

# A Quick Guide to *SudoDEM*: A Discrete Element Code for Non-spherical Particles

Shiwei Zhao<sup>1,2</sup> and Jidong Zhao<sup>1</sup>

<sup>1</sup>*The Hong Kong University of Science and Technology*

<sup>2</sup>*South China University of Technology*

Updated on June 13, 2020, version 1.2.1



# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Why <i>SudoDEM</i> . . . . .	5
1.2	Related Work . . . . .	6
1.3	Disclaimers . . . . .	6
<b>2</b>	<b>Requirements</b>	<b>7</b>
2.1	Basic Linux Commands . . . . .	7
2.2	Python 2.7 . . . . .	8
2.3	Package Setup . . . . .	9
2.3.1	Binary Installation . . . . .	9
2.3.2	Non-root Compilation . . . . .	10
<b>3</b>	<b>Examples</b>	<b>17</b>
3.1	Example 0: run a simulation . . . . .	17
3.2	Example 1: packing of super-ellipsoids . . . . .	20
3.3	Example 2: triaxial tests of super-ellipsoids . . . . .	28
3.4	Example 3: packing of GJKparticles . . . . .	38
3.5	Example 4: packing of poly-superellipsoids . . . . .	46
3.6	Example 5: packing of superellipses . . . . .	50
<b>4</b>	<b>Post-processing</b>	<b>55</b>
4.1	Data . . . . .	55
4.2	Scene Visualization . . . . .	56
4.2.1	SudoDEM3D . . . . .	56
4.2.2	SudoDEM2D . . . . .	60
<b>5</b>	<b>Python Class Reference</b>	<b>63</b>
5.1	SudoDEM3D . . . . .	63
5.1.1	Basic classes . . . . .	63
5.1.2	Module <code>_superquadrics_utils</code> . . . . .	64
5.1.3	Module <code>_gjkparticle_utils</code> . . . . .	66

5.1.4	Module snapshot . . . . .	68
5.2	SudoDEM2D . . . . .	69
5.2.1	Basic classes . . . . .	69
5.2.2	Module _superellipse_utils . . . . .	70
5.2.3	Module _utils . . . . .	72
5.2.4	Module utils . . . . .	73
	Acknowledgments . . . . .	74

# Chapter 1

## Introduction

### 1.1 Why *SudoDEM*

A sound name is the beginning to get things started. The word *SudoDEM* is coined as a combination of ‘sudo’ and ‘DEM’. The prefix ‘sudo’ is a program for Unix-like computer operating systems that allows users to run programs as the superuser, i.e., ‘super user do’, which here means a powerful and flexible DEM simulator. On the other hand, the pronunciation of ‘sudo’ sounds like ‘pseudo’, implying that *SudoDEM* does and will consist of features distinguishing from the conventional DEM codes.

*SudoDEM* is specifically designed for modeling non-spherical particles using discrete element method (DEM), which inherits a basic framework of an open-source DEM code, YADE<sup>1</sup>. A profound modification was performed on YADE for higher efficiency in modeling non-spherical particles. The project is hosted on ResearchGate with a goal of developing a robust 2/3D DEM code for convex particles, e.g., super-ellipsoids, poly-superellipsoids, cylinders, cones, polyhedrons. In *SudoDEM*, some general optimization algorithms e.g., Levenberg-Marquardt and Nelder-Mead simplex are adopted for contact detection of superquadric particles. The popular Gilbert-Johnson-Keerthi (GJK) algorithm used in computer graphic simulation is employed for convex and non-convex (under construction) polytopes.

The project of *SudoDEM* is hosted on an individual [website](#), and the synchronous update can be also found on the [Researchgate Page](#).

---

<sup>1</sup><https://yade-dev.gitlab.io/trunk/>

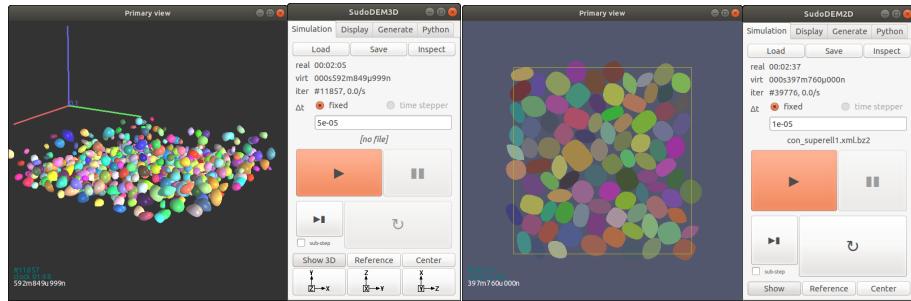


Figure 1.1: GUIs of *SudoDEM*<sup>3D</sup> (left) and *SudoDEM*<sup>2D</sup> (right).

## 1.2 Related Work

The users are appreciated to cite our work as, but not limited to, listed below.

- (1) Zhao S., Zhao J. (2019). *SudoDEM*: an open-source discrete element code for non-spherical particles, in preparation.
- (2) Zhao S., Zhao J. (2019). A poly-superellipsoid-based approach on particle morphology for DEM modeling of granular media. *International Journal for Numerical and Analytical Methods in Geomechanics*, 43(13): 2147–2169.
- (3) Zhao S., Evans T. M., Zhou X. (2018). Effects of curvature-related DEM contact model on the macro-and micro-mechanical behaviours of granular soils. *Géotechnique*, 68(12): 1085–1098.
- (4) Zhao S., Zhang N., Zhou X., Zhang L. (2017). Particle shape effects on fabric of granular random packing. *Powder technology*, 310, 175–186.

## 1.3 Disclaimers

BECAUSE THE CODE IS FREE OF CHARGE, THERE IS NO WARRANTY ‘AS IS’; NOT EVEN FOR MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

# Chapter 2

## Requirements

A Linux distribution (Ubuntu 14 to 18 suggested) with Python 2.7 and some basic python packages (e.g., numpy) are assumed to have been installed at the users' side, but no other third-party packages (e.g., boost, qt) need installing.

### 2.1 Basic Linux Commands

Only several commands the users should know are enough to set off running a program. A terminal (i.e., command window) is open by pressing a shortcut ‘CTRL+ALT+T’ or whatever other operations.

- `sudo apt-get install package1` install a *package* to the system with root permission.
- `pwd`: show the current path.
- `cd directory`: change the directory to *directory*. The *directory* can be a relative or absolute path. Two special notations ‘.’ and ‘..’ denote the current directory and the parent directory, respectively, which are useful to type a relative path. For example, ‘`cd .. /sub`’ will change the directory to the subdirectory ‘*sub*’ of the parent directory of the current path. ‘`cd /home/user`’ will change the directory to the absolute path ‘`/home/user`’.
- `prog` or `./path/prog`: run an executable named *prog*. If *prog* is exposed to the terminal, i.e., *prog* is in the searching path of the system, then just type the name *prog* like the command ‘`cd`’ at the last item; otherwise, a path should be specified to locate *prog*.

---

<sup>1</sup>or using `sudo apt install package`

- ‘CTRL+C’: terminate a program at the terminal.

```

1 $> sudo apt-get install python-numpy
2 $> pwd
3 $> cd /home/
4 $> sudodem3d

```

## 2.2 Python 2.7

Python is sensitive to indent which is used to identify a script block. ‘#’ is used to comment a line which will not be executed. Here a piece of script is shown to explain the primitives of Python:

```

1 import math # import the module math which includes some basic
    math functions
2
3 # we have numbers
4 a = 2 # integer number
5 b = 5
6 c = 2.0 # float number
7 d = a/b # the value is an integer
8 e = c/b # float number
9 print a, d, e # print the values of a, d and e to the terminal
10
11 #we have strings
12 a = 'abc' # we can assign any type of values to any type of
    variables
13 b = "this is a 'Python' script."
14 print a, b
15
16 #we have list , tuple and dict to store and manage the above basic
    primitives
17 c = [1,2,3, 'ss'] # a list
18 d = (1,2,3, 'ss') # a tuple
19 c[0] = 5 # change the first item in the list c, and the index
    starts from 0.
20 d[0] = 5 # failed! a tuple is constant, which is the
    distinguishing property from a list
21 d={1:22,2:33,4: 'ssss','w':123} #a dict: 1, 2, 4 and 'w' are keys
    of the dict, and 22, 33, 'ssss' and 123 are the corresponding
    values.
22 print d[1], d[4], d['w']
23
24 # we initialize two objects c and d for storing data at a for
    loop later
25 c = list() # an empty list

```

```

26 d = dict() # an empty dict
27 #test expression
28 if 1>2:
29     print "1 is greater than 2, really?"
30 else:
31     if 2 in [1, 2, 3]: # if 2 is in the list [1, 2, 3]
32         #ok, let's start a for loop
33         for i in range(5): # equal to for i in [0, 1, 2, 3, 4]
34             print i # you can see what's going on in the for loop
35             # we append some data to a list c and a dict d
36             c.append(math.cos(i)*10 + 2.0**5) # we append the value of
37             10*cos(i) + 2.0^5 to the list c. We use the math function cos
38             from the module math.
39             if i not in d.keys(): # if i is not a key or index of the
40                 dict d.
41                 d[i] = str(i)
42
43 #we can capsule our commands by a function
44 def GoodJob(input_number, keyword1 = 1, keyword2 = True):
45     if keyword2: # that is if keyword2 == True
46         print "the input number is", input
47     input_number += keyword1 # i.e., the value of input_number is
48     updated to input_number + keyword1
49     return input_number # we can return the value
50
51 # call the function
52 a = GoodJob(2) # a = 3, with print info 'the input number is 3'
53 b = GoodJob(3, keyword1 = 5, keyword2 = False) # b = 8, without
54     print info.

```

Ok, that is all you need to know prior to running a simulation. There are also other further techniques you might be interested in, e.g., how to open/read-/write/close a file, and how to write a Python script with the object-oriented programming, which will be roughly introduced in the examples of this tutorial.

## 2.3 Package Setup

The project of *SudoDEM* is hosted on the [website](#). Two ways to get the package installed are provided:

### 2.3.1 Binary Installation

#### Step 1: Get the package<sup>2</sup>

---

<sup>2</sup>The binary is only compiled for Ubuntu 14 to 18.

