*(float)tc*, *(float)en*, *(float)es*`)` → dict

Get viscoelastic interaction parameters from analytical solution of a pair spheres collision problem.

Parameters

'm' : float sphere mass 'tc' : float collision time 'en' : float normal restitution coefficient 'es' : float tangential restitution coefficient.

Returns

dict with keys:

kn : float normal elastic coefficient computed as:

$$k_n = \frac{m}{t_c^2}\left(\pi^2 + (\ln e_n)^2\right)$$

cn : float normal viscous coefficient computed as:

$$c_n = -\frac{2m}{t_c}\ln e_n$$

kt : float tangential elastic coefficient computed as:

$$k_t = \frac{2}{7}\frac{m}{t_c^2}\left(\pi^2 + (\ln e_t)^2\right)$$

ct : float tangential viscous coefficient computed as:

$$c_t = -\frac{2}{7}\frac{m}{t_c}\ln e_t.$$

For details see [Pournin2001].

`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*]]]`)` → tuple

`yade._utils.kineticEnergy(`[*(bool)findMaxId=False*]`)` → object

Compute overall kinetic energy of the simulation as

$$\sum \frac{1}{2}\left(m_i v_i^2 + \omega(\mathbf{I}\omega^T)\right).$$

No transformation of inertia tensor (in local frame) $\mathbf{I}$ is done, although it is multiplied by angular velocity $\boldsymbol{\omega}$ (in global frame); the value will not be accurate for aspherical particles.

`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.`**`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*]`)` → 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):

$$\mathbf{sigma}_{ij} = \frac{2}{V}\sum RN\mathbf{n}_i\mathbf{n}_j + \frac{2}{V}\sum RT\mathbf{n}_i\mathbf{t}_j$$

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.

`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.:param float volume: overall volume which must be specified for aperiodic simulations. For periodic simulations, current volume of the Cell is used.

`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.`**`setRefSe3`**`()` → None

Set reference positions and orientations of all bodies equal to their current positions and orientations.

`yade._utils.`**`spiralProject`**`(`*(Vector3)pt*, *(float)dH_dTheta*[, *(int)axis=2*[, *(float)periodStart=nan*[, *(float)theta0=0*]]]`)` → tuple

`yade._utils.`**`stressTensorOfPeriodicCell`**`(`[*(bool)smallStrains=False*]`)` → Matrix3

Compute overall (macroscopic) stress of periodic cell using equation published in [Kuhl2001]:

$$\boldsymbol{\sigma} = \frac{1}{V}\sum_{c} l^c[\mathbf{N}^c f_N^c + \mathbf{T}^{c^T}\cdot \mathbf{f}_T^c],$$

where V is volume of the cell, l^c length of interaction c, f_N^c normal force and $\mathbf{f}_T^c$ shear force. Sumed are values over all interactions c. $\mathbf{N}^c$ and $\mathbf{T}^{c^T}$ are projection tensors (see the original publication for more details):

$$\mathbf{N} = \mathbf{n}\otimes\mathbf{n} \rightarrow N_{ij} = n_i n_j$$

$$\mathbf{T}^T = \mathbf{I}_{sym}\cdot\mathbf{n} - \mathbf{n}\otimes\mathbf{n}\otimes\mathbf{n} \rightarrow T^T_{ijk} = \frac{1}{2}(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk})n_l - n_i n_j n_k$$

$$\mathbf{T}^T \cdot \mathbf{f}_T \equiv T^T_{ijk} f_k = (\delta_{ik} n_j/2 + \delta_{jk} n_i/2 - n_i n_j n_k) f_k = n_j f_i/2 + n_i f_j/2 - n_i n_j n_k f_k,$$

where n is unit vector oriented along the interaction (normal) and δ is Kronecker's delta. As $\mathbf{n}$ and $\mathbf{f}_T$ are perpendicular (therfore $n_i f_i = 0$) we can write

$$\sigma_{ij} = \frac{1}{V} \sum l [n_i n_j f_N + n_j f^T_i/2 + n_i f^T_j/2]$$

Parameters smallStrains (*bool*) – if false (large strains), real values of volume and interaction lengths are computed. If true, only refLength of interactions and initial volume are computed (can save some time).

Returns macroscopic stress tensor as Matrix3

`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`**`(`*(tuple)ids*, *(Vector3)direction*`)` → float

Return summary force on bodies with given *ids*, projected on the *direction* vector.

`yade._utils.`**`sumTorques`**`(`*(tuple)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 maximum 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.`**`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.

7.11 yade.ymport module

Import geometry from various formats ('import' is python keyword, hence the name 'ymport').

