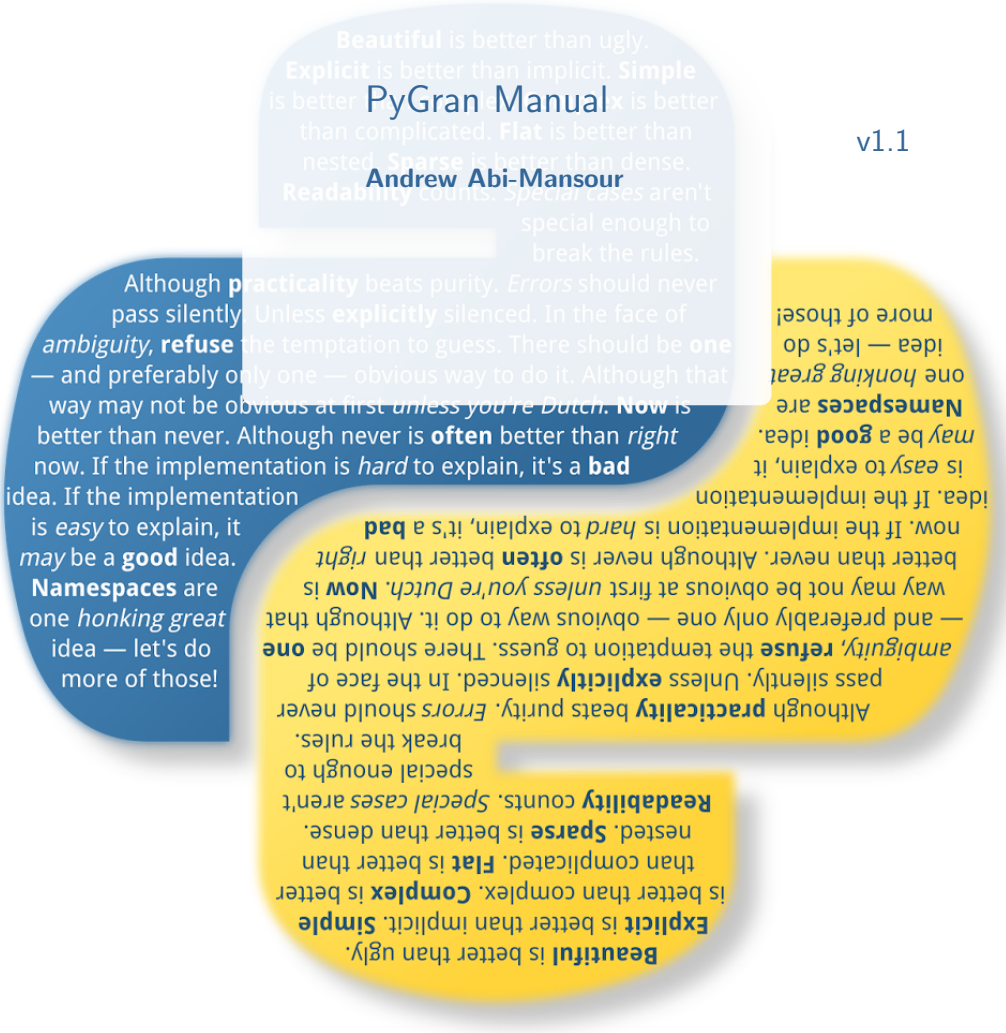
PyGran Manual

v1.1

Andrew Abi-Mansour



python™

Contents

Part I

Preliminary

Chapter 1

Introduction

1.1 What is PyGran?

PyGran is an object-oriented library written primarily in Python for Discrete Element Method (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 2 core modules (Figure (1.2)) in *PyGran*: *simulation* (Part II) which provides *engines* for running DEM simulations and enables analysis of contact mechanical models, and *analysis* (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.