Download the compiled package from the [download page](#) and unzip it to anywhere (e.g., `/home/xxx/`). If you just get a package of the main program which does not include the third-party libraries, then you need to download the package "3rdlibs.tar.xz" for the third-party libraries and extract it into the subfolder "lib". After that, make sure you get the following structure of folders:

```
/home/xxx/SudoDEM/
  └── bin/
      └── sudodem3d
  └── lib/
      └── 3rdlibs/
          └── sudodem/
  └── share/
```

### Step 2: Set the environmental variable PATH

Append the path of the executable `sudodem3d`, i.e., '`/home/xxx/SudoDEM/bin`' to the environmental variable PATH by adding the following line to the config file '`/home/xxx/.bashrc`':

```
1 PATH=${PATH}:/home/xxx/SudoDEM/bin
```

### 2.3.2 Non-root Compilation

#### (1) Directory trees

After unzipping the source package, you have the following directory tree:

```
/home/xxx/SudoDEM
  ├── SudoDEM2D
  ├── SudoDEM3D
  ├── scripts
  ├── INSTALL
  ├── LICENSE
  └── Readme.md
```

You may add subfolders, e.g., '3rdlib', 'build2d', 'build3d', and 'sudodeminstall', inside the parent folder 'SudoDEM'. Thus, the directory tree

looks like

```
/home/xxx/SudoDEM
└── SudoDEM2D
└── SudoDEM3D
└── 3rdlib
└── build2d
└── build3d
└── sudodeminstall
└── scripts
├── INSTALL
├── LICENSE
└── Readme.md
```

The subfolder ‘3rdlib’ is for the third-party libraries. Note that the subfolder ‘3rdlib’ hosts the header files and the compiled dynamic libraries, while for launching SudoDEM you need to copy all dynamic libraries ‘\*.so.\*’ files into the install directory, e.g., ‘lib/3rdlibs/’. The compilation of the 2D and 3D versions will be done within the subfolders ‘build2d’ and ‘build3d’, respectively. The compiled outputs (binary and library) will be installed in the subfolder ‘sudodeminstall’.

## (2) Compilation of third-party libraries

```
/home/xxx/SudoDEM/3rdlib
└── boost-1_6_7
└── eigen3.3.5
└── libQGLViewer-2.6.3
└── minieigen
```

Four major libraries:

- Boost-1.67

Download the source from its official page ([https://www.boost.org/users/history/version\\_1\\_67\\_0.html](https://www.boost.org/users/history/version_1_67_0.html)) or the github repo ([https://github.com/SwaySZ/boost-1\\_6\\_7](https://github.com/SwaySZ/boost-1_6_7)).

```

1 cd boost-1\6\_7
2 ./bootstrap.sh --prefix=$PWD/../boost167 --with-
   libraries=python,thread,filesystem,iostreams,regex,
   serialization,system,date_time link=shared runtime-link=
   shared --without-icu
3 ./b2 -j3
4 ./b2 install

```

- Eigen-3.3.5: no need to compile but all head files will be included. You may download the source from the repo (<https://github.com/SwaySZ/Eigen-3.3.5>).
- MiniEigen  
Download the source from the repo (<https://github.com/SwaySZ/minieigen>). Set the directory of your compiled boost in the file ‘CMake-Lists.txt’:

```

1 set(BOOST_ROOT "/home/swayzhao/software/DEM/3rdlib/boost167
  ")

```

Then,

```

1 mkdir build
2 cd build
3 cmake ../
4 make

```

After compilation, you will get the shared library ‘minieigen.so’. Copy it to the folder ‘lib/3rdlibs/py’.

- LibQGLViewer-2.6.3  
Download the source from the repo (<https://github.com/SwaySZ/libQGLViewer-2.6.3>)

```

1 cd QGLViewer
2 qmake
3 make

```

Tools and dependencies:

- biuld-essential, cmake
- freeglut3-dev

- zlib1g-dev (boost)
- python-dev (boost)
- pyqt4-dev-tools
- qt4-default
- python-numpy python-tk
- libbz2-dev
- python-xlib python-qt4
- ipython3.0 python-matplotlib
- libxi-dev
- libglib2.0-dev
- libxmu-dev

### (3) Compilation of *SudoDEM* main programs

Here we show the example to compile *SudoDEM3D* as follows. In the folder ‘*SudoDEM3D*’, you have the directory tree:

```
/home/xxx/SudoDEM/SudoDEM3D
├── cMake
└── CMakeLists.txt

├── core
├── doc
├── gui
├── lib
├── pkg
└── py
```

Prior to compiling, you may need to edit the file ‘CMakeLists.txt’, in which you may change the installation directory, paths of library header files, etc.

(a) The installation path is set by

```
1 SET(CMAKE_INSTALL_PREFIX "${CMAKE_CURRENT_SOURCE_DIR}../
sudodeminstall/SudoDEM3D")
```

(b) The root directory of BOOST library is set by

```
1 set(BOOST_ROOT "${CMAKE_CURRENT_SOURCE_DIR}../3rdlib/boost167
")
```

(c) For the QGLViewer, the paths for both header files and library are set by the following two lines:

```
1 set(QGLVIEWER_INCLUDE_DIR "${CMAKE_CURRENT_SOURCE_DIR}../3rdlib/
libQGLViewer-2.6.3/")
2 set(QGLVIEWER_LIBRARIES "${CMAKE_CURRENT_SOURCE_DIR}../3rdlib/
libQGLViewer-2.6.3/QGLViewer/libQGLViewer.so")
```

(d) Copy your system-installed numpy and PyQt4 to the path ‘./lib/3rdlibs/py’. You may also change the search path of numpy in the file FindNumPy.cmake under the folder ‘cMake’:

```
1 set(DPDIR "${CMAKE_CURRENT_SOURCE_DIR}../dem2dinstall/SudoDEM/
lib/3rdlibs/py/")
```

Compile and install *SudoDEM3D*:

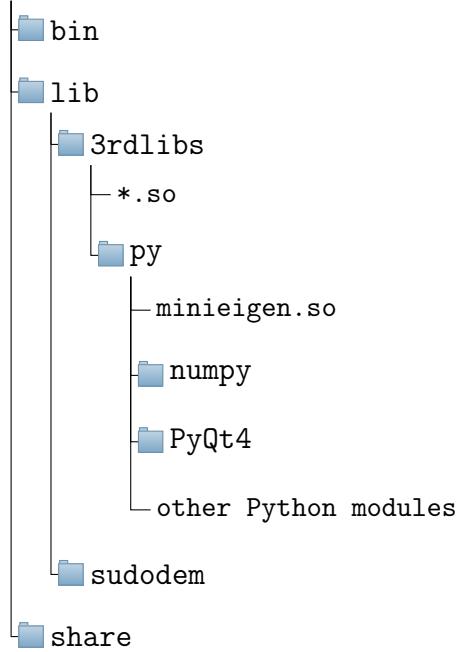
```
1 cd build3d
2 cmake ../SudoDEM3D
3 make -j3
4 make install
```

The binary ‘sudodem3d’ will be installed in the path ‘/home/xxx/SudoDEM/sudodeminstall/SudoDEM3D/bin/’. Append the path to the environmental variable PATH by adding the following line to the config file ‘/home/xxx/.bashrc’:

```
1 PATH=${PATH}:/home/xxx/SudoDEM/sudodeminstall/
SudoDEM3D/bin
```

You are expected to get the following directory tree after installation:

```
/home/xxx/SudoDEM/sudodeminstall
```



Note: the installation procedure of *SudoDEM* will overwrite the above directories except ‘3rdlibs’. You need to copy all the 3rd-party libraries compiled above (boost, libQGLViewer, minieigen) to ‘3rdlibs’. If you encounter any problems about the 3rd-libraries, you may check the directory tree for the binary package of *SudoDEM*.

**IMPORTANT:** Add rpath to all 3rd-party libraries under ‘3rdlibs’ by executing the script ‘changerpath.sh’ in the folder ‘scripts’.

---

```

1 cd 3rdlibs
2 cp /home/xxx/SudoDEM/scripts/changerpath.sh .
3 chmod +x changerpath.sh
4 ./changerpath.sh
  
```

---

Note: ‘changerpath.sh’ will add rpath ‘\$ORIGIN’ to all dynamic libraries under ‘3rdlibs’ and rpath ‘\$ORIGIN/../../’ to all dynamic libraries under ‘3rdlibs/py/PyQt4’. An rpath would make a dynamic library to search its dependencies (other dynamic libraries) from the rpath-specified path in the first place, which helps to avoid invoking other versions of dependencies in the system path.



# Chapter 3

## Examples

We prepared several examples to show a quick start of using *SudoDEM*, and the corresponding script for each example is available on the [website](#) or the Github repository (<https://github.com/SwaySZ/ExamplesSudoDEM>).

### 3.1 Example 0: run a simulation

This example will show the ingredients of a simulation by simulating a sphere free falling onto a ground. We prepare a Python script named ‘example0.py’ at our work directory (e.g., ‘/home/xxx/example’). Open a terminal, and type ‘sudodem3d example0.py’ if the work directory is at ‘/home/xxx/example/’ otherwise providing a relative or absolute path of ‘example0.py’.

