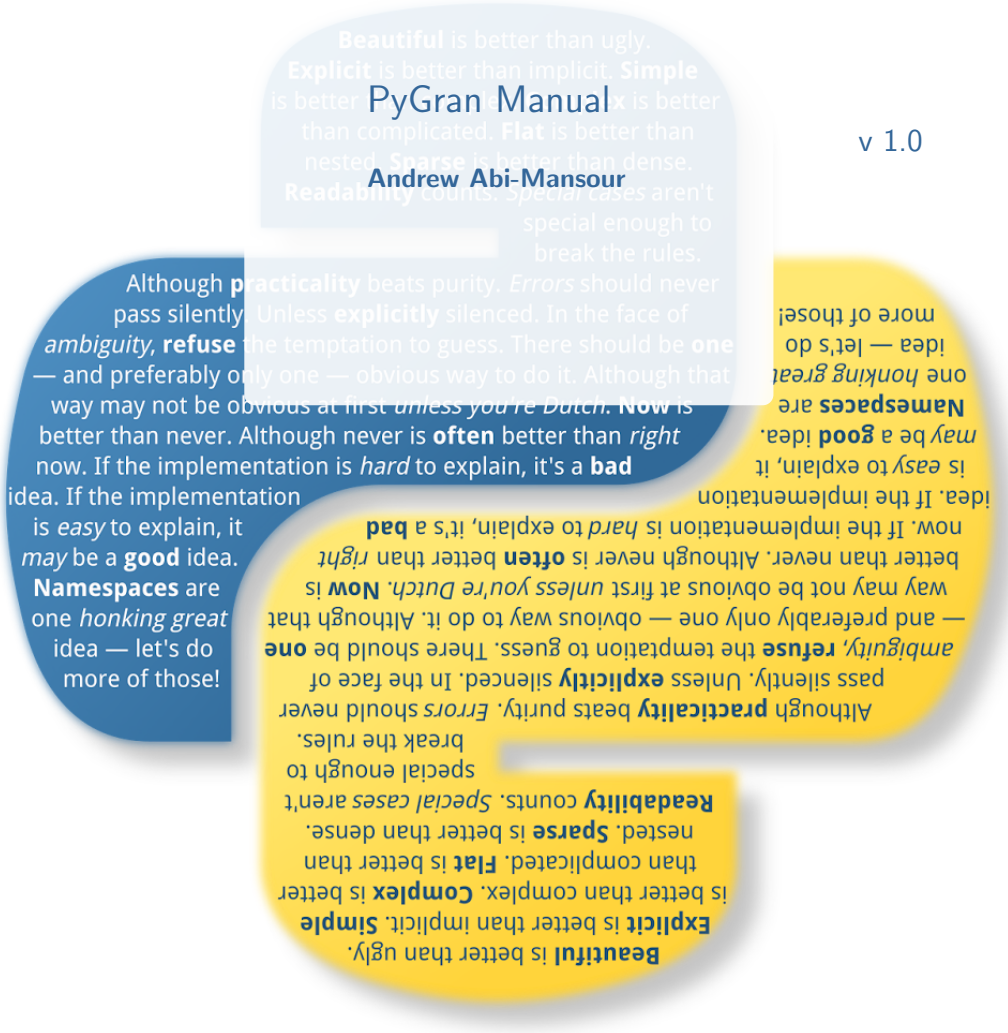
PyGran Manual

v 1.0

Andrew Abi-Mansour



python™

Contents

I Preliminary **4**

1 Introduction **5**

2 Prerequisites **7**

2.1 OS support 7

2.2 Dependencies 7

2.2.1 Core packages 7

2.2.2 Optional packages 8

2.3 Installation 9

2.3.1 Experimental version 9

2.3.2 Stable version 10

2.3.3 Configuration with *LIGGGHTS* 10

II Simulation **11**

3 Numerical Analysis **12**

3.1 Contact mechanical models 13

3.2 Examples 13

3.2.1 Hertz-Mindlin vs Spring-Dashpot 13

3.2.2 Coefficient of restitution 15

3.2.3 Cohesive particles 17

4 DEM Simulation **19**

4.1 Engines in *PyGran* 19

4.1.1 Fundamentals 19

4.2 MPI-based Engine: *LIGGGHTS* 20

4.2.1 Constructors and destructors 23

4.2.2 Input/output methods 25

4.2.3 Virtual and surface walls 25

4.2.4 External body forces 25

4.2.5 Time integration 25

III Post-processing **27**

5 Overview **28**

6 Particles **31**

7 Mesh **33**

7.1 Particle-mesh analysis 33

7.2 Coupled CFD-DEM simulations 34

8 Advanced techniques: extensions and custom objects **36**

8.1 coarse-graining 36

Part I

Preliminary

Chapter 1

Introduction

PyGran is an object-oriented library written primarily in Python for DEM simulation and analysis. The main purpose of *PyGran* is to provide an easy and intuitive way for performing technical computing in DEM, enabling flexibility in how users interact with data from the onset of a simulation and until the post-processing stage. In addition to providing a brief tutorial on installing *PyGran* for Unix systems (Part I), this manual focuses on two core modules (Figure (1.1)) in *PyGran*: *Simulator* (Part II) which provides *engines* for running DEM simulations and enables analysis of contact mechanical models, and *Analyzer* (Part III) which contains methods and modules for processing DEM data. *PyGran* is released under the GNU General Public License (GPL v2.0), and its code base is available from github.