Nr.	Code metadata description	
C1	Current code version	v1.1
C2	Permanent link to code/repository used for this code version	Github
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 2/3, cython, Open-MPI
C6	Compilation requirements, operating environments & dependencies	GCC, Linux, MPI, VTK, NumPy, SciPy
C7	Link to manual	Manual-v1.1
C8	E-mail	support@pygran.com

Table 1.1: Code metadata for PyGran v1.1

1.2 Brief introduction to DEM

The Discrete Element Method (DEM) is the application of Newtonian mechanics to a set of interacting particles that are usually assumed to be spherical in shape. For a spherical particle i that undergoes translation and rotation and of moment of inertia I_i and volume V_i, its dynamical equations are:

$$\rho V_i a_i = \sum_j F_{ij} + F_{b_i}, \tag{1.1}$$

$$I_i \alpha_i = \sum_j T_{ij}. \tag{1.2}$$

The true mass density of the particles is ρ, F_ij is the surface contact force between particle i and its neighbors (summed over j), F_{b_i} is a body force acting on particle i (such as gravity, i.e. $F_{b_i} = \rho V_i g$), and T_{ij} is the torque between particle i and its neighbors that is limited by the static friction coefficient μ.

DEM simulation involves the numerical solution of Eqs. (1.1-1.2) by decomposing the space the particles occupy into a grid of connected nearest-neighbor lists that enable fast and efficient computation of inter-particle surface forces. For every discrete timestep, all forces acting on each particle are computed,

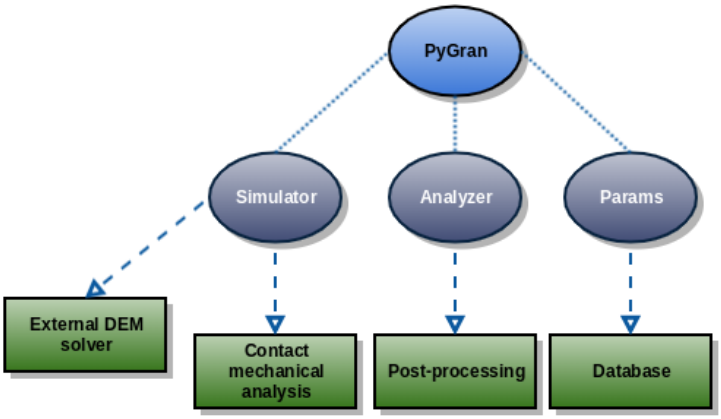


Figure 1.1: A diagram that shows the hierarchical structure of *PyGran* in terms of its core modules and submodules that can be imported from a Python input script.

and the particle positions are updated based on the discretized form of Eqs. (1.1-1.2).

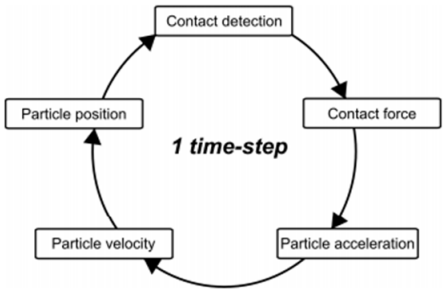


Figure 1.2: A flowchart that shows the different stages involved in completing a single DEM timestep.

The form of the surface contact forces depends on the type of the bodies interacting with each other. Visco-elastic particles for instance can be modeled as two connected spring-dashpots. More sophisticated models assume particles behave as elasto-plastic spheres that undergo plastic deformation when the pressure exceeds a yield stress characteristic of the material the particles are composed of. Irrespective of the method energy dissipation is modeled, DEM

usually assumes the two spherical particles in contact experience an elastic repulsive force that follows Hertz' law. In addition to dissipation, the particles can experience an attractive cohesive force depending on their size. The contact models implemented in *PyGran* are discussed in chapter 3.

Chapter 2

Prerequisites

2.1 OS support

In the current version (v1.1), *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 (1.1) 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:

- *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.

2.2.2 Optional packages

The packages are needed to achieve full optimal functionality in *PyGran* are:

- *Cython* : for improved performance in *analysis.core* module
- *mpi4py* : for running DEM simulations in parallel with MPI
- *vtk* : for reading input files in VTK file format
- *matplotlib* : for generating 2D plots

If *PyGran* is installed through PyPi, then all of the core dependencies will be installed as well. However, for optimal performance, it is recommended that cython is installed on the system via:

```
pip install cython–user
```

How to setup and install *PyGran* is explained in the next section.