`yade.ymport.`**`gengeo`**`(`*mntable*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, ***kw*`)`

Imports geometry from LSMGenGeo library and creates spheres.

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/regular-sphere-pack/regular-sphere-pack.py, usage of LSMGenGeo library in scripts/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.

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/regular-sphere-pack/regular-sphere-pack.py, usage of LSMGenGeo library in scripts/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.stl`(*file*, *dynamic=False*, *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, create spheres, insert them to the simulation.

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.textExt`(*fileName*, *format='x_y_z_r'*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, ***kw*)

Load sphere coordinates from file in specific format, create spheres, insert them to the simulation.

Parameters *filename*: string *format*:

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

Chapter 8

External modules

8.1 miniEigen (math) module

Basic math functions for Yade: small matrix, vector and quaternion classes. This module internally wraps small parts of the Eigen library. Refer to its documentation for details. All classes in this module support pickling.

class `miniEigen.Matrix3`

3x3 float matrix.

Supported operations (`m` is a Matrix3, `f` if a float/int, `v` is a Vector3): `-m`, `m+m`, `m+=m`, `m-m`, `m-=m`, `m*f`, `f*m`, `m*=f`, `m/f`, `m/=f`, `m*m`, `m*=m`, `m*v`, `v*m`, `m==m`, `m!=m`.

`__init__``()` → None

___init___((Matrix3)m) → None

___init___((float)m00, (float)m01, (float)m02, (float)m10, (float)m11, (float)m12, (float)m20, (float)m21, (float)m22) → object

`determinant``()` → float

`diagonal``()` → Vector3

`inverse``()` → Matrix3

`polarDecomposition``()` → tuple

`toVoigt``(`$\Big[$*(bool)strain=False*$\Big]$`)` → Vector6

Convert 2nd order tensor to 6-vector (Voigt notation), symmetrizing the tensor; if *strain* is `True`, multiply non-diagonal compoennts by 2.

`trace``()` → float

`transpose``()` → Matrix3

class `miniEigen.Quaternion`

Quaternion representing rotation.

Supported operations (`q` is a Quaternion, `v` is a Vector3): `q*q` (rotation composition), `q*=q`, `q*v` (rotating `v` by `q`), `q==q`, `q!=q`.

`Rotate``(`*(Vector3)v*`)` → Vector3

`__init__``()` → None

___init___((Vector3)axis, (float)angle) → object

___init___((float)angle, (Vector3)axis) → object

___init___((float)w, (float)x, (float)y, (float)z) → None : Initialize from coefficients.

Note: The order of coefficients is w, x, y, z. The [] operator numbers them differently, 0...4 for x y z w!

___init___((Quaternion)other) → None

conjugate() → Quaternion

inverse() → Quaternion

norm() → float

normalize() → None

setFromTwoVectors(*(Vector3)u*, *(Vector3)v*) → Quaternion

toAngleAxis() → tuple

toAxisAngle() → tuple

toRotationMatrix() → Matrix3

class miniEigen.**Vector2**

3-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector3): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list,tuple, …) of 2 floats.

__init__() → None

___init___((Vector2)other) → None

___init___((float)x, (float)y) → None

dot(*(Vector2)arg2*) → float

norm() → float

normalize() → None

squaredNorm() → float

class miniEigen.**Vector2i**

2-dimensional integer vector.

Supported operations (`i` if an int, `v` is a Vector2i): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*i`, `i*v`, `v*=i`, `v==v`, `v!=v`.

Implicit conversion from sequence (list,tuple, …) of 2 integers.

__init__() → None

___init___((Vector2i)other) → None

___init___((int)x, (int)y) → None

dot(*(Vector2i)arg2*) → float

norm() → int

normalize() → None

squaredNorm() → int

class miniEigen.**Vector3**

3-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector3): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`, plus operations with `Matrix3` and `Quaternion`.

Implicit conversion from sequence (list,tuple, …) of 3 floats.

__init__() → None

___init___((Vector3)other) → None

___init___((float)x, (float)y, (float)z) → None

cross(*(Vector3)arg2*) → Vector3

dot(*(Vector3)arg2*) → float

norm() → float

normalize() → None

normalized() → Vector3

squaredNorm() → float

class `miniEigen.Vector3i`
3-dimensional integer vector.

Supported operations (`i` if an int, `v` is a Vector3i): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*i`, `i*v`, `v*=i`, `v==v`, `v!=v`.

Implicit conversion from sequence (list,tuple, …) of 3 integers.

__init__() → None
___init___((Vector3i)other) → None

___init___((int)x, (int)y, (int)z) → None

cross(*(Vector3i)arg2*) → Vector3i

dot(*(Vector3i)arg2*) → float

norm() → int

squaredNorm() → int

class `miniEigen.Vector6`
6-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector6): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list,tuple, …) of 6 floats.