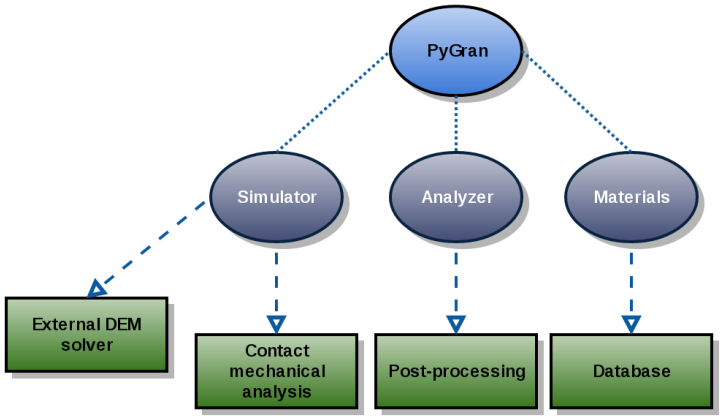


Figure 1.1: A diagram that shows the hierarchical structure of *PyGran* in terms of its core modules and submodules that can be imported from a Python input script.

Chapter 2

Prerequisites

2.1 OS support

In the current version (1.0), *PyGran* is configured to run on Unix or Unix-like operating systems. While *PyGran* can be run on a Windows platform, it has not yet been fully tested. *PyGran* supports Python 3.X and is fully backwards compatible with Python 2.2 (and later verions). Table (2.1) summarizes some of the technical details of the *PyGran* source code.

2.2 Dependencies

2.2.1 Core packages

PyGran is designed to work in conjunction with NumPy, SciPy, and other Python libraries. The following packages must be installed before *PyGran* is configured to run. For running DEM simluations with *LIGGGHTS* [1] in parallel, OpenMPI or MPICH2 must be installed on the system. Furthermore, the following four Python packages must be installed:

Nr.	Code metadata description	
C1	Current code version	v1.0
C2	Permanent link to code/repository used for this code version	https://github.com/PyGran
C3	Legal Code License	GNU General Public License v2.0
C4	Code versioning system used	Git
C5	Software code languages, tools, and services used	Python, Cython, MPI4Py
C6	Compilation requirements, operating environments	OS: Linux, Mac OS X; C compiler: gcc
C7	If available Link to developer documentation/manual	
C8	Developer email address	andrew.gaam@gmail.com

Table 2.1: Code metadata

- *Numpy* : for exposing trajectory data as *ndarray* objects and performing linear algebra floating-point operations
- *Scipy* : for efficient nearest neighbor searching routines, sorting, and non-linear solvers.
- *Cython* : for improved performance in *Analyzer.core* module
- *mpi4py* : for running DEM simulations in parallel with MPI

2.2.2 Optional packages

These packages are needed to achieve full functionality in *PyGran*.

- *PyVTK* : for reading mesh files in VTK file format
- *matplotlib* : for generating 2D plots

For reading input trajectory files in vtk/vtu format, the VTK library must be installed.

All of *PyGran*'s dependencies can be installed with pip (or pip3 for Python 3.x):

```
pip install numpy scipy mpi4py cython vtk PyVTK matplotlib --user
```

2.3 Installation

2.3.1 Experimental version

Even though *git* is not required to install or run *PyGran*, its availability makes it easier and convenient to download the latest version of the *PyGran* source code via

```
git clone https://github.com/PyGran
```

This clones the repository to a directory called 'PyGran':

```
cd PyGran
```

For updating an existing repository, *git* can be used to sync the source code with the online repository via

```
git pull origin master
```

Alternatively, one can download the source code as a tar ball (or zip file) from github.com, and then manually extract the files. *PyGran* uses Python's 'setuptools' to check for and/or download dependencies. For building *PyGran*, run from the 'PyGran' directory:

```
python setup.py build
```

For installing *PyGran*, run from the 'PyGran' directory:

```
python setup.py install
```

For a comprehensive list of options on running 'setup.py', see the doc strings in setuptools.

2.3.2 Stable version

A stable release of *PyGran* can be downloaded from github and then installed using the method described in subsection (2.3.1), or alternatively using pip:

```
pip install PyGran --user
```

Similarly one could use pip3 to install *PyGran* for Python 3.X.

2.3.3 Configuration with *LIGGGHTS*

PyGran has been successfully tested with *LIGGGHTS* [1] v3.7 and v3.8. For running DEM simulations with *LIGGGHTS*, the latter must be compiled as a shared library (shared object on Unix/Linux systems), which *PyGran* will attempt to find. Be default, *PyGran* searches for 'libliggghts.so'. The user can specify the name of the shared object and its location by writing its full path to *PyGran*/.config file. If .config file does not exist, then it must be created. For example, if a *PyGran* installation exists in /home/user/.local/lib/python3.5/site-packages/PyGran-1.0-py3.5-linux-x86_64.egg then inserting the line:

```
library=/home/user/LIGGGHTS-PUBLIC/src/lmp_mpi.so
```

to /home/user/.local/lib/python3.5/site-packages/PyGran-1.0-py3.5-linux-x86_64.egg/ instructs *PyGran* to look for the file lmp_mpi.so in the user's LIGGGHTS-PUBLIC/src directory.

Part II

Simulation

Chapter 3

Numerical Analysis

PyGran provides a convenient way for users to define materials as Python dictionaries in the *Materials* module. For instance, properties of stearic acid shown in Code (3.1) are available in the *Materials* module. This dictionary can then be used for running simulation or performing analysis.

```
stearicAcid = {
    'youngsModulus': 4.15e7,
    'poissonsRatio': 0.25,
    'coefficientFriction': 0.5,
    'coefficientRollingFriction': 0.0,
    'cohesionEnergyDensity': 0.033,
    'coefficientRestitution': 0.9,
    'coefficientRollingViscousDamping': 0.1,
    'yieldPress': 2.2e6,
    'characteristicVelocity': 0.1,
    'density': 997.164
    }
```

Listing 3.1: A Python dictionary that defines material properties of stearic acid can be conveniently used in various *PyGran* modules and routines.