2.3 Quick installation with PyPi

The easiest way to install the latest stable version of *PyGran* is with PyPi:

```
pip install PyGran –user
```

Similarly one could use pip3 to install *PyGran* for Python 3.X. For optimal and full performance, gcc and OpenMPI/MPICH2 must be installed (on Linux):

```
pip install cython numpy matplotlib PyGran –user
```

2.4 Installation from source code

2.4.1 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 source code via

```
git clone git@github.com:Andrew-AbiMansour/PyGran.git
```

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 develop
```

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.

2.4.2 Stable version

A stable release of *PyGran* can be downloaded from github and then installed using the method described in subsection (2.3.1), except the branch to pull the source code from is 'master':

```
git pull origin master
```

2.4.3 Configuration with *LIGGGHTS*

PyGran has been successfully tested with *LIGGGHTS-PUBLIC* [1] versions 3.4, 3.7, and 3.8. For running DEM simulation with *LIGGGHTS*, the latter must be compiled as a shared library (shared object on Unix/Linux systems), which *PyGran* will attempt to find. By default, *PyGran* searches for any available installation of 'libliggghts.so' and write its path name to the 1st line in *PyGran*/.config. *PyGran* will also attempt to find the *LIGGGHTS* version and write it to the 2nd line in the .config file. Alternatively, in case multiple versions are installed on the system, users can specify the name of the shared object and its location by writing its full path and version to the .config file. For example, if a *PyGran* installation is in /home/user/.local/lib/python3.5/site-packages/PyGran-1.0-py3.5.egg then inserting the line:

```
library=/home/user/LIGGGHTS-PUBLIC/src/lmp_mpi.so
version=3.8.0
```

to /home/user/.local/lib/python3.5/site-packages/PyGran-1.0-py3.5.egg/ instructs *PyGran* to look for the file lmp_mpi.so in the user's LIGGGHTS-PUBLIC/src directory. The 2nd line in the .config file indicates which version of *LIGGGHTS* the user intends to use. *PyGran* attempts to find this version by searching for the 'version_liggghts.txt' file included in the *LIGGGHTS-PUBLIC* source code. If this text file is not found, the user can create one anywhere on the system and insert '3.8.0' to the 1st line in that file.

Part II

Simulation

Chapter 3

Numerical Analysis

PyGran provides a convenient way for users to define materials as Python dictionaries in the *params* module. For instance, properties of stearic acid (in S.I. units) shown in Code (3.1) are available in the *params* 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.simulation.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 particle-wall collisions as shown in the next section.

3.1 Contact mechanical models

PyGran.simulation.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.simulation.models.model*, it is important to specify the particle radius in the 'material' dictionary. Code 3.2 shows how *PyGran.simulation* 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.simulation as sim
import matplotlib.pylab as plt

# 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]

  # Plot force-displacement curves
  plt.plot(deltan, force)
```

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

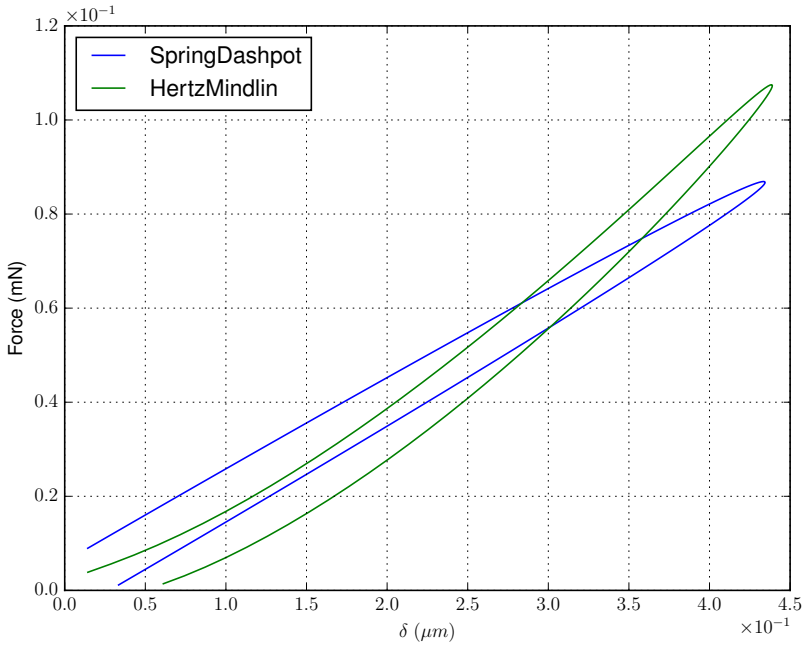


Figure 3.1: Force as a function of normal displacement computed for the Spring-Dashpot and Hertz-Mindlin models available in the *simulation.models* module.