__init__() → None
___init___((Vector6)other) → None

___init___((float)v0, (float)v1, (float)v2, (float)v3, (float)v4, (float)v5) → object

norm() → float

normalize() → None

normalized() → Vector6

squaredNorm() → float

toSymmTensor([*(bool)strain=False*]) → Matrix3
Convert Vector6 in the Voigt notation to the corresponding 2nd order symmetric tensor (as Matrix3); if *strain* is `True`, multiply non-diagonal components by .5

class `miniEigen.Vector6i`
6-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector6): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list,tuple, …) of 6 floats.

__init__() → None
___init___((Vector6i)other) → None

___init___((int)v0, (int)v1, (int)v2, (int)v3, (int)v4, (int)v5) → object

norm() → int

normalize() → None

normalized() → Vector6i

squaredNorm() → int

8.2 gts (GNU Triangulated surface) module

A package for constructing and manipulating triangulated surfaces.

PyGTS is a python binding for the GNU Triangulated Surface (GTS) Library, which may be used to build, manipulate, and perform computations on triangulated surfaces.

The following geometric primitives are provided:

> Point - a point in 3D space Vertex - a Point in 3D space that may be used to define a Segment Segment - a line defined by two Vertex end-points Edge - a Segment that may be used to define the edge of a Triangle Triangle - a triangle defined by three Edges Face - a Triangle that may be used to define a face on a Surface Surface - a surface composed of Faces

A tetrahedron is assembled from these primitives as follows. First, create Vertices for each of the tetrahedron's points:

> import gts
>
> v1 = gts.Vertex(1,1,1) v2 = gts.Vertex(-1,-1,1) v3 = gts.Vertex(-1,1,-1) v4 = gts.Vertex(1,-1,-1)

Next, connect the four vertices to create six unique Edges:

> e1 = gts.Edge(v1,v2) e2 = gts.Edge(v2,v3) e3 = gts.Edge(v3,v1) e4 = gts.Edge(v1,v4) e5 = gts.Edge(v4,v2) e6 = gts.Edge(v4,v3)

The four triangular faces are composed using three edges each:

> f1 = gts.Face(e1,e2,e3) f2 = gts.Face(e1,e4,e5) f3 = gts.Face(e2,e5,e6) f4 = gts.Face(e3,e4,e6)

Finally, the surface is assembled from the faces:

> s = gts.Surface() for face in [f1,f2,f3,f4]:
>
> > s.add(face)

Some care must be taken in the orientation of the faces. In the above example, the surface normals are pointing inward, and so the surface technically defines a void, rather than a solid. To create a tetrahedron with surface normals pointing outward, use the following instead:

> f1.revert() s = Surface() for face in [f1,f2,f3,f4]:
>
> > **if not face.is_compatible(s):** face.revert()
> >
> > s.add(face)

Once the Surface is constructed, there are many different operations that can be performed. For example, the volume can be calculated using:

> s.volume()

The difference between two Surfaces s1 and s2 is given by:

> s3 = s2.difference(s1)

Etc.

It is also possible to read in GTS data files and plot surfaces to the screen. See the example programs packaged with PyGTS for more information.

class `gts.Edge`(*inherits Segment → Object → object*)

Bases: `gts.Segment`

Edge object

`__init__`

x.___init___(...) initializes x; see help(type(x)) for signature

`belongs_to_tetrahedron`

Returns True if this Edge e belongs to a tetrahedron. Otherwise False.

Signature: e.belongs_to_tetrahedron()

contacts
Returns number of sets of connected triangles share this Edge e as a contact Edge.

Signature: e.contacts()

face_number
Returns number of faces using this Edge e on Surface s.

Signature: e.face_number(s)

is_boundary
Returns True if this Edge e is a boundary on Surface s. Otherwise False.

Signature: e.is_boundary(s)

is_ok
True if this Edge e is not degenerate or duplicate. False otherwise. Degeneracy implies e.v1.id == e.v2.id.

Signature: e.is_ok()

is_unattached
True if this Edge e is not part of any Triangle.

Signature: e.is_unattached()

class `gts.Face`(*inherits Triangle → Object → object*)
Bases: `gts.Triangle`

Face object

__init__
x.___init___(...) initializes x; see help(type(x)) for signature

is_compatible
True if Face f is compatible with all neighbors in Surface s. False otherwise.

Signature: f.is_compatible(s).

is_ok
True if this Face f is non-degenerate and non-duplicate. False otherwise.