The *PyGran.Simulator.models* module contains classes for 3 contact mechanical models: *SpringDashpot* [2], *HertzMindlin*, and *ThorntonNing* [3]. While

these models can be used to run a DEM simulation with *LIGGGHTS*, they also provide a way for investigating numerical aspects of contact models as shown in the next section.

3.1 Contact mechanical models

PyGran.Simulator.models.model is the basic class from which contact models are derived. This class contains methods that are overwritten by a sublcass that implements a specific contact model. The 3 contact models implemented in *PyGran* are: Spring-Dashpot [2], Hertz-Mindlin [4], and Thornton-Ning [3]. In the next section, it is demonstrated how these models can be used to perform simple numerical experiments.

3.2 Examples

3.2.1 Hertz-Mindlin vs Spring-Dashpot

When instantiating a subclass of *PyGran.Simulator.models.model*, it is important to specify the effective particle radius in the 'material' dictionary. Code 3.2 shows how *PyGran.Simulator* can be used to compute the force-displacement curves for two different visco-elastic models: spring-dashpot, and Hertz-Mindlin models for a particle of effective radius equal to 100 μm.

```
import PyGran.Simulator as Sim

# Use the following two viscoelastic models
models = [Sim.models.SpringDashpot, Sim.models.HertzMindlin]

# Define material properties
powderX = {
  'youngsModulus': 1e8,
  'poissonsRatio': 0.25,
  'coefficientRestitution': 0.9,
```

```
  'characteristicVelocity': 0.1,
  'density': 997.164,
  'radius': 1e-4
}

for model in models:

  model = model(material=powderX)
  time, soln, force = model.displacement()

  # Extract normal displacement
  deltan = soln[:,0]

  # Ignore negative (attractive) forces
  deltan = deltan[force >= 0]
  force = force[force >= 0]
```

Listing 3.2: A *PyGran* script that uses the *Simulator* module to compute the visco-elastic force between two spheres of reduced radius set to 100 μm.

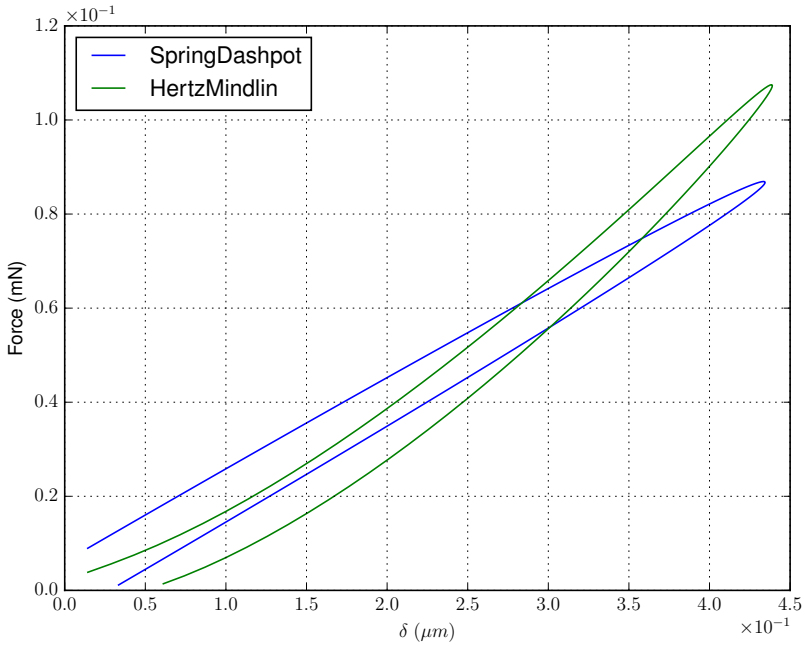


Figure 3.1: Force as a function of displacement (δ) computed for the Spring-Dashpot and Hertz-Mindlin models available in the *Simulator.models* module.

3.2.2 Coefficient of restitution

An elasto-plastic contact model suggested by Thornton and Ning [3] is available in the *PyGran.Simulator* module.

```
import PyGran.Simulator as Sim
from numpy import arange, fabs

cModel = Sim.models.ThorntonNing

# Define material properties
powderX = {
  'youngsModulus': 1e8,
  'poissonsRatio': 0.25,
  'coefficientRestitution': 0.9,
  'characteristicVelocity': 0.1,
  'density': 997.164,
```

```
  'radius': 1e-4
}

# Initialize variables
COR = []
pressure = arange(1e6, 4e6, 1e5)

for yieldPress in pressure:

  powderX['yieldPress'] = yieldPress
  model = cModel(material=powderX)

  time, disp, force = model.displacement()
  deltav = disp[:,1]

  COR.append(fabs(deltav[-1] / deltav[0]))
```

Listing 3.3: A *PyGran* script that uses the *Simulator* module to compute the elasto-plastic force between two spheres of reduced radius set to 100 μm.