Here is the script of ‘example0.py’:

```
1 # A sphere free falling on a ground under gravity
2
3 from sudodem import utils # module utils has some auxiliary
   functions
4
5 #1. we define materials for particles
6 mat = RolFrictMat(label="mat1",Kn=1e8,Ks=7e7,frictionAngle=math.
   atan(0.0),density=2650) # material 1 for particles
7 wallmat1 = RolFrictMat(label="wallmat1",Kn=1e8,Ks=0,frictionAngle
   =0.) # material 2 for walls
8
9 # we add the materials to the container for materials in the
   simulation.
10 # O can be regarded as an object for the whole simulation.
11 # We can see that materials is a property of O, so we use a dot
   operation (O.materials) to get the property (materials).
12 O.materials.append(mat) # materials is a list, i.e., a container
   for all materials involving in the simulations
```

```

13 O.materials.append(wallmat1)
14 # adding a material to the container (materials) so that we can
   easily access it by its label. See the definition of a wall
   below.
15
16 #create a spherical particle
17 sp = sphere((0,0,10.0),1.0, material = mat)
18 O.bodies.append(sp)
19 # create the ground as an infinite plane wall
20 ground = utils.wall(0,axis=2,sense=1, material = 'wallmat1') #
   utils.wall defines a wall with normal parallel to one axis of
   the global Cartesian coordinate system. The first argument (
   here = 0) specifies the location of the wall, and with the
   keyword axis (here = 2, axis has a value of 0, 1, 2 for x, y,
   z axies respectively) we know that the normal of the wall is
   along z axis with centering at z = 0 (0, specified by the
   first argument); the second keyword sense (here = 1, sense
   can be -1, 0, 1 for negative, both, and positive sides
   respectively) specifies that the positive side (i.e., the
   normal points to the positive direction of the axis) of the
   wall is activated, at which we will compute the interaction
   of a particle and this wall; the last keyword materials
   specifies a material to this wall, and we assign a string (i.
   e., the label of wallmat1) to the keyword, and directly
   assigning the object of the material (i.e., wallmat1 returned
   by RolFrictMat) is also acceptable.
21 O.bodies.append(ground) # add the ground to the body container
22
23
24 # define a function invoked periodically by a PyRunner
25 def record_data():
26     #open a file
27     fout = open('data.dat','a') #create/open a file named 'data.
28     #we want to record the position of the sphere
29     p = O.bodies[0] # the sphere has been appended into the list
   O.bodies, and we can use the corresponding index to access it
   . We have added two bodies in total to O.bodies, and the
   sphere is the first one, i.e., the index is 0.
30     # p is the object of the sphere
31     # as a body object, p has several properties, e.g., state,
   material, which can be listed by p.dict() in the commandline.
32     pos = p.state.pos # get the position of the sphere
33     print>>fout, O.iter, pos[0], pos[1], pos[2] # we print the
   iteration number and the position (x,y,z) into the file fout.
34     # then, close the file and release resource
35     fout.close()
36
37 # create a Newton engine by which the particles move under Newton

```

```

's Law
38 newton=NewtonIntegrator(damping = 0.1,gravity=(0.,0.0,-9.8),label
    ="newton")
39 # we set a local damping of 0.1 and gravitational acceleration
    -9.8 m/s2 along z axis.
40
41 # the engine container is the kernel part of a simulation
42 # During each time step, each engine will run one by one for
    completing a DEM cycle.
43 O.engines=[

44     ForceResetter(), # first, we need to reset the force container
        to store upcoming contact forces
45     # next, we conduct the broad phase of contact detection by
        comparing axis-aligned bounding boxes (AABBs) to rule out
        those pairs that are definitely not contacting.
46     InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Wall_Aabb()],

        verletDist=0.2*0.01),
47     # then, we execute the narrow phase of contact detection
48     InteractionLoop( # we will loop all potentially contacting
        pairs of particles
49         [Ig2_Sphere_Sphere_ScGeom(), Ig2_Wall_Sphere_ScGeom()], # for
            different particle shapes, we will call the corresponding
            functions to compute the contact geometry, for example,
            Ig2_Sphere_Sphere_ScGeom() is used to compute the contact
            geometry of a sphere-sphere pair.
50         [Ip2_RolFrictMat_RolFrictMat_RolFrictPhys()], # we will
            compute the physical properties of contact, e.g., contact
            stiffness and coefficient of friction, according to the
            physical properties of contacting particles.
51         [RollingResistanceLaw(use_rolling_resistance=False)] # Next
            , we compute contact forces in terms of the contact law with
            the info (contact geometric and physical properties) computed
            at last two steps.
52     ),
53     newton, # finally, we compute acceleration and velocity of
        each particle and update their positions.
54     #at each time step, we have a PyRunner function help to hack
        into a DEM cycle and do whatever you want, e.g., changing
        particles' states, and saving some data, or just stopping the
        program after a certain running time.
55     PyRunner(command='record_data()',iterPeriod=10000,label='
        record',dead = False)
56 ]
57
58 # we need to set a time step
59 O.dt = 1e-5
60
61 # clean the file data.dat and write a file head
62 fout = open('data.dat','w') # we open the file in a write mode so

```

```

    that the file will be clean
63 print>>fout, 'iter , x, y, z' # write a file head
64 fout.close() # close the file
65
66 # run the simulation
67 # you have two choices:
68 # 1. give a command here, like ,
69 # O.run()
70 # you can set how many iterations to run
71 O.run(1600000)
72 # 2. click the run button at the GUI controller

```

Run the simulation by typing the following command in a terminal:

---

```
1 sudodem3d example0.py
```

---

For multi-threads, e.g., using 2 threads <sup>1</sup>

---

```
1 sudodem3d -j2 example0.py
```

---

The recorded data 'data.dat' looks like below:

```

1 iter , x, y, z
2 10000 0.0 0.0 9.95588676912
3 20000 0.0 0.0 9.82357353912
4 30000 0.0 0.0 9.60306030912
5 40000 0.0 0.0 9.29434707912
6 50000 0.0 0.0 8.89743384912
7 60000 0.0 0.0 8.41232061912
8 70000 0.0 0.0 7.83900738912

```

We can take a quick view on the recorded data using *gnuplot*. Install *gnuplot* by typing the following command in a terminal:

---

```
1 sudo apt install gnuplot
```

---

Then, open *gnuplot* and plot the data as shown in Fig. 3.1:

---

```

1 $> gnuplot
2 gnuplot> plot 'data.dat' using ($1/10000):4; set xlabel 'iterations
[x10000]'; set ylabel 'position z [m]'
```

---

## 3.2 Example 1: packing of super-ellipsoids

The surface function of a superellipsoid in the local Cartesian coordinates can be defined as

$$\left( \left| \frac{x}{r_x} \right|^{\frac{2}{\epsilon_1}} + \left| \frac{y}{r_y} \right|^{\frac{2}{\epsilon_1}} \right)^{\frac{\epsilon_1}{\epsilon_2}} + \left| \frac{z}{r_z} \right|^{\frac{2}{\epsilon_2}} = 1 \quad (3.1)$$

---

<sup>1</sup>Note: it is not acceptable to use multi-threads for the exemplified case.

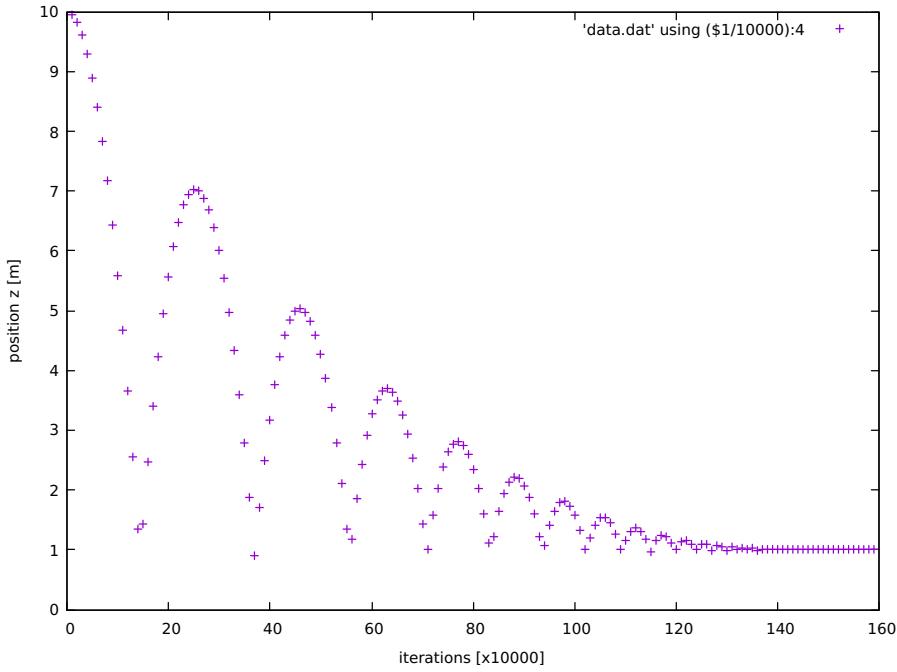


Figure 3.1: The z-position of a sphere free falling onto a ground.

where  $r_x, r_y$  and  $r_z$  are referred to as the semi-major axis lengths in the direction of x, y, and z axes, respectively; and  $\epsilon_i(i = 1, 2)$  are the shape parameters determining the sharpness of particle edges or squareness of particle surface. Varying  $\epsilon_i$  between 0 and 2 yields a wide range of convex-shaped superellipsoids. Two functions for generating a superellipsoid are highlighted below (see Sec. 5.1.2):

- `NewSuperquadrics2( $r_x, r_y, r_z, \epsilon_1, \epsilon_2$ , mat, rotate, isSphere)`
- `NewSuperquadrics_rot2( $r_x, r_y, r_z, \epsilon_1, \epsilon_2$ , mat,  $q_w, q_x, q_y, q_z$ , isSphere)`

Here we give an example of a simulation of superellipsoids free falling into a cubic box.

```

1 ######
2 # granular packing of super-ellipsoids
3 # We position particles at a lattice grid ,
4 # then particles free fall into a cubic box.
5 #####
6 # import some modules
7 from sudodem import _superquadrics_utils
8
9 from sudodem._superquadrics_utils import *