Signature: f.is_ok()

is_on
True if this Face f is on Surface s. False otherwise.

Signature: f.is_on(s).

is_unattached
True if this Face f is not part of any Surface.

Signature: f.is_unattached().

neighbor_number
Returns the number of neighbors of Face f belonging to Surface s.

Signature: f.neighbor_number(s).

neighbors
Returns a tuple of neighbors of this Face f belonging to Surface s.

Signature: f.neighbors(s).

class `gts.Object`(*inherits object*)
Bases: `object`

Base object

__init__
x.___init___(...) initializes x; see help(type(x)) for signature

id
GTS object id

is_unattached
True if this Object o is not attached to another Object. Otherwise False.

Trace: o.is_unattached().

class **gts.Point**(*inherits Object → object*)
Bases: gts.Object

Point object

__init__
x.___init___(...) initializes x; see help(type(x)) for signature

closest
Set the coordinates of Point p to the Point on Segment s or Triangle t closest to the Point p2

Signature: p.closest(s,p2) or p.closest(t,p2)

Returns the (modified) Point p.

coords
Returns a tuple of the x, y, and z coordinates for this Point p.

Signature: p.coords(x,y,z)

distance
Returns Euclidean distance between this Point p and other Point p2, Segment s, or Triangle t. Signature: p.distance(p2), p.distance(s) or p.distance(t)

distance2
Returns squared Euclidean distance between Point p and Point p2, Segment s, or Triangle t.

Signature: p.distance2(p2), p.distance2(s), or p.distance2(t)

is_in
Tests if this Point p is inside or outside Triangle t. The planar projection (x,y) of Point p is tested against the planar projection of Triangle t.

Signature: p.in__circle(p1,p2,p3) or p.in__circle(t)

Returns a +1 if p lies inside, -1 if p lies outside, and 0 if p lies on the triangle.

is_in_circle
Tests if this Point p is inside or outside circumcircle. The planar projection (x,y) of Point p is tested against the circumcircle defined by the planar projection of p1, p2 and p3, or alternatively the Triangle t

Signature: p.in__circle(p1,p2,p3) or p.in__circle(t)

Returns +1 if p lies inside, -1 if p lies outside, and 0 if p lies on the circle. The Points p1, p2, and p3 must be in counterclockwise order, or the sign of the result will be reversed.

is_in_rectangle
True if this Point p is in box with bottom-left and upper-right Points p1 and p2.

Signature: p.is__in__rectange(p1,p2)

is_inside
True if this Point p is inside or outside Surface s. False otherwise.

Signature: p.in__inside(s)

is_ok
True if this Point p is OK. False otherwise. This method is useful for unit testing and debugging.

Signature: p.is__ok().

orientation_3d
Determines if this Point p is above, below or on plane of 3 Points p1, p2 and p3.

Signature: p.orientation__3d(p1,p2,p3)

Below is defined so that p1, p2 and p3 appear in counterclockwise order when viewed from above the plane.

The return value is positive if p4 lies below the plane, negative if p4 lies above the plane, and zero if the four points are coplanar. The value is an approximation of six times the signed volume of the tetrahedron defined by the four points.

`orientation_3d_sos`

Determines if this Point p is above, below or on plane of 3 Points p1, p2 and p3.

Signature: p.orientation_3d_sos(p1,p2,p3)

Below is defined so that p1, p2 and p3 appear in counterclockwise order when viewed from above the plane.

The return value is +1 if p4 lies below the plane, and -1 if p4 lies above the plane. Simulation of Simplicity (SoS) is used to break ties when the orientation is degenerate (i.e. the point lies on the plane definedby p1, p2 and p3).

`rotate`

Rotates Point p around vector dx,dy,dz by angle a. The sense of the rotation is given by the right-hand-rule.

Signature: p.rotate(dx=0,dy=0,dz=0,a=0)

`scale`

Scales Point p by vector dx,dy,dz.

Signature: p.scale(dx=1,dy=1,dz=1)

`set`

Sets x, y, and z coordinates of this Point p.

Signature: p.set(x,y,z)

`translate`

Translates Point p by vector dx,dy,dz.

Signature: p.translate(dx=0,dy=0,dz=0)

`x`

x value

`y`

y value

`z`

z value

class `gts.Segment`(*inherits Object → object*)

Bases: `gts.Object`

Segment object

`__init__`

x.___init___(...) initializes x; see help(type(x)) for signature

`connects`

Returns True if this Segment s1 connects Vertices v1 and v2. False otherwise.

Signature: s1.connects(v1,v2).