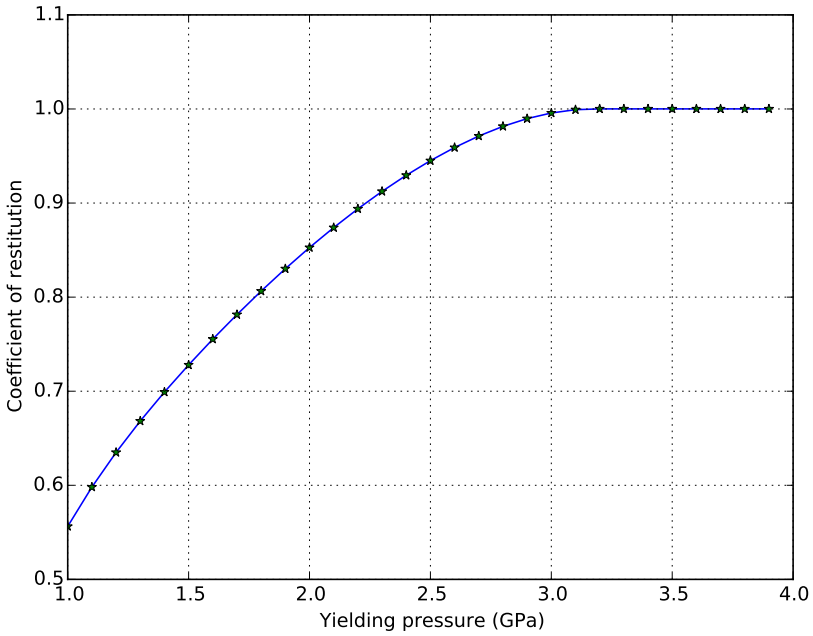


Figure 3.2: The coefficient of restitution for two spheres of reduced radius of 100 μm computed using the Thornton-Ning model implemented in *PyGran*.

3.2.3 Cohesive particles

Cohesion models are implemented in the 3 contact models discussed previously. The JKR model is available only in the Thornton-Ning model, which requires the 'cohesionEnergyDensity' keyword when supplying the 'material' dictionary to the model. Code **??** shows how the force-displacement curve is computed with the Thornton-Ning model.

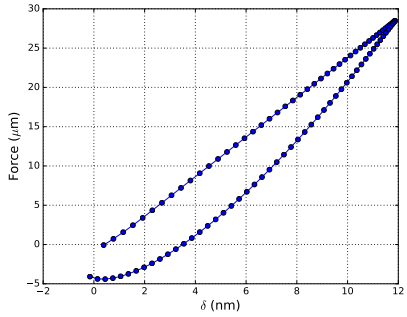
```
# Define powder properties with cohesion
powder = {'radius': 2e-5, 'yieldPress': 4e7, 'density': 1500.0,
  'youngsModulus': 6e9, 'cohesionEnergyDensity': 0.04,
  'poissonsRatio': 0.25, 'characteristicVelocity': 0.04}

# Compute the force-displacement curve
model = Sim.ThorntonNing(material=powder)
```

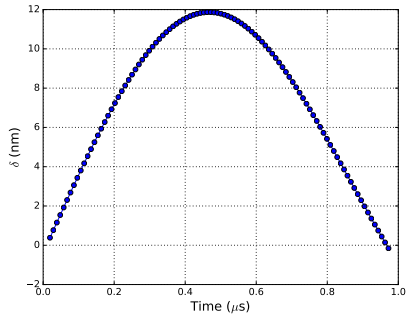
```
time, delta, force = model.displacement()

# Limit solution to the physical domain
deltan = delta[:,0]
force = force[deltan >= model.deltaf]
deltan = deltan[deltan >= model.deltaf]
time = time[deltan >= model.deltaf]
```

Listing 3.4: Cohesion can be turned on by supplying a 'cohesionEnergyDensity' value to the contact models implemented in *PyGran*.



(a) Force-displacement curve for elasto-plastic cohesive particles



(b) Displacement curve as a function of time for elasto-plastic cohesive particles

Figure 3.3: The curves are computed for spheres of reduced radius 100 μm and surface energy 0.04 J/m^2 using the Thornton-Ning model implemented in *PyGran*.

Chapter 4

DEM Simulation

4.1 Engines in *PyGran*

External N-body (DEM) solvers such as *LAMMPS* or *LIGGGHTS* can be called from *PyGran.Simulator* provided there is a supported interface that can import this solver as a separate module (in the form of a shared object). An *engine* provides an interface for *PyGran* to call specific methods in the DEM solver. While *PyGran* provides an *engine* for LIGGGHTS, it can be readily used with solvers such as Yade and ESyS-particle that have their own Python APIs.

4.1.1 Fundamentals

Any *PyGran* engine must provide a *Simulator.DEMPy* class that is instantiated by *Simulator.DEM*. The latter must be created by the user at the onset of any DEM simulation:

```
sim = Simulator.DEM(**args)
```

where *args* is a Python dictionary that contains keywords specific to the *engine* selected to setup and run the DEM simulation. The keyword 'engine' (by default

PyGran.Simulator.engines.liggghts) is used to specify which engine to use for running DEM simulation. Currently, *PyGran* supports only *LIGGGHTS* [1] as a viable engine. For that purpose, *LIGGGHTS* must be compiled as a shared library which *PyGran* will attempt to find (or alternatively the user can specify the location of the shared object in a '.config' file created in the *PyGran* directory and adding the line 'library=path/to/libliggghts.so'.

Other specific keywords for the *LIGGGHTS* engine in *PyGran* are discussed in the next section.

4.2 MPI-based Engine: *LIGGGHTS*

The Python dictionary supplied to *Simulator.DEM* can contain many keywords summarized below. The default value used (if any) for each is shown in square brackets.