3.2.2 Coefficient of restitution

An elasto-plastic contact model suggested by Thornton and Ning [3] is available in the *PyGran.simulation* 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 *simulation* module to compute the elasto-plastic force between two spheres of effective radius set to 100 μm.

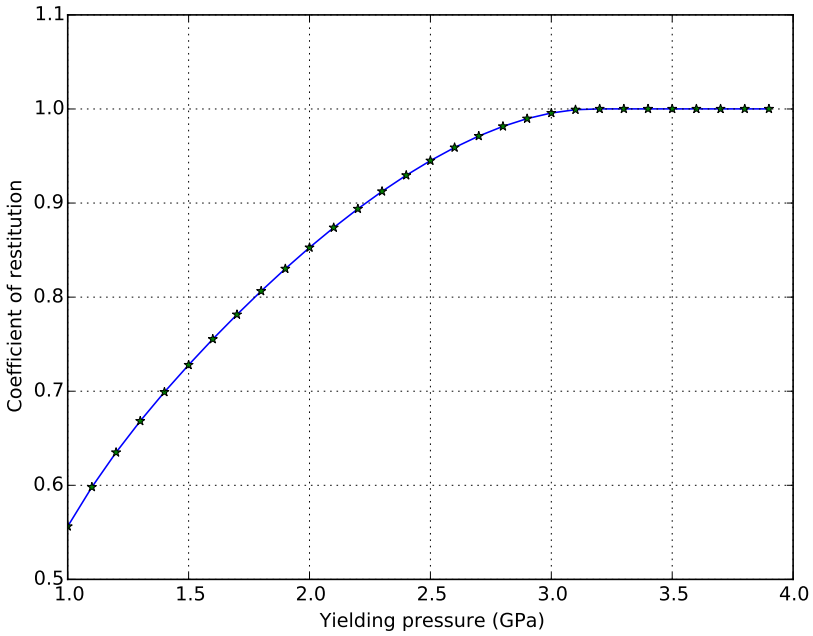


Figure 3.2: The coefficient of restitution for two spheres of reduced radius of 100 μm computed using the Thornton-Ning model implemented in *PyGran*.

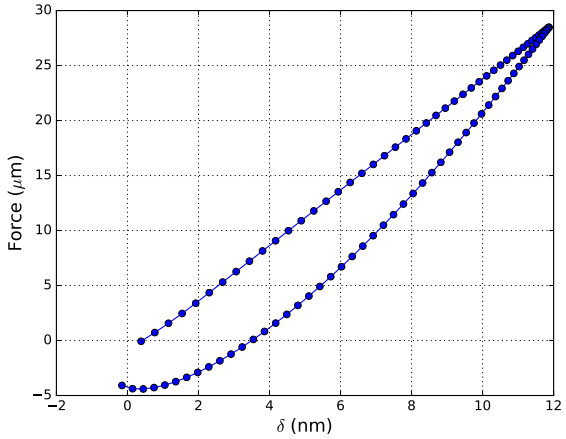
3.2.3 Cohesive particles

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' (in J/m^2) keyword when supplying the 'material' dictionary to the model. Code 3.4 shows how the force-displacement curve is computed with the Thornton-Ning model for a cohesive wall-particle collision.