```

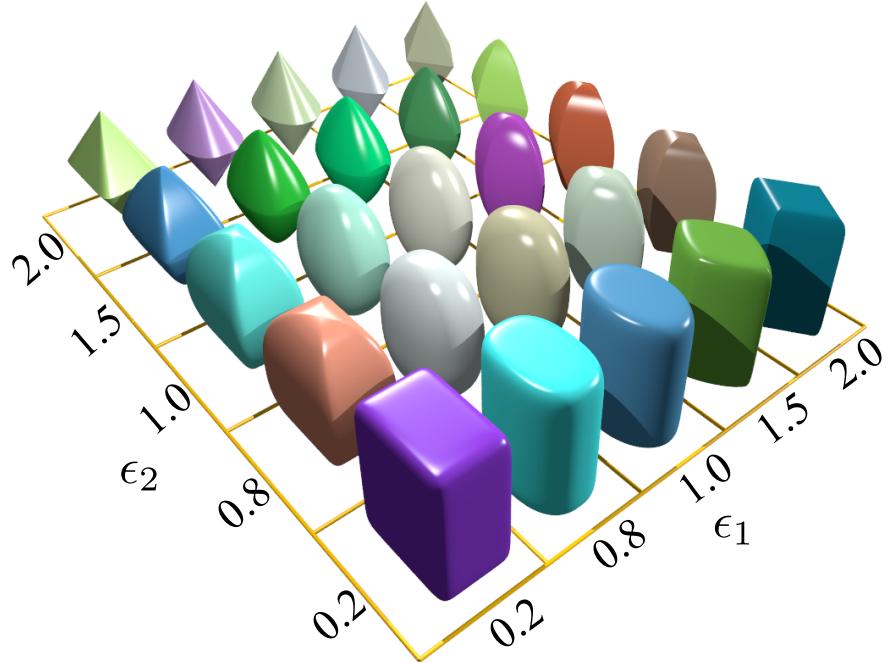


Figure 3.2: Superellipsoids.

```

10 import math
11 import random as rand
12 import numpy as np
13
14 #####define some parameters#####
15 isSphere=False
16
17 num_x = 5
18 num_y = 5
19 num_z = 20
20 R = 0.1
21
22 #####define some auxiliary functions#####
23
24 #define a lattice grid
25 #R: distance between two neighboring nodes
26 #num_x: number of nodes along x axis
27 #num_y: number of nodes along y axis
28 #num_z: number of nodes along z axis
29 #return a list of the postions of all nodes
30 def GridInitial(R,num_x=10,num_y=10,num_z=20):
31     pos = list()
32     for i in range(num_x):

```

```

33     for j in range(num_y):
34         for k in range(num_z):
35             x = i*R*2.0
36             y = j*R*2.0
37             z = k*R*2.0
38             pos.append([x,y,z])
39     return pos
40
41 #generate a sample
42 def GenSample(r, pos):
43     for p in pos:
44         epsilon1 = rand.uniform(0.7,1.6)
45         epsilon2 = rand.uniform(0.7,1.6)
46         a = r
47         b = r*rand.uniform(0.4,0.9) #aspect ratio
48         c = r*rand.uniform(0.4,0.9)
49
50         body = NewSuperquadrics2(a,b,c,epsilon1,epsilon2,p_mat,True,
51         isSphere)
52         body.state.pos=p
53         O.bodies.append(body)
54 ######setup a simulation#####
55 # material for particles
56 p_mat = SuperquadricsMat(label="mat1",Kn=1e5,Ks=7e4,frictionAngle
57 =math.atan(0.3),density=2650,betan=0,betas=0)
58 # betan and betas are coefficients of viscous damping at contact,
59 # no viscous damping with 0 by default.
60 # material for the side walls
61 wall_mat = SuperquadricsMat(label="wallmat",Kn=1e6,Ks=7e5,
62 frictionAngle=0.0,betan=0,betas=0)
63 # material for the bottom wall
64 wallmat_b = SuperquadricsMat(label="wallmat",Kn=1e6,Ks=7e5,
65 frictionAngle=math.atan(1),betan=0,betas=0)
66 # add materials to O.materials
67 O.materials.append(p_mat)
68 O.materials.append(wall_mat)
69 O.materials.append(wallmat_b)
70
71 # create the box and add it to O.bodies
72 O.bodies.append(utils.wall(-R,axis=0,sense=1,material=wall_mat
73 ))#left wall along x axis
74 O.bodies.append(utils.wall(2.0*R*num_x-R,axis=0,sense=-1,
75 material=wall_mat))#right wall along x axis
76 O.bodies.append(utils.wall(-R,axis=1,sense=1,material=wall_mat
77 ))#front wall along y axis
78 O.bodies.append(utils.wall(2.0*R*num_y-R,axis=1,sense=-1,
79 material=wall_mat))#back wall along y axis
80 O.bodies.append(utils.wall(-R,axis=2,sense=1,material=wallmat_b
81 ))#top wall along z axis

```

```

    ))#bottom wall along z axis
73
74 # create a lattice grid
75 pos = GridInitial(R,num_x=num_x,num_y=num_y,num_z=num_z) # get
    positions of all nodes
76 # create particles at each nodes
77 GenSample(R, pos)
78
79 # create engines
80 # define a Newton engine
81 newton=NewtonIntegrator(damping = 0.1, gravity=(0.,0.,-9.8),label=
    "newton",isSuperquadrics=1) # isSuperquadrics: 1 for
    superquadrics
82
83 O.engines=[

84     ForceResetter() , InsertionSortCollider ([Bo1_Superquadrics_Aabb
        () ,Bo1_Wall_Aabb()], verletDist=0.2*0.1),
85     InteractionLoop (
86         [ Ig2_Wall_Superquadrics_SuperquadricsGeom(),
87           Ig2_Superquadrics_Superquadrics_SuperquadricsGeom () ],
88         [Ip2_SuperquadricsMat_SuperquadricsMat_SuperquadricsPhys()
89          ], # collision "physics"
90         [ SuperquadricsLaw() ] # contact law
91     ),
92     newton
93 ]
O.dt=5e-5

```

With a GUI controller, we can click 'show 3D' button at the panel 'Simulation' to display the simulating scene. On the 'Display' panel, select the render 'Gl1\_Superquadrics' to configure the resolution of solid or wireframe particles, as shown in Fig. 3.3.

Figs. 3.4 - 3.6 show simulations of packing of superellipsoids at different states for the presented example.

Several properties and methods for a shape Superquadrics are listed below:

- Properties:
  - $r_x$ ,  $r_y$ ,  $r_z$ : float, semi-major axis lengths in x, y, and z axes.
  - $\epsilon_1$ ,  $\epsilon_2$ : float, shape parameters.
  - isSphere: bool, particle is spherical or not.
- Methods:
  - getVolume(): float, return particle's volume.



Figure 3.3: GUI for display configuration of superellipsoids.

- `getrxyz()`: Vector3, return particle's rxyz.
- `geteps()`: Vector2, return particle's eps1 and eps2.

Here is an example to get the volume of a particle with an id of 100:

```
1 volume = O.bodies[100].shape.getVolume()
```

A general method to list all properties of an object is as follows:

```
1 O.bodies[100].dict()
```

The output is like follows:

```
1 {'bound': <Aabb instance at 0x56072ec83da0>,
2  'chain': -1,
3  'clumpId': -1,
4  'flags': 3,
5  'groupMask': 1,
6  'id': 100,
7  'iterBorn': 0,
8  'material': <SuperquadricsMat instance at 0x56072e6ed5f0>,
```

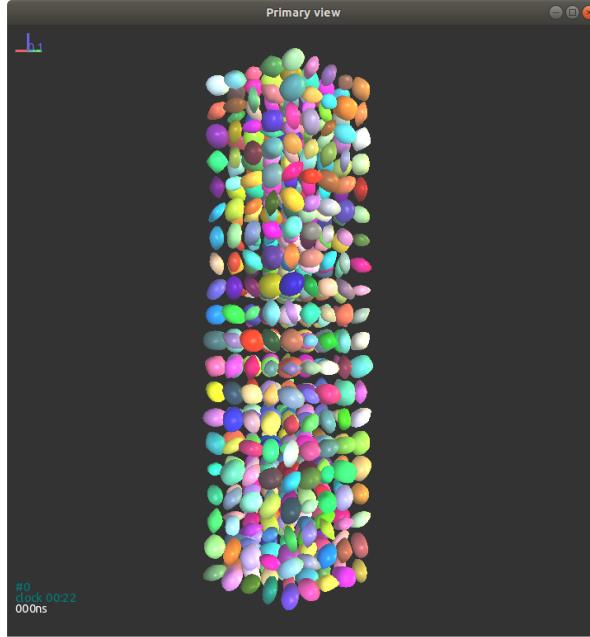


Figure 3.4: Initial packing of superellipsoids.

```

9  'shape': <Superquadrics instance at 0x56072ec83b60>,
10 'state': <State instance at 0x56072ec839e0>,
11 'timeBorn': 0.0}

```

We can see that the properties are printed in a dict form. Note that the value enclosed by pointy brackets is another object (actually, instance of an object with a specified memory address), which means that we can further access it using the ".dict()" method. Again, the properties of the object *State* can be accessed by:

```
1 O.bodies[100].state.dict()
```

The output is as follows:

```

1 {'angMom': Vector3(0,0,0),
2  'angVel': Vector3(0,0,0),
3  'blockedDOFs': 0,
4  'densityScaling': 1.0,
5  'inertia': Vector3(0.0045389863901528615,0.007287422614611933,
6                      0.0067053718092146865),
7  'isDamped': True,
8  'mass': 3.225715855242557,
9  'refOri': Quaternion((1,0,0),0),
10 'refPos': Vector3(0,0,0),

```

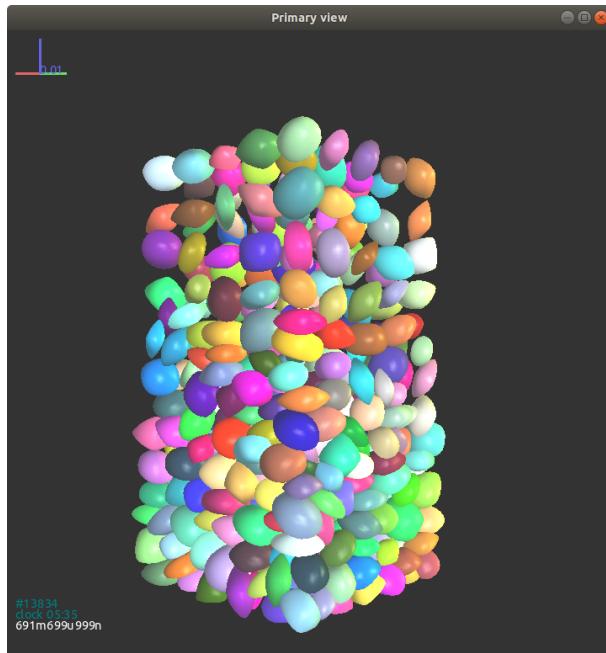


Figure 3.5: Packing of superellipsoids during falling.

```

11 'se3': (Vector3(0,0.8,3),
12 Quaternion((0.3970010520699211,0.5339678220536823,
13 -0.7465042060609055),2.1754719537556495)),
14 'vel': Vector3(0,0,0)}
```