- Abstract parameters:
  - model [*SpringDashpot*]: a *Simulator.models.Model* object that specifies the contact mechanical model. *Model* can be *SpringDashpot*, *HertzMindlin*, or *ThorntonNing*
  - nSim [1]: number of concurrent simulations to launch for parametrization studies
- Computational parameters:
  - units ['si']: a string specifying which unit system to use (see *LIGGGHTS* manual)
  - nns_type ['bin']: a number specifying the algorithm used to build the nearest-neighbor list (bin, nsq ,or multi)

  - nns_skin [4 * max_radius]: a number specifying the skin distance (see *LIGGGHTS* manual)

- System parameters:
  - dim [3]: an integer specifying the spatial dimensions of the simulation
  - box: a tuple if size dim * 3 specifying the box size: (x_min, x_max, y_min, y_max, ...)
  - boundary [sim_box]: a tuple of size *dim* containing strings that specify the boundary conditions (e.g. ('b_x', 'b_y', 'b_z') in 3D) where the string can be 'p' (periodic), 'f' (fixed), 's' (shrink-wrapped), or 'm' (bounded shrink wrapped). A boundary string of the form 'pf' signifies periodic conditions at the lower face and fixed conditions at the upper face. See the *LIGGGHTS* manual for additional information.
  - gravity: a tuple of length $dim + 1$ speciying the direction and magnitude of gravitional acceleration, e.g. (0, 0, -1, 9.81) specifies gravity to act in the negative z direction with magnitude 9.81.

- Component parameters: Each component in *PyGran* can be a particle or mesh type, with its own distinct material. The latter is a python dictionary that contains material parameters specific to the contact model used (see manual).
  - species: a tuple of size equal to the number of particle components, with each item being a dictionary that defines each component with the following keywords: style, material, radius, density, vol_lim, and other optional arguments that can be supplied (see the *LIGGGHTS* manual)
  - mesh: a dictionary of size equal to the number of meshes to be imported. Each item is a dictionary that defines a mesh with the key-

words: file (path to an stl file), material (see manual), mtype (mesh/-surface/stress, mesh/surface, or ...), and args (a tuple of additional arguments, see the *LIGGGHTS* manual.

- Input/output parameters:
  - output [...]: a string specifying the name of the directory to store all simulation data in. By default, it is the name of the contact model used with the day/hour/min/sec time.
  - restart [...]: a tuple of size 5 containing: (freq, dir, fname, resume, lastfile). freq: an integer specifying how often to write a restart file, dir: string specifying the name of the directory to write the restart files to, fname: string for the restart filename, resume: boolean variable for resuming simulation, lastfile: a string specifying the filename to resume the simulation from, can be 'None' for resuming the simulation from the last written restart file.
  - dump_modify [('append', 'yes')]: a tuple of arguments to pass to *LIGGGHTS* for dumping files. See *LIGGGHTS* manual.
  - traj [...]: a dictionary containing keywords for dump style. Keywords are: 'sel': species (1,2,.. 'all'), 'freq': int, 'dir': str, 'style': 'custom', 'pfile': 'traj.dump', 'mfile': 'mesh*.vtk', 'args': ('id', 'type', 'x', 'y', 'z', 'radius'), 'margs': ('id', 'stress'). *PyGran* by default writes particles as a dump file and nothing for a mesh. To change this behavior, 'pfile' can be None (no output) or a string specifying the output filename, and similarly for 'mfile' specifying the mesh output filename, which must have a vtk/stl/vtm/vtu extension. The *args* and *margs* keywords contain the attributes for particles or mesh(es) depending on the style and file extension specified. See *LIGGGHTS* manual.

4.2.1 Constructors and destructors

- insert(**args): creates particles by insertion. Returns insertion fix ID. This method has the following arguments:
  - species: either an integer (1,2,3,...) specifying the species type to insert, or the keyword 'all' to insert all defined species
  - insert ['by_pack']: insertion mechanism defined by a string that can be: 'by_pack', 'by_rate', or 'by_stream'. See *LIGGGHTS* manual.
  - value: an integer (e.g. number of particles), or a float (e.g. volume or mass fraction), depending on the mechanism of insertion
  - mech ['particles__region']: a string specifying the mechanism of insertion. If inserting by pack, this argument can be: 'particles_in_region', 'volumefraction_region', or 'mass_in_region'. If inserting by stream or rate, this argument can be: 'particlerate', 'nparticles', 'mass', or 'massrate')
  - vel_type ['constant']: a string that defines the initial velocity type: 'constant', 'uniform' (random number), or 'Gaussian'.
  - vel [(0,0,0)]: a tuple specifying the initial velocity components of all particles. Its size is: *dim* when *vel_type* is constant, *dim* * 2 when *vel_type* is 'uniform', or 'Gaussian', e.g. (v_x, v_y, v_z, dv_x, dv_y, dv_z) with the last 3 entries specifying the fluctuation amplitude in the velocity components. See *LIGGGHTS* manual.
  - insert ['by_pack']: a string specifying how particles are inserted ('by_rate', 'by_pack', or 'by_stream'). See
  - region ['sim_box']: a tuple whose 1st element is a string that indictes the region type such as 'block', 'cone', 'cylinder', etc. and the remaining elements specify the region boundaries, e.g. ('block', x_min,

x_max, y_min, y_max, z_min, z_max) defines a rectangular 3D region. See See *LIGGGHTS* manual. By default, the region of insertion is the entire simulation box.

- freq ['once']: an integer (or 'once') specifying how often to insert particles.
- all_in ['yes']: specifies whether all centers of mass of the particles inserted must be within the defined region boundaries or not. Can be either 'yes' or 'no'.
- rate: a number indicating the rate of insertion, must be supplid when *mech* is 'by_stream' or 'by_rate'.
- rate_type: a string defining the type of insertion by rate, can be either 'particlerate' or 'massrate'.
- omega [('constant',0,0,0)]: a tuple specifying the angular velocities: ('constant', omegax, omegay, omegaz).
- orientation: an optional argument, a string that defines the orientation of non-spherical particles, can be 'random' or 'template' or 'constant q1 q2 q3 q4'. See *LIGGGHTS* manual.
- set_property: an optional argument, a tuple of size 2, with the 1st item being a string that defines the variable name of a fix property/atom holding a scalar value for each particle, and the 2nd item is the new value used to initialize the property upon insertion.