`intersection`

Returns the intersection of Segment s with Triangle t

This function is geometrically robust in the sense that it will return None if s and t do not intersect and will return a Vertex if they do. However, the point coordinates are subject to round-off errors. None will be returned if s is contained in the plane defined by t.

Signature: s.intersection(t) or s.intersection(t,boundary).

If boundary is True (default), the boundary of s is taken into account.

Returns a summit of t (if boundary is True), one of the endpoints of s, a new Vertex at the intersection of s with t, or None if s and t don't intersect.

intersects

Checks if this Segment s1 intersects with Segment s2. Returns 1 if they intersect, 0 if an endpoint of one Segment lies on the other Segment, -1 otherwise

Signature: s1.intersects(s2).

is_ok

True if this Segment s is not degenerate or duplicate. False otherwise. Degeneracy implies s.v1.id == s.v2.id.

Signature: s.is_ok().

midvertex

Returns a new Vertex at the mid-point of this Segment s.

Signature: s.midvertex().

touches

Returns True if this Segment s1 touches Segment s2 (i.e., they share a common Vertex). False otherwise.

Signature: s1.touches(s2).

v1

Vertex 1

v2

Vertex 2

class gts.**Surface**(*inherits Object → object*)

Bases: `gts.Object`

Surface object

Nedges

The number of unique edges

Nfaces

The number of unique faces

Nvertices

The number of unique vertices

__init__

x.___init___(...) initializes x; see help(type(x)) for signature

add

Adds a Face f or Surface s2 to Surface s1.

Signature: s1.add(f) or s2.add(f)

area

Returns the area of Surface s. The area is taken as the sum of the signed areas of the Faces of s.

Signature: s.area()

boundary

Returns a tuple of boundary Edges of Surface s.

Signature: s.boundary()

center_of_area

Returns the coordinates of the center of area of Surface s.

Signature: s.center_of_area()

center_of_mass

Returns the coordinates of the center of mass of Surface s.

Signature: s.center_of_mass()

cleanup

Cleans up the Vertices, Edges, and Faces on a Surface s.

Signature: s.cleanup() or s.cleanup(threhold)

If threhold is given, then Vertices that are spaced less than the threshold are merged. Degenerate Edges and Faces are also removed.

coarsen

Reduces the number of vertices on Surface s.

Signature: s.coarsen(n) and s.coarsen(amin)

n is the smallest number of desired edges (but you may get fewer). amin is the smallest angle between Faces.

copy

Copys all Faces, Edges and Vertices of Surface s2 to Surface s1.

Signature: s1.copy(s2)

Returns s1.

difference

Returns the difference of this Surface s1 with Surface s2.

Signature: s1.difference(s2)

distance

Calculates the distance between the faces of this Surface s1 and the nearest Faces of other s2, and (if applicable) the distance between the boundary of this Surface s1 and the nearest boundary Edges of other s2.

One or two dictionaries are returned (where applicable), the first for the face range and the second for the boundary range. The fields in each dictionary describe statistical results for each population: {min,max,sum,sum2,mean,stddev,n}.

Signature: s1.distance(s2) or s1.distance(s2,delta)

The value delta is a spatial increment defined as the percentage of the diagonal of the bounding box of s2 (default 0.1).

edges

Returns tuple of Edges on Surface s that have Vertex in list. If a list is not given then all of the Edges are returned.

Signature: s.edges(list) or s.edges()

face_indices

Returns a tuple of 3-tuples containing Vertex indices for each Face in Surface s. The index for each Vertex in a face corresponds to where it is found in the Vertex tuple vs.

Signature: s.face_indices(vs)

faces

Returns tuple of Faces on Surface s that have Edge in list. If a list is not given then all of the Faces are returned.

Signature: s.faces(list) s.faces()

fan_oriented

Returns a tuple of outside Edges of the Faces fanning from Vertex v on this Surface s. The Edges are given in counter-clockwise order.

Signature: s.fan_oriented(v)

intersection

Returns the intersection of this Surface s1 with Surface s2.

Signature: s1.intersection(s2)

is_closed

True if Surface s is closed, False otherwise. Note that a closed Surface is also a manifold.

Signature: s.is_closed()

is_manifold

True if Surface s is a manifold, False otherwise.

Signature: s.is_manifold()

is_ok

True if this Surface s is OK. False otherwise.

Signature: s.is_ok()

is_orientable

True if Faces in Surface s have compatible orientation, False otherwise. Note that a closed surface is also a manifold. Note that an orientable surface is also a manifold.

Signature: s.is_orientable()

is_self_intersecting

Returns True if this Surface s is self-intersecting. False otherwise.

Signature: s.is_self_intersecting()

manifold_faces

Returns the 2 manifold Faces of Edge e on this Surface s if they exist, or None.