For an object *Shape*, we list all properties which may vary for different child classes.

```
1 O.bodies[100].shape.dict()
```

The output is as follows:

```

1 {'color': Vector3(0.6407663308273844,0.07426042997942327,
2 0.05872324344642611),
3 'eps': Vector2(1.3975848801318045,1.5376309088540607),
4 'eps1': 1.3975848801318045,
5 'eps2': 1.5376309088540607,
6 'highlight': False,
7 'isSphere': False,
8 'rx': 0.1,
9 'rxyz': Vector3(0.1,0.06469580193313924,0.07572423080016706),
10 'ry': 0.06469580193313924,
11 'rz': 0.07572423080016706,
12 'wire': False}
```

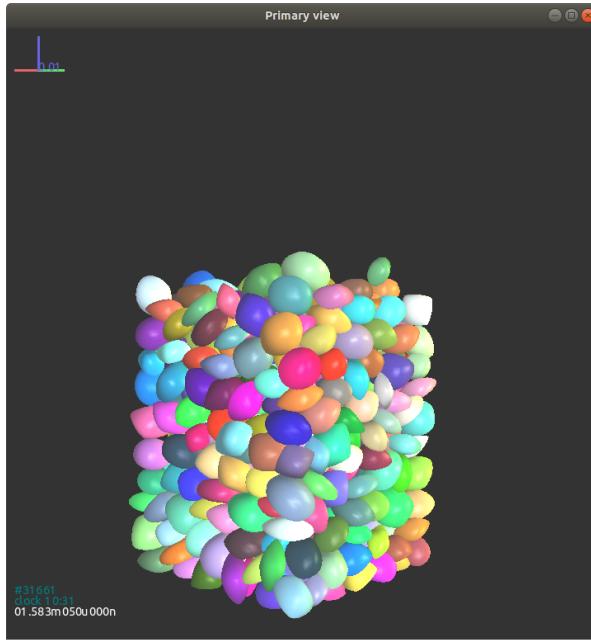


Figure 3.6: Final packing of superellipsoids.

Here we give an example to list ids and positions of all superellipsoids:

```

1 for b in O.bodies: # loop all bodies in the simulation
2     # we have to check if the present body is a superellipsoid or
3     # others, e.g., a wall
4     if isinstance(b.shape, Superquadrics): # if it is a
5         # superellipsoid, then to do
6         pid = b.id # get the id of particle b
7         pos = b.state.pos # get the position of particle b
8         print pid, pos

```

### 3.3 Example 2: triaxial tests of super-ellipsoids

This example shows a triaxial compression simulation of superellipsoids using an Engine *CompressionEngine*. The users can also customized their engines or compression procedures easily via a PyRunner Engine. Here is a basic setup of *CompressionEngine* for consolidation and compression below.

- Consolidation

```

1 triax=CompressionEngine(
2     goalx = 1.0e5, # confining stress 100 kPa
3     goaly = 1.0e5,

```

```

4     goalz = 1.0e5,
5     savedata_interval = 10000,
6     echo_interval = 2000,
7     continueFlag = False,
8     max_vel = 100.0,
9     gain_alpha = 0.5,
10    f_threshold = 0.01, #1-f/goal
11    fmin_threshold = 0.01, #the ratio of deviatoric stress
12      to mean stress
13    unbf_tol = 0.01, #unblanced force ratio
14    file='record-con'
15 )

```

- $goalx = goaly = goalz$  = confining stress
- $savedata\_interval$ : the interval of DEM iterations for saving some data (Axial Strain, Volumetric Strain, Mean Stress and DeviatorStress) into a file specified by *file*. Deprecated for consolidation.
- $echo\_interval$ : the interval of DEM iterations for outputting information (Iterations, Unbalanced force ratio, stresses in x, y, z directions) in the terminal
- $continueFlag$ : reserved keyword for continuing consolidation or compression.
- $max\_vel$ : the maximum velocity of a wall, which is useful to constrain the movement of a wall during confining.
- $gain\_alpha \alpha$ : guiding the movement of the confining walls.

$$v_w = \alpha \frac{f_w - f_g}{\Delta t \sum k_n} \quad (3.2)$$

where  $v_w$  is the velocity of a confining wall at the next time step;  $f_w$  and  $f_g$  are the present and goal confining forces, respectively;  $\Delta t$  is the time step, and  $\sum k_n$  is the summation of normal contact stiffness from all particle-wall contacts on this wall.

- $f\_threshold \alpha_f$ : a target ratio of the deviation of the present stress from the goal to the goal, i.e.,

$$\alpha_f = \frac{|f_w - f_g|}{f_g} \quad (3.3)$$

- $fmin\_threshold \beta_f$ : a target ratio of the deviatoric stress  $q$  to the mean stress  $p$ , i.e.,

$$\beta_f = \frac{q}{p}, \quad p = \frac{\sigma_{ii}}{3}, \quad q = \sqrt{\frac{3}{2}\sigma'_{ij}\sigma'_{ij}} \quad (3.4)$$

$$\sigma_{ij} = \frac{1}{V} \sum_{c \in V} f_i^c l_j^c, \quad \sigma'_{ij} = \sigma_{ij} - p \delta_{ij} \quad (3.5)$$

where  $\sigma_{ij}$  is the stress tensor with an assembly, and  $\sigma'_{ij}$  is its corresponding deviatoric part;  $V$  is the total volume of the specimen;  $f^c$  and  $l^c$  are the contact force and branch vector at contact  $c$ , respectively, and  $\delta_{ij}$  is the Kronecker delta.

- *unbf\_tol*  $\gamma_f$ : a target unbalanced force ratio defined as

$$\gamma_f = \frac{\sum_{B_i \in V} |\sum_{j \in B_i} f_j^c + f_j^b|}{2 \sum_{c \in V} |f^c|} \quad (3.6)$$

Note: the consolidation ends up with that all  $\alpha_f$ ,  $\beta_f$  and  $\gamma_f$  reach the targets.

- *file*: the file name for the recording file

- Compression

```

1 triax=CompressionEngine(
2     z_servo = False ,
3     goalx = 1.0e5 ,
4     goaly = 1.0e5 ,
5     goalz = 0.01 ,
6     ramp_interval = 10 ,
7     ramp_chunks = 2 ,
8     savedata_interval = 1000 ,
9     echo_interval = 2000 ,
10    max_vel = 10.0 ,
11    gain_alpha = 0.5 ,
12    file='sheardata' ,
13    target_strain = 0.4 ,
14    gain_alpha = 0.5
15 )

```

- *z\_servo*: a False value will trigger a strain (or displacement) control in the z direction, and in this case *goalz* needs a strain rate.
- *ramp\_interval*: iterations needed for reaching a constant strain rate. This keyword is designed to mimic a continuing loading at laboratory, but it seems not too valued and setting a small value to kindly deprecate it.

- *ramp\_chunks*: number of intervals to reach the constant strain rate combined with the keyword *ramp\_interval*. It will be deprecated as does *ramp\_interval*.
- *target\_strain*: at this level of strain the shear procedures ends. The strain is ogarithmic strain.

Note: to ensure quasi-static behavior during shear, the shear strain rate should be sufficiently small. Specifically, the inertial number  $I$  should be maintained less than  $10^{-3}$

$$I = \dot{\epsilon}_1 \frac{\hat{d}}{\sqrt{\sigma_0/\rho}} \quad (3.7)$$

where  $\dot{\epsilon}_1 = d\epsilon_1/dt$  is the major principal strain rate;  $\hat{d}$  and  $\rho$  are the average diameter and material density of particles respectively;  $\sigma_0$  is the confining stress.

As a demonstration, we first give an example of consolidation as follows:

```

1 #####
2 from sudodem import plot, _superquadrics_utils
3 from sudodem._superquadrics_utils import *
4
5 import math
6 import random as rand
7 rand.seed(11245642)
8 #####
9 epsilon = 1.4
10 isSphere = False
11 boxsize=[16.2e-3,16.2e-3,16.2e-3]
12 #####
13 #some auxiliary functions
14 #generate a sample of monodisperse superellipsoids with random
   orientations and positions
15 def gen_sample(width, depth, height, r_max = 1.0e-3):
16     b_num_tot = 5000
17     for j in range(b_num_tot):
18         rb = r_max*0.5
19         x = rand.uniform(rb, width-rb)
20         y = rand.uniform(rb, depth - rb)
21         z = rand.uniform(rb, height - rb)
22         r = r_max*0.5
23         b = NewSuperquadrics2(r*1.25,r,r,epsilon,epsilon,mat,True,
24                               isSphere)
25         b.state.pos=(x, y, z)
26         O.bodies.append(b)
27 #Materials definition

```

```

27 mat = SuperquadricsMat(label="mat1",Kn=3e4,Ks=3e4,frictionAngle=
28     math.atan(0.1),density=2650e4) #define Material with default
29     values
30 wallmat1 = SuperquadricsMat(label="wallmat1",Kn=1e6,Ks=0.,
31     frictionAngle=0.) #define Material with default values
32 wallmat2 = SuperquadricsMat(label="wallmat2",Kn=1e6,Ks=0.,
33     frictionAngle=0.) #define Material with default values
34 O.materials.append(mat)
35 O.materials.append(wallmat1)
36 O.materials.append(wallmat2)
37 #####create confining walls
38 #####create particles
39 O.bodies.append(utils.wall(0, axis=0, sense=1, material = 'wallmat1
40     '))#left x
41 O.bodies.append(utils.wall(boxsize[0], axis=0, sense=-1, material =
42     'wallmat1'))#right x
43 O.bodies.append(utils.wall(0, axis=1, sense=1, material = 'wallmat1
44     '))#front y
45 O.bodies.append(utils.wall(boxsize[1], axis=1, sense=-1, material =
46     'wallmat1'))#back y
47 O.bodies.append(utils.wall(0, axis=2, sense=1, material = 'wallmat2
48     '))#bottom z
49 O.bodies.append(utils.wall(boxsize[2], axis=2, sense=-1, material =
50     'wallmat2'))#top z
51 #####function for saving data (e.g., simulation states here)
52 def savedata():
53     O.save(str(O.time)+'.xml.bz2')
54 triax=CompressionEngine(
55     goalex = 1.0e5, # confining stress 100 kPa
56     goaly = 1.0e5,
57     goalz = 1.0e5,
58     savedata_interval = 10000,
59     echo_interval = 2000,
60     continueFlag = False,
61     max_vel = 10.0,
62     gain_alpha = 0.5,
63     f_threshold = 0.01, #1-f/goal