- createParticles(type, style, *args): Creates particles of type 'type' (1,2, ...) using style 'box', 'region', 'single', or 'random'). args can contain the additional keywords: 'basis', 'remap', 'units', or 'all_in'. See *LIGGGHTS* manual.

- remove(id): remove a fix 'id' that is returned when fixing an insertion, an external force, a moving wall, etc.

4.2.2 Input/output methods

- dumpSetup(name='dump'): specifies which data to write as output. If supplied, 'name' is a string that specifies the dump ID which is returned by this function. By default name is 'dump'.
- extractCoords: returns all particle positions as a numpy array.

4.2.3 Virtual and surface walls

In addition to these functions, the simulation class provides the following methods:

- setupWall: creates virtual or mesh (surface) walls
- moveMesh: specifies the motion of a surface wall
- region: creates a region

4.2.4 External body forces

- add_viscous(**args): adds a force proportional to the particle velocity. args keywords: gamma (viscosity coefficient), scale: optional arg that scales the force (default 1), species: optional argument defining the species to apply the force to (1,2,...), the default is 'all'.

4.2.5 Time integration

LIGGGHTS uses the leapfrog integration scheme to solve Newton's equations in time. The function *DEM.run(nsteps, dt, itype)* is used by *PyGran.Simulator*

to create one or more integrators for all dynamical species. The three arguments for this function are:

- nsteps: number of timesteps to run (integer)
- dt: incremental timestep (float)
- itype: an optional argument that is a string or a list of strings. The string must specify the integrator type to be 'nve/sphere', 'multisphere', etc. The length of the string must be equal to all the different particle species created in the simulation, and each list item can also contains additional arguments such as: 'nve/limit relative 0.1'. See the *LIGGGHTS* manual for all types of integrators. By default, *PyGran* assigns the right integrator type based on the particle/species types defined to create the DEM simulation class.

Part III

Post-processing

Chapter 5

Overview

The *Analyzer* module enables programmers to read and analyze DEM trajectory files an in intuitive way. The most fundamental class in this module is *System*, which uses a factory class to instantiate a *SubSystem* subclass. In principle, any implementation of *SubSystem* can be instantiated with this factory. A *System* class is always instantiated by passing the filename of a specific *SubSystem*:

```
Granular = System(SubSystem='path/to/file')
```

The *System* is an iterator. Thus, it can be iterated/looped over when the supplied *SubSystem* contains a time series. For example, the statement

```
for frame in System(SubSystem='path/to/file'):
```

loops over every frame stored in the suppled file, returning the frame number at each instant.

SubSystem is an abstract class that encapsulates common attributes and methods for basic DEM objects such as *Particles* and *Mesh*, both being derived classes of *SubSystem*. While *SubSystem* is a mutable object, its properties cannot be directly modified by the user, i.e. they can modified only by the methods in *SubSystem*. The basic data structure in this object is a Python dictionary

(*SubSystem.data*) which contains references to the *SubSystem* attributes and is used to generate the dynamic interface of a *SubSystem* object. The attributes of *SubSystem* change from one frame to another for the same system, thus, *SubSystem.data* is updated every single time the *System* is evolved in time.

Instantiation and slicing

SubSystem can be instantiated using a Python dictionary that contains all the attributes (such as nodes, positions, velocities, ...) that define a *SubSystem*. These are usually read from an input trajectory file, or supplied by the user for building particle systems. *SubSystem* can be created and manipulated in ways similar to those of *Numpy* arrays; the general syntax for slicing a *SubSystem* object is

```
SliceSub = SubSystem[sel]
```

where *sel* is an integer, an *ndarray* (of type *int* or *bool*), or a Python *slice*. For example, if *sel* is i, then *SliceSub* becomes a single *SubSystem* class containing element i. Similarly, if *sel* is $i : j$, then the resultant *SliceSub* becomes a *SubSystem* class containing elements $i - j$. If *SubSystem* is a *Particles* class and *sel* is the boolean *ndarray Particles.radius* $>$ *value*, then *SliceSub* contains every particle whose radius is greater than *value*. More complex selections that involve multiple conditional statements can be created with the bitwise '|' and '&' operators. For instance, a boolean array that selects all static particles along the z direction or those in a cylindrical region of maximum height h, radius r, and center $(0, 0, 0)$ can be constructed as follows

```
sel = (Particles.vz != 0) | ((Particles.z <= h) & (Particles.x**2 +
    Particles.y**2 <= r**2))
```

A sliced class can then be instantiated as before by using the *sel* array as an argument to the *__getitem__* ([...]) operator in *Particles*.

Generators

Looping over *SubSystem* is equivalent to looping over all stored elements such that the attributes of each can be accessed but not modified, i.e.

```
for element in SubSystem:
    # Access element.property
```

Here 'element' represents the basic unit in a *SubSystem* subclass, such as a particle or a mesh triangle.

PyGran provides two *SubSystem* classes for reading particles and meshes, respectively: *System.Particles* and *System.Mesh*. These two classes are discussed in detail in the next chapters.