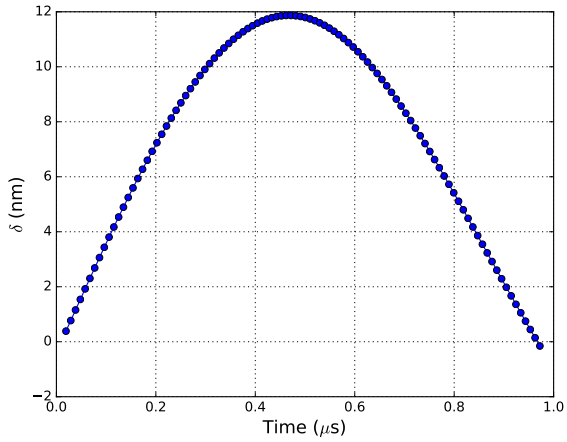
```
# Define powder properties with cohesion
powder = {'radius': 2e-5, 'yieldPress': 4e7, 'density': 1500.0,
  'youngsModulus': 6e9, 'cohesionEnergyDensity': 0.04,
  'poissonsRatio': 0.25, 'characteristicVelocity': 0.04}

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

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

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 a sphere 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.simulation* provided there is a supported Python interface that can import this solver as a separate module (in the form of a shared object on Unix or dynamic link library on Windows). 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 for post-processing with solvers such as Yade and ESyS-particle that have their own Python APIs.

4.1.1 Fundamentals

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

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

where *args* is a Python dictionary that contains keywords specific to the *en-*

gine selected to setup and run the DEM simulation. The keyword 'engine' (by default *PyGran.simulation.engines.liggghts*) in *args* is used to specify which engine to use for running DEM simulation. Currently, *PyGran* supports only *LIGGGHTS* [1] as an N-body solver. For that purpose, *LIGGGHTS* must be installed as a shared library (see subsection 2.3.3). The *__init__* constructor in *simulation.DEM* creates and/or changes directory to the user-specified output directory and initiates logging for *PyGran* (pygran.log) and *LIGGGHTS* (liggghts.log).

Other keywords for the *LIGGGHTS* engine in *simulation.DEM* are discussed in the next section.

4.2 MPI-based Engine: *LIGGGHTS*

The Python dictionary supplied to *simulation.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 *simulation.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 properties. 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, density, radius, and other optional arguments that can be supplied (see the *LIGGGHTS* manual). The particle shape ('sphere' or 'multisphere') is specified with

'style'. The 'material' keyword is a python dictionary (see). The mass density is specified with the 'density' keyword (float). The particle size distribution is by default constant. In earlier versions of *LIGGGHTS* (¡ v3.4), polydispersity could be modeled with Gaussian distribution. However, in version 3.6 and above, *LIGGGHTS* supports only monodispersed distributions. Therefore, 'radius' could be either a float number (which implies a constant particle size) or a tuple such as ('random number', mu, sigma) or ('constant', number).

- 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 keywords: 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 Particle creation

The *simulation.DEM* class provides 2 methods for creating particles summarized below.

- 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 specified by an integer (type = 1,2, ...). The style is a string that specifies the mode of insertion: 'box', 'region', 'single', or 'random'). args can contain the additional keywords: 'basis', 'remap', 'units', or 'all_in'. See *LIGGGHTS* manual for further information.

4.2.2 Input/output methods

The simulation class provides methods for extracting information from the DEM solver (*LIGGGHTS*) and writing output data to a file.

- writeSetup(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

For setting up virtual walls and specifying mesh wall movement, the following 2 functions are available:

- setupWall(wtype, species=None, plane=None, peq=None): creates virtual or mesh (surface) walls. 'wtype' is a string that can be either 'prim-

itive' (i.e. virtual) or 'mesh' surface wall. For the former, 'species' is an integer that specifies which component the wall material is made of, and 'plane' is a string ('planex', 'planey', or 'planez') that determines the axis perpendicular to the wall plane, and 'peq' is a float that sets the plane equation.

- moveMesh(name, *args): specifies the motion of a surface wall

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 velocity verlet integration scheme to solve Newton's equations in time. The function *DEM.run(nsteps, dt)* is used by *PyGran.simulation* to create one or more integrators for all dynamical species. The 2 arguments for this function are:

- nsteps: number of timesteps to run (integer)
- dt: incremental timestep (float)

4.2.6 General attributes

The following attributes are accessible in *simulation.DEM*:

- pfile [None]: string specifying the file name of the particle trajectory
- mfile [None]: a string or a listof strings specifying the file name(s) of the mesh(es) trajectory

- nSim [1]: number of concurrent simulations to run

4.2.7 Destructors

The *simulation.DEM* class provides the following 2 methods for destroying objects:

- remove(id): remove a fix 'id' that is returned when fixing an insertion, an external force, a moving wall, etc. This function can also be used to destroy surface mesh walls.
- close(): frees allocated memory and changes directory back to the current working directory. This is automatically called when the *DEM* class is instantiated with the *with* statement.

Part III

Post-processing

Chapter 5

Overview

The *analysis* 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.

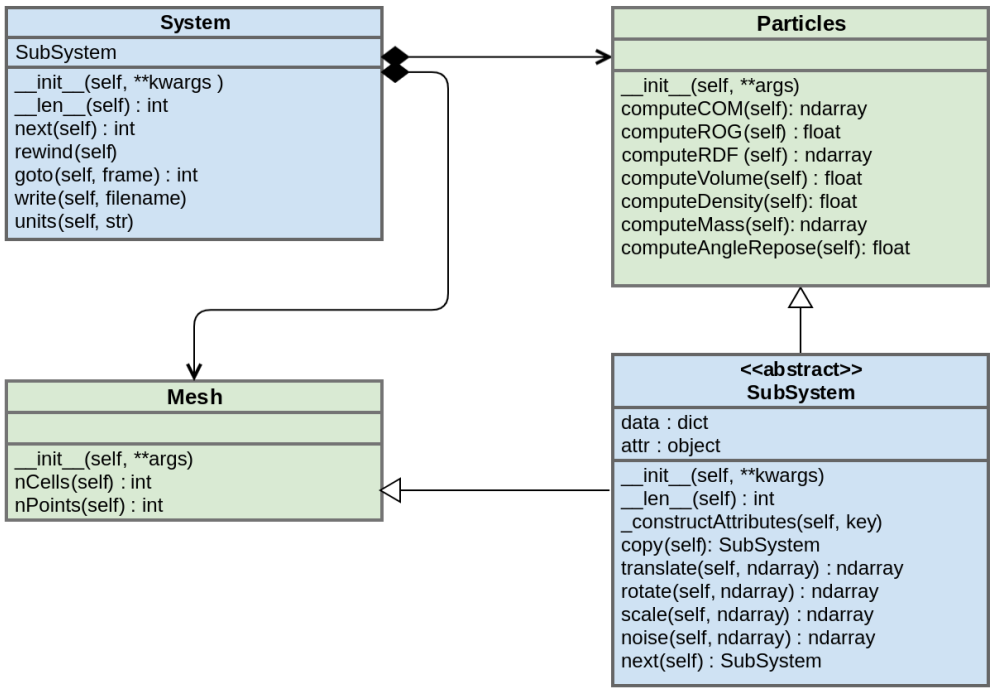


Figure 5.1: A UML diagram of the three fundamental objects and some of their methods and attributes in the *analysis* module. *SubSystem* contains dynamic attributes (*attr*) that are known only during runtime.

Chapter 6

Systems & SubSystems

6.1 System constructor

The *analysis.System* class is the most fundamental class in *PyGran*. It uses a factory to create objects derived from *SubSystem* that describe the state of a granular system (Fig. (5.1). These subclasses can be instantiated from an input *data* dictionary or copied from another instance of *SubSystem*. *System* creates an instance (or a list of instances) of *SubSystem* from input filename strings (or list of strings) that are passed to *System.__init__* by a factory object.

System contains all the objects, methods, and properties that describe the state of a DEM system. *System* also handles I/O operations and ensures proper frame to frame propagation when reading input trajectory files. The frame is controlled only by emphSystem when the latter is looped over through methods defined in a *SubSystem* sublass (read/write functions). Since DEM simulations consist of a set of particles in contact with surface triangulations (representing walls), *System* creates subclasses of *SubSystem* such as *Particles* and *Mesh* (Fig. (5.1)) based on input trajectory files. The 4 different unit systems supported

by this class are summarized in Table (**??**).

6.2 SubSystem constructor

This 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 the *SubSystem* attributes and is used to instantiate a *SubSystem* object, i.e.