```

```

64     fmin_threshold = 0.01, #the ratio of deviatoric stress to mean
       stress
65     unbf_tol = 0.01, #unblanced force ratio
66     file='record-con',
67 )
68 #####
69 #####
70
71
72
73 newton=NewtonIntegrator(damping = 0.3, gravity=(0.,0.,0.),label="
    newton",isSuperquadrics=1)
74 O.engines=[
75     ForceResetter(),
76     InsertionSortCollider([Bo1_Superquadrics_Aabb(),Bo1_Wall_Aabb
        ()],verletDist=0.1e-3),
77     InteractionLoop(
78         [Ig2_Wall_Superquadrics_SuperquadricsGeom(),
79          Ig2_Superquadrics_Superquadrics_SuperquadricsGeom()],
80          [Ip2_SuperquadricsMat_SuperquadricsMat_SuperquadricsPhys()
81          ],
82          [SuperquadricsLaw()])
83     ),
84     triax,
85     newton,
86     PyRunner(command='quiet_ball()',iterPeriod=2,iterLast = 50000,
87     label='calm',dead = False),
88     PyRunner(command='savedata()',realPeriod=7200.0,label='
    savedata',dead = False)#saving data every two hours
89 ]
90
91
92
93 #delete the balls outside the box
94 def del_balls():
95     num_del = 0
96     #wall position
97     left = O.bodies[0].state.pos[0]#x
98     right = O.bodies[1].state.pos[0]
99     front = O.bodies[2].state.pos[1]#y
100    back = O.bodies[3].state.pos[1]
101    bottom = O.bodies[4].state.pos[2]#z
102    top = O.bodies[5].state.pos[2]
103    for b in O.bodies:
104        if isinstance(b.shape,Superquadrics):
105            (x,y,z) = b.state.pos
106            if x > right or x < left or y < front or
107                y > back or z < bottom or z > top:
108                O.bodies.erase(b.id)
109                num_del += 1

```

```

105     print str(num_del)+" particles are deleted!"
106
107
108
109 def quiet_ball():
110     global calm_num
111
112     newton.quiet_system_flag = True
113     if calm_num > 2000:
114         O.engines[-2].iterPeriod= 5
115         calm_num += 5
116     else:
117         calm_num += 2
118     if calm_num > 40000:
119         if calm_num < 40010:
120             print "calm procedure is over"
121             del_balls()
122         if calm_num > 50000:
123             O.engines[-2].dead = True
124             O.save("init_assembly.xml.bz2")
125             #consolidation begins
126             triax.dead=False
127             print "consolidation begins"
128
129
130 O.dt=1e-4
131
132
133 triax.dead=True
134 calm_num = 0

```

The output in the terminal is akin to the follows:

```

1 calm procedure is over
2 0 particles are deleted!
3 consolidation begins
4 Iter 50000 Ubf 0.483161, Ss_x:9.67986e-05, Ss_y:4.15364e-05,
   Ss_z:6.1184e-05, e:0.992618
5 Iter 52000 Ubf 0.461741, Ss_x:1.2412, Ss_y:1.06812, Ss_z:1.10116,
   e:0.854445
6 Iter 54000 Ubf 0.263699, Ss_x:1.75334, Ss_y:1.98567, Ss_z
   :1.62181, e:0.757469
7 Iter 56000 Ubf 0.118329, Ss_x:3.06302, Ss_y:3.54818, Ss_z
   :3.22986, e:0.679106
8 Iter 58000 Ubf 0.0110747, Ss_x:22.0177, Ss_y:21.3303, Ss_z
   :21.3485, e:0.614824
9 Iter 60000 Ubf 0.000577066, Ss_x:91.4199, Ss_y:89.6039, Ss_z
   :89.1057, e:0.58984

```

```

10 Iter 62000 Ubf 0.000128293, Ss_x:100.012, Ss_y:99.7251, Ss_z
   :99.6123, e:0.587807
11 consolidation completed!

```

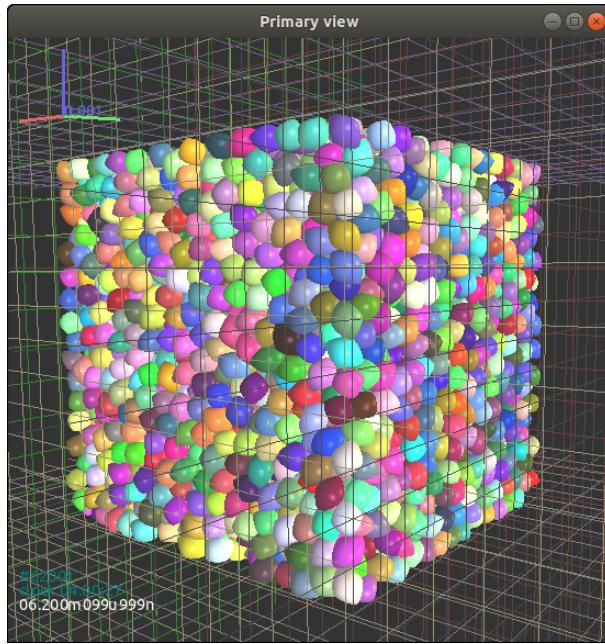


Figure 3.7: Packing of superellipsoids after consolidation.

After consolidation (see the snapshot in Fig. 3.7), we obtain a state file by

```
1 O.save("final_con.xml.bz2")
```

Then, the consolidated assembly is subjected to triaxial compression with the following scripts.

```

1 #####
2 from sudodem import plot, _superquadrics_utils
3
4 from sudodem._superquadrics_utils import *
5
6 import math
7
8 saving_num = 0
9 #####
10 def savedata():
11     global saving_num
12     saving_num += 1
13     #print "testing, no data saved"

```

```

14         O.save('shear'+str(saving_num)+'.xml.bz2')
15
16 def quiet_system():
17     for i in O.bodies:
18         i.state.vel=(0.,0.,0.)
19         i.state.angVel=(0.,0.,0.)
20 suffix=""
21 O.load("final_con"+suffix+".xml.bz2")
22 triax = O.engines[3]
23 O.dt = 5e-5
24 friction = 0.5
25 O.materials[0].frictionAngle=math.atan(friction)
26 if 1:
27     for itr in O.interactions:
28         id = min(itr.id1,itr.id2)
29         if id > 5:
30             itr.phys.tangensOfFrictionAngle = friction
31             #itr.phys.betan = 0.5
32
33 ######
34 #shear begins
35 #####
36 def shear():
37     quiet_system()
38     triax.z_servo = False
39     triax.ramp_interval = 10
40     triax.ramp_chunks = 2
41     triax.file="sheardata"+suffix
42     triax.goalz = 0.01
43     triax.goalx = 1e5
44     triax.goaly = 1e5
45     triax.savedata_interval = 2500
46     triax.echo_interval = 2000
47     triax.target_strain = 0.4
48
49 shear()
50
51 O.engines[-1].iterPeriod = 2500
52 O.engines[-1].realPeriod = 0.0

```

The compression ends up with an axial strain of 40%, and the final configuration of particles is shown in Fig. 3.8.

The macroscopic mechanical response of a granular material during shearing can be quantified by the stress tensor that is calculated from discrete measurements with the following definition:

$$\sigma_{ij} = \frac{1}{V} \sum_{c \in V} f_i^c l_j^c \quad (3.8)$$

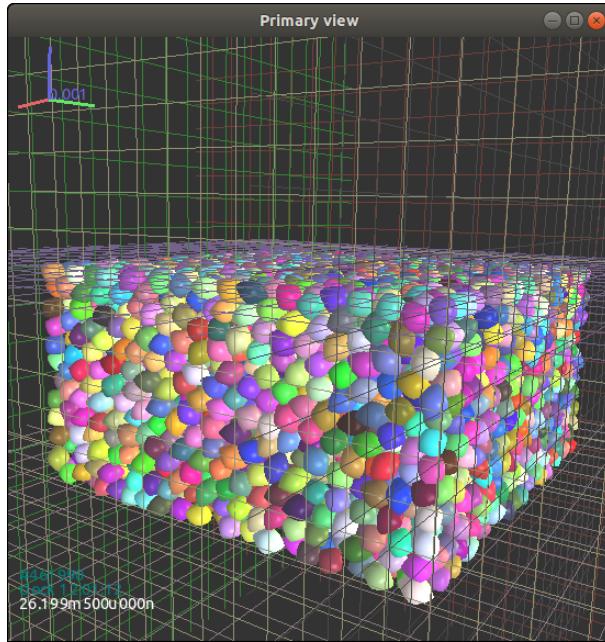


Figure 3.8: Packing of superellipsoids after shear.

where  $V$  is the total volume of the assembly,  $f^c$  is the contact force at the contact  $c$ , and  $l^c$  is the branch vector joining the the centres of the two contacting particles at contact  $c$ . The mean stress  $p$  and the deviatoric stress  $q$  are defined as:

$$p = \frac{1}{3}\sigma_{ii}, \quad q = \sqrt{\frac{3}{2}\sigma'_{ij}\sigma'_{ij}} \quad (3.9)$$

where  $\sigma'_{ij}$  is the deviatoric part of stress tensor  $\sigma_{ij}$ .

Given that the cubical specimens are confined by rigid walls, the axial strain  $\epsilon_z$  and the volumetric strain  $\epsilon_v$  can be approximately calculated from the positions of the boundary walls, i.e.,

$$\epsilon_z = \int_{H_0}^H \frac{dh}{h} = -\ln \frac{H_0}{H}, \quad \epsilon_v = \epsilon_x + \epsilon_y + \epsilon_z = \int_{V_0}^V \frac{dv}{v} = -\ln \frac{V_0}{V} \quad (3.10)$$

where  $H$  and  $V$  are the height and volume of the specimen during shearing, respectively, and  $H_0$  and  $V_0$  are their initial values before shear. Positive values of volumetric strain represent dilatancy.