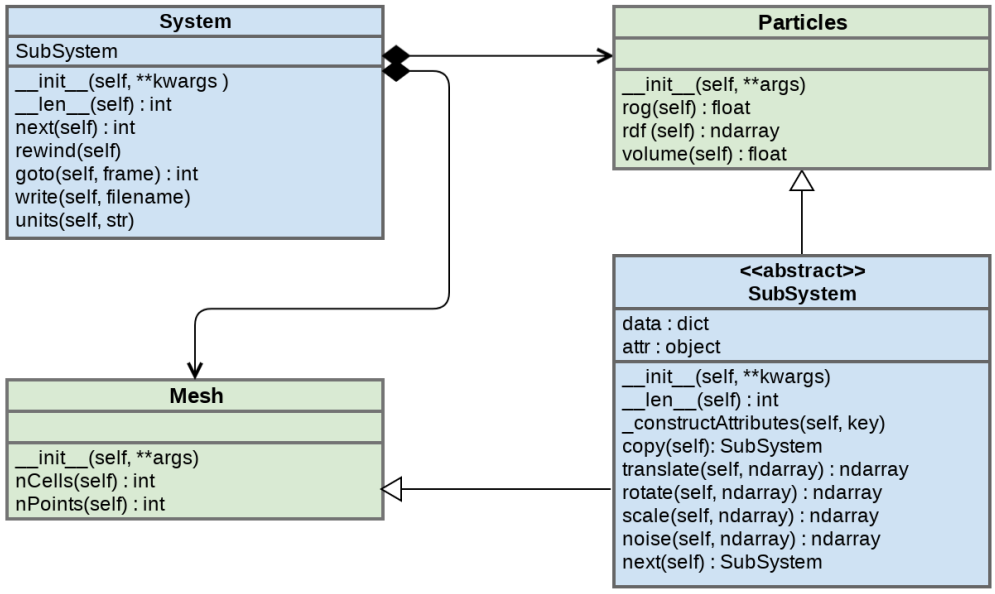


Figure 5.1: A UML diagram of the three fundamental objects and some of their methods and attributes in the *Analyzer* module. *SubSystem* contains dynamic attributes (*attr*) that are known only during runtime.

Chapter 6

Particles

The *Analyzer.Particles* class provides a way to store, manipulate, and operate on particle attributes generated by DEM simulation. This class is a subclass of *Analyzer.SubSystem* and can therefore be sliced and looped over. Furthermore, this class provides several basic routines for computing properties usually encountered in powder technology (such as mass density, radial distribution function, radius of gyration, etc.) as well as particle-based operators discussed below.

Binary operations

Extended assignments can be made to *Particles* with '+='. For example, *Particles_i* is appended to *Particles* with the following statement:

```
Particles += Particles_i
```

If *Particles_i* has fewer attributes than those in *Particles*, then this assignment is rejected. Otherwise, any additional attributes of *Particles_i* not found in *Particles* are neglected.

Two *Particles* classes can be concatenated with the '+' operator. This op-

eration can lead to reduction in the number of attributes if one of the classes being added has fewer attributes than the other(s). In this case, the resultant *Particles* will acquire concentenated attributes specified by the class with minimum number of attributes. Two *Particles* classes can also be multiplied wth '*' to yield a new object whose vector attributes are the geometric mean of the external product of the vector attributes of the two objects being multiplied. For instance, if three classes *Particles_i*, *Particles_j*, and *Particles_k* contain n_i, n_j, and n_k particles, respectively, then the following code

```
Particles = Particles_i + Particles_j * Particles_k
```

yields a new *Particles* object containing $n_i + n_j n_k$ particles and with vector attributes $[a_{i,1}, ..., a_{i,n_i}, \sqrt{a_{j,1} \times a_{k,1}}, ... \sqrt{a_{j,n_j n_k} \times a_{k,n_j n_k}}]$.

Basic methods

Some of the basic methods available to *Particles* are shown in Fig. (5.1). Furthermore, the *PyGran.Analyzer* module provides a *Neighbors* class that is instantiated with a *Particles* object to provide methods for nearest neighbor analysis. With this class, properties such as coordination numbers, overlap distances, and force chains can be readily computed (see subsection **??**).

Input/output

Any class derived from *SubSystem* must implement read/write methods. In the currecnt version, *PyGran* supports reading and writing particle trajectory files for *LIGGGHTS*. The input trajectory can be a dump or a vtk [5] file.

Chapter 7

Mesh

The *Analyzer.Mesh* class uses the VTK library to read input mesh files and expose the stored attributes (nodes, positions, stresses, etc.) to the user.

Surface walls are represented in *PyGran* by the *System.Mesh* class, a subclass of *Subsystem* (Fig. (5.1)). This class uses the VTK library [5] to read an input mesh trajectory (one or more sequence of VTK file(s)) and expose all of the stored file variables to the user. This is particularly useful for analyzing DEM simulation involving mesh-particle interaction or coupled CFD-DEM simulations as demonstrated in section 7.2. In its current version, *PyGran* supports reading only *vtk* and *vtu* input ASCII or binary files.

7.1 Particle-mesh analysis

Typically, DEM simulations consist of a set of particles interacting with 1 or more mesh(es). When coupled with *Particles*, the *Mesh* object enables a quick and easy way to analyze such simulations. An example script is shown below for a uni-axial

7.2 Coupled CFD-DEM simulations

Coupled CFD-DEM simulations are being increasingly employed in the industry to study fluidized beds [6, 7]. A sample *PyGran* script for analyzing a fluidized bed simulated with *LIGGGHTS* [1] and *OpenFOAM* [8] is shown in code 7.1. The script reads the particle (dump) trajectory file and the fluid (vtk) trajectory file to compute the pressure drop, inlet velocity, and the bed velocity along the direction of motion (z-axis).