Signature: s.manifold_faces(e)

next

x.next() -> the next value, or raise StopIteration

parent

Returns Face on this Surface s that has Edge e, or None if the Edge is not on this Surface.

Signature: s.parent(e)

quality_stats

Returns quality statistics for this Surface f in a dict. The statistics include the {min, max, sum, sum2, mean, stddev, and n} for populations of face_quality, face_area, edge_length, and edge_angle. Each of these names are dictionary keys. See Triangle.quality() for an explanation of the face_quality.

Signature: s.quality_stats()

remove

Removes Face f from this Surface s.

Signature: s.remove(f)

rotate

Rotates Surface s about vector dx,dy,dz and angle a. The sense of the rotation is given by the right-hand-rule.

Signature: s.rotate(dx,dy,dz,a)

scale

Scales Surface s by vector dx,dy,dz.

Signature: s.scale(dx=1,dy=1,dz=1)

split

Splits a surface into a tuple of connected and manifold components.

Signature: s.split()

stats

Returns statistics for this Surface f in a dict. The stats include n_faces, n_incompatible_faces,, n_boundary_edges, n_non_manifold_edges, and the statisics {min, max, sum, sum2, mean, stddev, and n} for populations of edges_per_vertex and faces_per_edge. Each of these names are dictionary keys.

Signature: s.stats()

strip
Returns a tuple of strips, where each strip is a tuple of Faces that are successive and have one edge in common.

Signature: s.split()

tessellate
Tessellate each face of this Surface s with 4 triangles. The number of triangles is increased by a factor of 4.

Signature: s.tessellate()

translate
Translates Surface s by vector dx,dy,dz.

Signature: s.translate(dx=0,dy=0,dz=0)

union
Returns the union of this Surface s1 with Surface s2.

Signature: s1.union(s2)

vertices
Returns a tuple containing the vertices of Surface s.

Signature: s.vertices()

volume
Returns the signed volume of the domain bounded by the Surface s.

Signature: s.volume()

write
Saves Surface s to File f in GTS ascii format. All the lines beginning with #! are ignored.

Signature: s.write(f)

write_oogl
Saves Surface s to File f in OOGL (Geomview) format.

Signature: s.write_oogl(f)

write_oogl_boundary
Saves boundary of Surface s to File f in OOGL (Geomview) format.

Signature: s.write_oogl_boundary(f)

write_vtk
Saves Surface s to File f in VTK format.

Signature: s.write_vtk(f)

class gts.Triangle(*inherits Object → object*)
Bases: gts.Object

Triangle object

__init__
x.___init___(...) initializes x; see help(type(x)) for signature

angle
Returns the angle (radians) between Triangles t1 and t2

Signature: t1.angle(t2)

area
Returns the area of Triangle t.

Signature: t.area()

`circumcenter`
Returns a Vertex at the center of the circumscribing circle of this Triangle t, or None if the circumscribing circle is not defined.

Signature: t.circumcircle_center()

`common_edge`
Returns Edge common to both this Triangle t1 and other t2. Returns None if the triangles do not share an Edge.

Signature: t1.common_edge(t2)

`e1`
Edge 1

`e2`
Edge 2

`e3`
Edge 3

`interpolate_height`
Returns the height of the plane defined by Triangle t at Point p. Only the x- and y-coordinates of p are considered.

Signature: t.interpolate_height(p)

`is_compatible`
True if this triangle t1 and other t2 are compatible; otherwise False.

Checks if this triangle t1 and other t2, which share a common Edge, can be part of the same surface without conflict in the surface normal orientation.

Signature: t1.is_compatible(t2)

`is_ok`
True if this Triangle t is non-degenerate and non-duplicate. False otherwise.

Signature: t.is_ok()

`is_stabbed`
Returns the component of this Triangle t that is stabbed by a ray projecting from Point p to z=infinity. The result can be this Triangle t, one of its Edges or Vertices, or None. If the ray is contained in the plan of this Triangle then None is also returned.

Signature: t.is_stabbed(p)

`normal`
Returns a tuple of coordinates of the oriented normal of Triangle t as the cross-product of two edges, using the left-hand rule. The normal is not normalized. If this triangle is part of a closed and oriented surface, the normal points to the outside of the surface.

Signature: t.normal()

`opposite`
Returns Vertex opposite to Edge e or Edge opposite to Vertex v for this Triangle t.

Signature: t.opposite(e) or t.opposite(v)

`orientation`
Determines orientation of the plane (x,y) projection of Triangle t

Signature: t.orientation()

Returns a positive value if Points p1, p2 and p3 in Triangle t appear in counterclockwise order, a negative value if they appear in clockwise order and zero if they are colinear.