The recorded data 'sheardata' looks like below:

```

1 AxialStrain VolStrain meanStress deviatorStress
2 0.000511000266 0.000490480823 106090.98 15931.56
3 0.003011001516 0.002450117535 131061.41 85718.91

```

```

4 0.005511002766 0.003867756447 148076.65 131980.95
5 0.008011004016 0.004825115794 159924.28 163706.90
6 0.010511005266 0.005409561745 168606.27 186520.14
7 0.013011006516 0.005673777079 175131.53 203243.28
8 0.015511007766 0.005650846281 180293.19 216248.17
9 0.018011009016 0.005393122254 184581.23 227010.23

```

Then we can plot the stress-strain variation and the loading path as shown in Figs. 3.9 and 3.10.

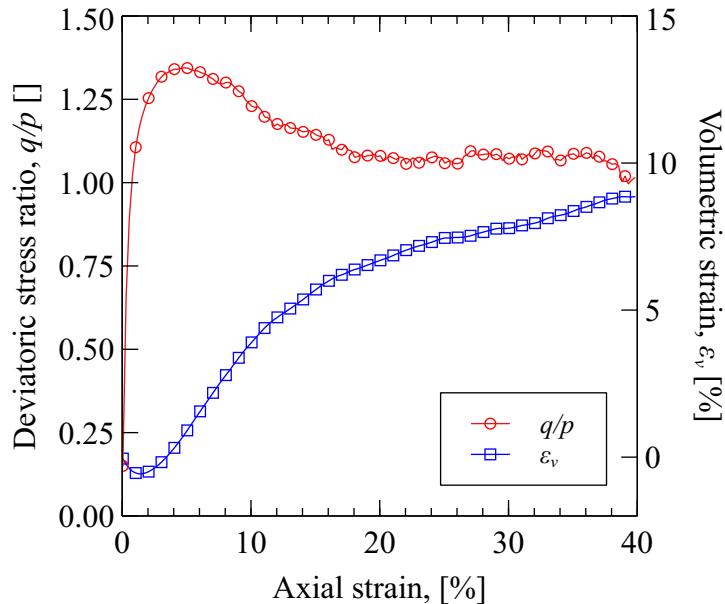


Figure 3.9: Variation of deviatoric stress ratio  $q/p$  and volumetric strain  $\epsilon_v$  with axial strain during shearing.

During shearing, a series of states (i.e., files "shearxxx.xml.bz2") has been saved sequentially so that it is readily to access other data for post-processing, e.g., variation of mean coordination number, anisotropy and so forth.

### 3.4 Example 3: packing of GJKparticles

There are five kinds of shapes, namely sphere, polyhedron, cone, cylinder and cuboid, in *SudoDEM*, for which the GJK algorithm of contact detection is performed<sup>2</sup>, so that we coined a name *GJKparticle* for grouping these particle shapes. In the Python module *\_gjkparticle\_utils* (see Sec. 5.1.3), several construction functions are provided including:

---

<sup>2</sup>GJKparticle is just experimental, and has not been rigorously tested yet.

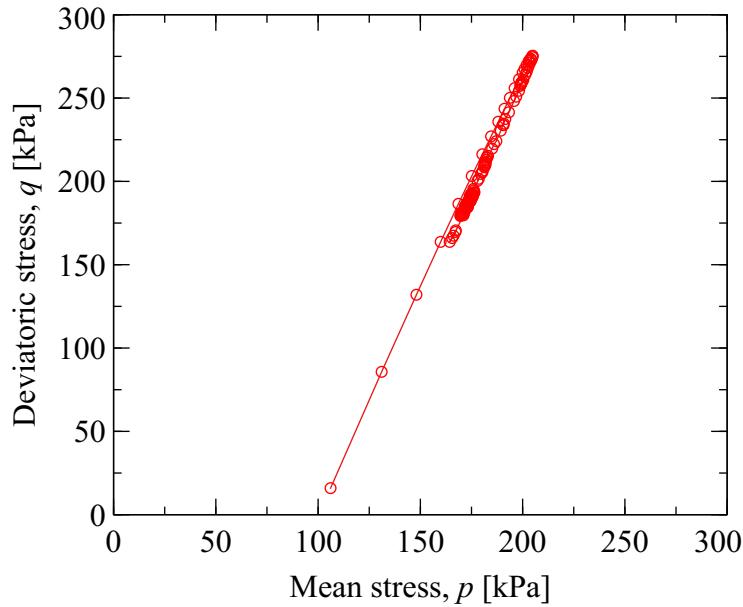


Figure 3.10: Loading path during shearing.

- GJKSphere(*radius, margin, mat*)
- GJKPolyhedron(*vertices, extent, margin, mat, rotate*)
- GJKCone(*radius, height, margin, mat, rotate*)
- GJKCylinder(*radius, height, margin, mat, rotate*)
- GJKCuboid(*extent, margin, mat, rotate*)

Note: a margin is used to expand a particle for achieving a better computational efficiency, which however should be large enough for ensuring that collision occurs in this small region but small enough for shape fidelity.

Similar to Example 1, we conduct some simulations of particles free falling into a cubic box. We first import some modules:

```

1 from sudodem import plot, _gjkparticle_utils
2 from sudodem import *
3
4 from sudodem._gjkparticle_utils import *
5 import random as rand
6 import math

```

Next, define some constants:

```

1 cube_v0 =
2   [[0 ,0 ,0],[1 ,0 ,0],[1 ,1 ,0],[0 ,1 ,0],[0 ,0 ,1],[1 ,0 ,1],[1 ,1 ,1],[0 ,1 ,1]]
3
4 isSphere=False
5
6 num_x = 5
7 num_y = 5
8 num_z = 20
9 R = 0.01

```

Define a lattice grid:

```

1 #R: distance between two neighboring nodes
2 #num_x: number of nodes along x axis
3 #num_y: number of nodes along y axis
4 #num_z: number of nodes along z axis
5 #return a list of the postions of all nodes
6 def GridInitial(R,num_x=10,num_y=10,num_z=20):
7     pos = list()
8     for i in range(num_x):
9         for j in range(num_y):
10            for k in range(num_z):
11                x = i*R*2.0
12                y = j*R*2.0
13                z = k*R*2.0
14                pos.append([x,y,z])
15

```

Set properties of materials:

```

1 p_mat = GJKParticleMat(label="mat1",Kn=1e4,Ks=7e3,frictionAngle=
2   math.atan(0.5),density=2650,betan=0,betas=0) #define Material
3   with default values
4 #mat = GJKParticleMat(label="mat1",Kn=1e6,Ks=1e5,frictionAngle
5   =0.5,density=2650) #define Material with default values
6 wall_mat = GJKParticleMat(label="wallmat",Kn=1e4,Ks=7e3,
7   frictionAngle=0.0,betan=0,betas=0) #define Material with
8   default values
9 wallmat_b = GJKParticleMat(label="wallmat",Kn=1e4,Ks=7e3,
10  frictionAngle=math.atan(1),betan=0,betas=0) #define Material
11  with default values
12
13 O.materials.append(p_mat)
14 O.materials.append(wall_mat)
15 O.materials.append(wallmat_b)

```

Create the cubic box:

```

1 # create the box and add it to O.bodies

```

```

2 O.bodies.append(utils.wall(-R, axis=0,sense=1, material = wall_mat
    ))#left wall along x axis
3 O.bodies.append(utils.wall(2.0*R*num_x-R, axis=0,sense=-1,
    material = wall_mat))#right wall along x axis
4 O.bodies.append(utils.wall(-R, axis=1,sense=1, material = wall_mat
    ))#front wall along y axis
5 O.bodies.append(utils.wall(2.0*R*num_y-R, axis=1,sense=-1,
    material = wall_mat))#back wall along y axis
6 O.bodies.append(utils.wall(-R, axis=2,sense=1, material =wallmat_b
    ))#bottom wall along z axis

```

Define engines and set a fixed time step:

```

1 newton=NewtonIntegrator(damping = 0.3,gravity=(0.,0.,-9.8),label=
    "newton",isSuperquadrics=4)
2
3 O.engines=[
4     ForceResetter(),
5     InsertionSortCollider([Bo1_GJKParticle_Aabb(),Bo1_Wall_Aabb()])
6     ],verletDist=0.2*0.01),
7     InteractionLoop(
8         [Ig2_Wall_GJKParticle_GJKParticleGeom(),
9          Ig2_GJKParticle_GJKParticle_GJKParticleGeom()],
10         [Ip2_GJKParticleMat_GJKParticleMat_GJKParticlePhys()], #
11             collision "physics"
12             [GJKParticleLaw()] # contact law — apply forces
13         ),
14     newton
15 ]
16 O.dt=1e-5

```

Polyhedral particles are generated as follows:

```

1 #generate a sample
2 def GenSample(r , pos):
3     for p in pos:
4         body = GJKPolyhedron([], [1.e-2,1.e-2,1.e-2],0.05*1e-2,p_mat,
5         False)
6         body.state.pos=p
7         O.bodies.append(body)
8         O.bodies[-1].shape.color=(rand.random() , rand.random() , rand.
9         random())
10
11 # create a lattice grid
12 pos = GridInitial(R,num_x=num_x,num_y=num_y,num_z=num_z) # get
13     positions of all nodes
14 # create particles at each nodes
15 GenSample(R, pos) #

```

For cones, we change the *GenSample* functions as follows:

```
1 #generate a sample
```

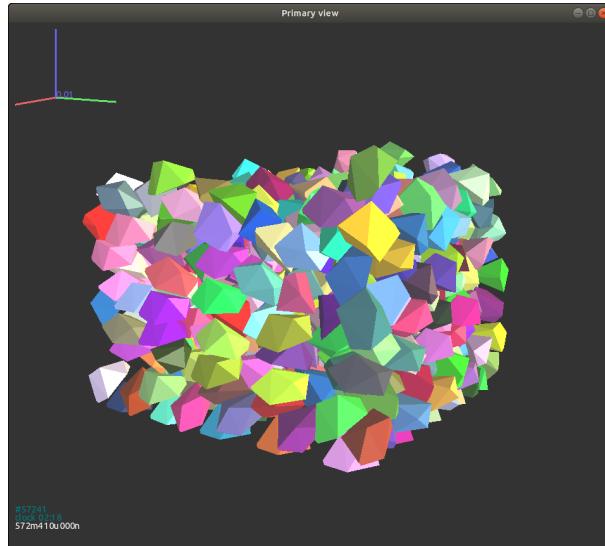


Figure 3.11: Final packing of polyhedrons without random initial orientations (the box not shown).