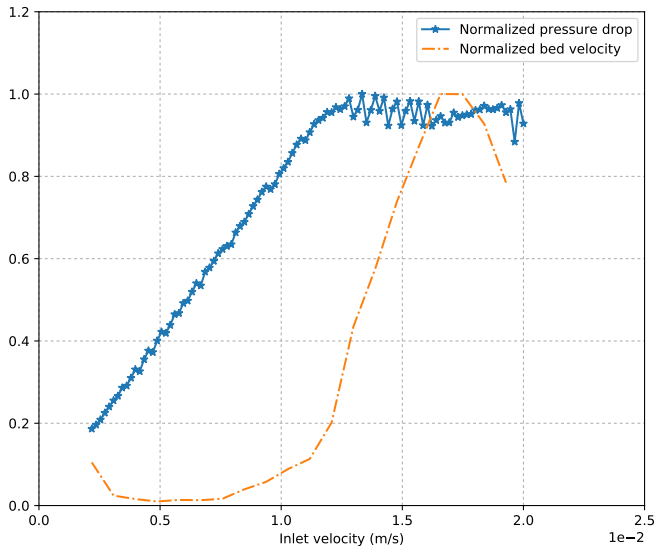


Figure 7.1: The normalized pressure drop and velocity of a fluidized bed (tutorial adopted from [9]) computed with *PyGran* as shown in code (7.1).

```
from PyGran.Analyzer import System
from scipy.linalg import norm

# Create a mesh trajectory file for the inlet & outlet files
inlet, outlet = 'CFD/inlet_*.vtk', 'CFD/outlet_*.vtk'
Traj = System(Particles='DEM/*.dump', Mesh=[inlet, outlet], vtk_type='
    poly')
```

```
# Compute mesh surface areas
iMesh, oMesh = Traj.Mesh
iArea, oArea = iMesh.CellArea.sum(), oMesh.CellArea.sum()

# Loop over inlet trajectory and compute the inlet pressure & vel
for i, timestep in enumerate(Traj):

  # Compute the weghted-average pressure inlet + outlet
  iPress[i].append((iMesh.p * iMesh.CellArea).sum() / iArea)
  oPress[i].append((oMesh.p * oMesh.CellArea).sum() / oArea)

  # Compute the weighted-average inlet velocity
  iVel[i].append(norm((iMesh.U.T * iMesh.CellArea).sum(axis=1) / iArea))

  # Compute mean particle position along the z-azis
  zMean[i].append(Traj.Particles.z.mean())
```

Listing 7.1: A Python code that shows how *PyGran* can be used to analyze coupled CFD-DEM simulations.

Chapter 8

Advanced techniques: extensions and custom objects

PyGran's extensible and object-oriented design makes it ideal for creating user-defined particles. Since *System* uses a Factory class to instantiate a *Particles* or *Mesh* object, it can in principle be used to instantiate a user-defined class. This is demonstrated in the next section for a simple coarse-grained class that demonstrates the use of the *filter* method to eliminate particles overlapping by a certain %.

8.1 coarse-graining

A simple user-defined *CoarseParticles* class can be defined as a subclass of *Particles* with two key arguments: 'scale', which controls the level of coarse-graining (or reduction) and 'percent' which is used to eliminate the resultant

coarse-grained particles overlapping by a certain percentage with respect to their radius. A script that implements this class is shown below.

```
from PyGran import Analyzer

class CoarseParticles(Analyzer.Particles):
  def __init__(self, **args):
    super(CoarseParticles, self).__init__(**args)

    if 'scale' in args and 'percent' in args:
      self.scale(args['scale'], ('radius',))
      CG = Analyzer.equilibrium.Neighbors(self).filter(percent=args['
    percent'])

      self.__init__(CoarseParticles=CG)

if __name__ == '__main__':
  Traj = Analyzer.System(CoarseParticles='traj.dump', units='micro',
    scale=3, percent=25.0)
  Traj.CoarseParticles.write('CG.dump')
```

The *CoarseParticles* object uses a recursive call to instantiate a derivative of the *Particles* class and therefore inherits all of the latter's properties and methods.

Bibliography

[1] Christoph Kloss and Christoph Goniva. Liggghts–open source discrete element simulations of granular materials based on lammps. *Supplemental Proceedings: Materials Fabrication, Properties, Characterization, and Modeling, Volume 2*, pages 781–788, 2011.

[2] Peter A Cundall and Otto DL Strack. A discrete numerical model for granular assemblies. *geotechnique*, 29(1):47–65, 1979.

[3] Colin Thornton and Zemin Ning. A theoretical model for the stick/bounce behaviour of adhesive, elastic-plastic spheres. *Powder technology*, 99(2):154–162, 1998.

[4] Nikolai V Brilliantov, Frank Spahn, Jan-Martin Hertzsch, and Thorsten Pöschel. Model for collisions in granular gases. *Physical review E*, 53(5):5382, 1996.

[5] Will J Schroeder, Bill Lorensen, and Ken Martin. *The visualization toolkit: an object-oriented approach to 3D graphics*. Kitware, 2004.

[6] Takuya Tsuji, Keizo Yabumoto, and Toshitsugu Tanaka. Spontaneous structures in three-dimensional bubbling gas-fluidized bed by parallel dem–cfd coupling simulation. *Powder Technology*, 184(2):132–140, 2008.

[7] Dalibor Jajcevic, Eva Siegmann, Charles Radeke, and Johannes G Khinast. Large-scale cfd–dem simulations of fluidized granular systems. *Chemical Engineering Science*, 98:298–310, 2013.

[8] Hrvoje Jasak, Aleksandar Jemcov, Zeljko Tukovic, et al. Openfoam: A c++ library for complex physics simulations. In *International workshop on coupled methods in numerical dynamics*, volume 1000, pages 1–20. IUC Dubrovnik, Croatia, 2007.

[9] Christoph Goniva, Christoph Kloss, Alice Hager, and Stefan Pirker. An open source cfd-dem perspective. In *Proceedings of OpenFOAM Workshop, Göteborg*, pages 22–24, 2010.