`perimeter`
Returns the perimeter of Triangle t.

Signature: t.perimeter()

quality
Returns the quality of Triangle t.

The quality of a triangle is defined as the ratio of the square root of its surface area to its perimeter relative to this same ratio for an equilateral triangle with the same area. The quality is then one for an equilateral triangle and tends to zero for a very stretched triangle. Signature: t.quality()

revert
Changes the orientation of triangle t, turning it inside out.

Signature: t.revert()

vertex
Returns the Vertex of this Triangle t not in t.e1.

Signature: t.vertex()

vertices
Returns the three oriented set of vertices in Triangle t.

Signature: t.vertices()

class gts.**Vertex**(*inherits Point → Object → object*)
Bases: gts.Point

Vertex object

__init__
x.___init___(...) initializes x; see help(type(x)) for signature

contacts
Returns the number of sets of connected Triangles sharing this Vertex v.

Signature: v.contacts().

If sever is True (default: False) and v is a contact vertex then the vertex is replaced in each Triangle with clones.

encroaches
Returns True if this Vertex v is strictly contained in the diametral circle of Edge e. False otherwise.

Only the projection onto the x-y plane is considered.

Signature: v.encroaches(e)

faces
Returns a tuple of Faces that have this Vertex v.

If a Surface s is given, only Vertices on s are considered.

Signature: v.faces() or v.faces(s).

is_boundary
True if this Vertex v is used by a boundary Edge of Surface s.

Signature: v.is_boundary().

is_connected
Return True if this Vertex v1 is connected to Vertex v2 by a Segment.

Signature: v1.is_connected().

is_ok
True if this Vertex v is OK. False otherwise. This method is useful for unit testing and debugging.

Signature: v.is_ok().

is_unattached
True if this Vertex v is not the endpoint of any Segment.

Signature: v.is_unattached().

neighbors

Returns a tuple of Vertices attached to this Vertex v by a Segment.

If a Surface s is given, only Vertices on s are considered.

Signature: v.neighbors() or v.neighbors(s).

replace

Replaces this Vertex v1 with Vertex v2 in all Segments that have v1. Vertex v1 itself is left unchanged.

Signature: v1.replace(v2).

triangles

Returns a list of Triangles that have this Vertex v.

Signature: v.triangles()

Chapter 9

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 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.

Note: This file is generated from doc/yade-articles.bib, doc/yade-conferences.bib and doc/yade-theses.bib.

9.1 Journal articles

9.2 Master and PhD theses

9.3 Conference materials

Chapter 10

References

All external articles referenced in Yade documentation.

Note: This file is generated from doc/references.bib.

Chapter 11

Indices and tables

- *genindex*
- *modindex*
- *search*

Bibliography

[gcc] http://gcc.gnu.org

[boost] http://www.boost.org

[ipython] http://ipython.scipy.org

[matplotlib] http://matplotlib.sf.net

[numpy] http://numpy.scipy.org

[scons] http://www.scons.org

[cgal] http://www.cgal.org

[gts] http://gts.sourceforge.net

[eigen] http://eigen.tuxfamily.org

[log4cxx] http://log4cxx.apache.org

[vtk] http://www.vtk.org

[epydoc] http://epydoc.sourceforge.net

[doxygen] http://www.doxygen.org

[python] http://www.python.org

[rest] http://docutils.sourceforge.net/rst.html

[sphinx] http://sphinx.pocoo.org

[OpenMP] http://www.openmp.org

[wm3] http://www.geometrictools.com/

[QGLViewer] http://www.libqglviewer.com/

[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.

[Chen2007a] Feng Chen, Eric. C. Drumm, Georges Guiochon (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)

[Dang2010] H. K. Dang, M. A. Meguid (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

[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.

[Duriez2010] J. Duriez, F.Darve, F.-V.Donze (2010), **A discrete modeling-based constitutive relation for infilled rock joints**. *International Journal of Rock Mechanics & Mining Sciences.* (in press)

[Harthong2009] Harthong, B., Jerier, J. F., Dorémus, 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

[Hassan2010] A. Hassan, B. Chareyre, F. Darve, J. Meyssonier, F. Flin (2010 (submitted)), **Microtomography-based discrete element modelling of creep in snow**. *Granular Matter.*

[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.