```
NewSS = SubSystem(**input_data)
```

Alternatively, *SubSystem* objects can be used to create new *SubSystem* objects (i.e. copy constructor):

```
CopySS = SubSystem(SubSystem=OriginalSS)
```

6.3 Particles

The *analysis.Particles* class provides a way to store, manipulate, and operate on particle attributes generated by DEM simulation. This class is a subclass of *analysis.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 operation 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.analysis* 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.

Custom SubSystems

User-defined subclasses of *SubSystem* can be easily created by using Python's inheritence feature. The keyword '__module__' must be passed to the subclass constructor in order to make sure *PyGran* imports the module containing the subclass.

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 code below for a simple coarse-grained class that demonstrates the use of the *filter* method to eliminate particles overlapping by a certain %.

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 analysis
import os

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

    if 'scale' in args and 'percent' in args:
      self.scale(args['scale'], ('radius',))
```

```
        CG = analysis.equilibrium.Neighbors(self).filter(percent=args['
    percent'])

        self.__init__(CoarseParticles=CG)

if __name__ == '__main__':
  Traj = analysis.System(CoarseParticles='traj.dump', scale=3, percent
    =10.0, module='coarseGrained')
  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.

6.4 Mesh

The *analysis.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 **??**. In its current version, *PyGran* supports reading only *vtk* and *vtu* input ASCII or binary files.

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.