[Jerier2009] Jerier, Jean-François, Imbault, Didier, Donzé, Fréederic-Victor, Doremus, Pierre (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

[Jerier2010] Jerier, Jean-François, Richefeu, Vincent, Imbault, Didier, Donzé, Fréderic-Victor (2010), **Packing spherical discrete elements for large scale simulations**. *Computer Methods in Applied Mechanics and Engineering.* DOI 10.1016/j.cma.2010.01.016

[Kozicki2005a] J. Kozicki (2005), **Discrete lattice model used to describe the fracture process of concrete**. *Discrete Element Group for Risk Mitigation Annual Report 1, Grenoble University of Joseph Fourier, France*, pages 95–101. (fulltext)

[Kozicki2006a] J. Kozicki, J. Tejchman (2006), **2d lattice model for fracture in brittle materials**. *Archives of Hydro-Engineering and Environmental Mechanics* (53), pages 71–88. (fulltext)

[Kozicki2007a] J. Kozicki, J. Tejchman (2007), **Effect of aggregate structure on fracture process in concrete using 2d lattice model"**. *Archives of Mechanics* (59), pages 365–384. (fulltext)

[Kozicki2008] J. Kozicki, F.V. Donzé (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] J. Kozicki, F.V. Donzé (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)

[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

[Nicot2007a] Nicot, F., L. Sibille, F.V. Donzé, F. Darve (2007), **From microscopic to macroscopic second-order work in granular assemblies**. *Int. J. Mech. Mater.* (39), pages 664–684. DOI 10.1016/j.mechmat.2006.10.003

[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

[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

[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. DOI ‘ <http://dx.doi.org/>‘_

[Sibille2007a] L. Sibille, F. Nicot, F.V. Donzé, F. Darve (2007), **Material instability in granular assemblies from fundamentally different models**. *International Journal For Numerical and Analytical Methods in Geomechanics* (31), pages 457–481. DOI 10.1002/nag.591

[Sibille2008a] L. Sibille, F.-V. Donzé, F. Nicot, B. Chareyre, F. Darve (2008), **From bifurcation to failure in a granular material: a dem analysis**. *Acta Geotechnica* (3), pages 15–24. DOI 10.1007/s11440-007-0035-y

[Smilauer2006] Václav Šmilauer (2006), **The splendors and miseries of yade design**. *Annual Report of Discrete Element Group for Hazard Mitigation*. (fulltext)

[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)

[Duriez2009a] J. Duriez (2009), **Stabilité des massifs rocheux : une approche mécanique**. PhD thesis at *Institut polytechnique de Grenoble*. (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)

[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)

[Chareyre2009] Chareyre B.„ Scholtès L. (2009), **Micro-statics and micro-kinematics of capillary phenomena in dense granular materials**. In *Powders and Grains 2009 (Golden, USA)*.

[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*.

[Chen2009] 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*.

[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)*.

[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)*.

[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.*

[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.*

[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.*

[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.*

[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)

[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-Marroqu?n, R. Garc?a-Rojo, H. J. Herrmann (2004), **Micro-mechanical investigation of the granular ratcheting**. In *Cyclic Behaviour of Soils and Liquefaction Phenomena.* (fulltext)

[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), **Mod?lisation du comportement d'ouvrages composites sol-g?osynth?tique par ?l?ments discrets - application aux tranch?es d'ancrage en t?te 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

[DeghmReport2006] F. V. Donz? (ed.), **Annual report 2006** (2006). *Discrete Element Group for Hazard Mitigation.* Universit? Joseph Fourier, Grenoble (fulltext)

[Duriez2010] J. Duriez, F.Darve, F.-V.Donze (2010), **A discrete modeling-based constitutive relation for infilled rock joints**. *International Journal of Rock Mechanics & Mining Sciences.* (in press)

[GarciaRojo2004] R. Garc?a-Rojo, S. McNamara, H. J. Herrmann (2004), **Discrete element methods for the micro-mechanical investigation of granular ratcheting**. In *Proceedings ECCOMAS 2004.* (fulltext)

[Hassan2010] A. Hassan, B. Chareyre, F. Darve, J. Meyssonier, F. Flin (2010 (submitted)), **Microtomography-based discrete element modelling of creep in snow**. *Granular Matter.*

[Hentz2003] S?ebastien Hentz (2003), **Mod?lisation d'une structure en b?ton arm? soumise ? un choc par la m?thode des el?ments discrets**. PhD thesis at *Universit? Grenoble 1 – Joseph Fourier.*

[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

[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

[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)

[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)

[Luding2008] Stefan Luding (2008), **Introduction to discrete element methods**. In *European Journal of Environmental and Civil Engineering.*

[McNamara2008] S. McNamara, R. Garc?a-Rojo, H. J. Herrmann (2008), **Microscopic origin of granular ratcheting**. *Physical Review E* (77). DOI 11.1103/PhysRevE.77.031304

[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.

[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)

[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**. *G?otechnique* (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)

Python Module Index