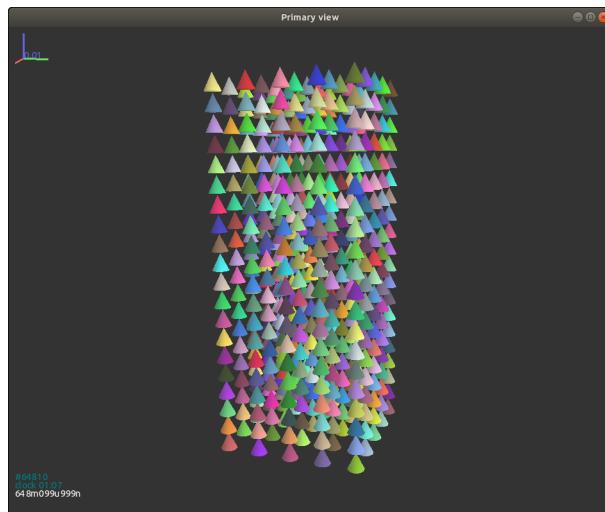


Figure 3.12: Final packing of cones without random initial orientations (the box not shown).

```

2 def GenSample(r , pos) :
3     for p in pos:
4         body = GJKCone(0.005 ,0.01 ,0.05*1e-2,p_mat , False)
5         body.state.pos=p
6         O.bodies.append(body)
7         O.bodies[-1].shape.color=(rand.random() , rand.random() , rand.

```

```
random() )
```

Fig. 3.12 shows final packing of cones without random initial orientations (i.e., the argument *rotate* is False). It can be seen that particles still align in the lattice grid due to no disturbance in the tangential direction.

Set the argument *rotate* to True as follows:

```
1 body = GJKCone(0.005,0.01,0.05*1e-2,p_mat,True)
```

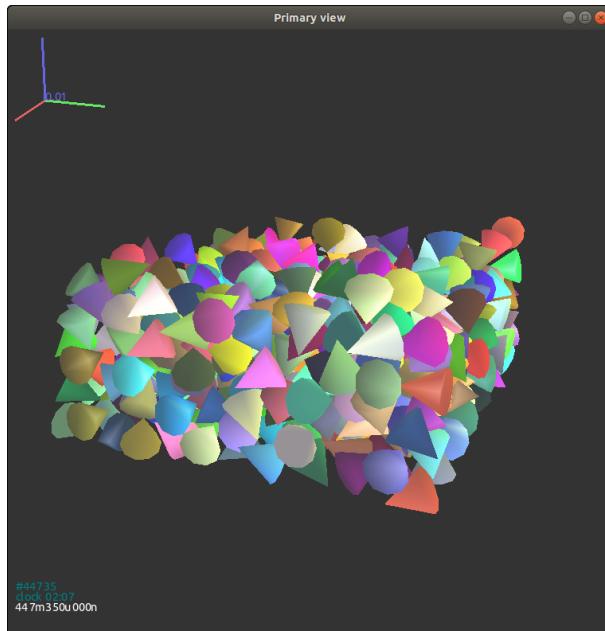


Figure 3.13: Final packing of cones with random initial orientations (the box not shown).

Rerun the simulation, and the initial particles have random orientations. After free falling under gravity, it can be seen that the lattice alignment corrupts, referring to Fig. 3.13

For cylindrical particles, the function *GenSample* is replaced as follows:

```
1 def GenSample(r, pos):
2     for p in pos:
3         body = GJKCylinder(0.005,0.01,0.05*1e-2,p_mat, False)
4         body.state.pos=p
5         O.bodies.append(body)
6         O.bodies[-1].shape.color=(rand.random(), rand.random(), rand.random())
```

Rerun the simulation, we obtain final packing of cylindrical particles as shown in Fig. 3.14, where it is evident that all particles stay in a lattice grid

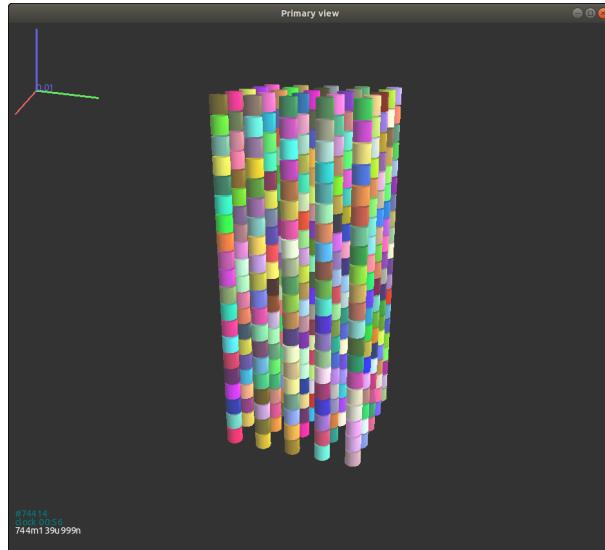


Figure 3.14: Final packing of cylinders without random initial orientations (the box not shown).

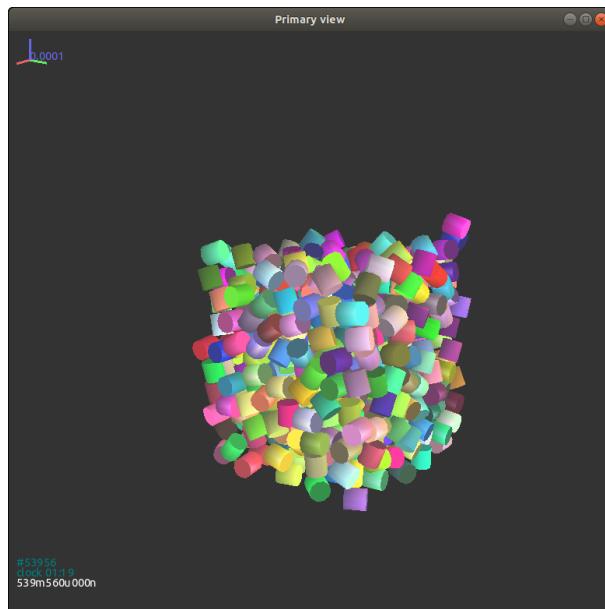


Figure 3.15: Final packing of cylinders with random initial orientations (the box not shown).

after free falling. Again, we reset the argument *rotate* to True so that all initial particles will have random orientations.

```
1 body = GJKCylinder(0.005,0.01,0.05*1e-2,p_mat,True)
```

Rerun the simulation again, the column of cylindrical particles collapse into the cubic box. The final state is visualized in Fig. 3.15.

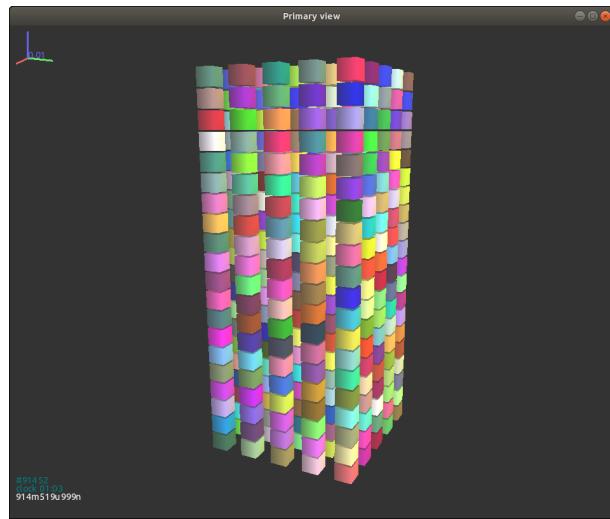


Figure 3.16: Final packing of cuboids without random initial orientations (the box not shown).

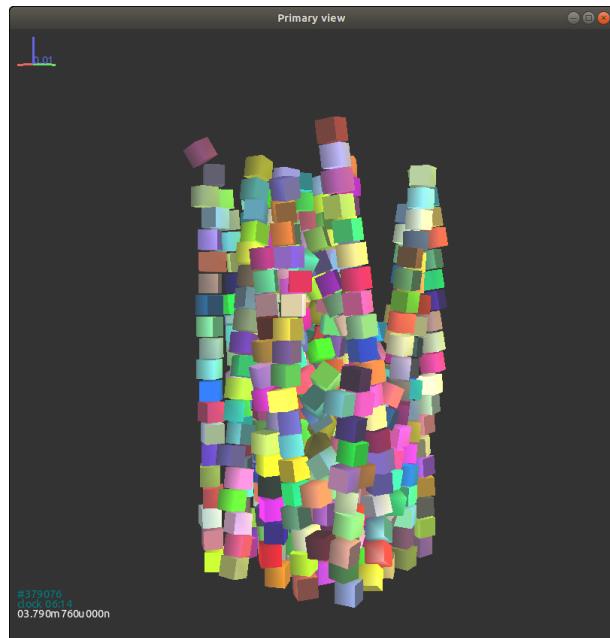


Figure 3.17: Final packing of cuboids with random initial orientations (the box not shown).

We then continue two more simulations of cuboids. Change the function *GenSample* as follows:

```

1 #generate a sample
2 def GenSample(r , pos):
3     for p in pos:
4         body = GJKCuboid([0.005 ,0.005 ,0.005] ,0.05*1e-2,p_mat ,False)
5         body.state.pos=p
6         O.bodies.append(body)
7         O.bodies[-1].shape.color=(rand.random() , rand.random() , rand.
random())

```

We obtain the final packing of cuboids without initial random orientations as shown in Fig. 3.16. Again, the alignment of particles remains a lattice form. Rerun the simulation with the argument *rotate* setting to True as follows:

```

1 body = GJKCuboid([0.005 ,0.005 ,0.005] ,0.05*1e-2,p_mat ,True)

```

The column of particles collapses but not too much due to small gaps between particles.

### 3.5 Example 4: packing of poly-superellipsoids

Poly-superellipsoid has a capability of capturing major features (e.g., flatness, asymmetry, and angularity) of realistic particles such as sands in nature. A poly-superellipsoid is constructed by assembling eight pieces of superellipoids, governed by the following surface function at a local Cartesian coordinate system:

$$\left( \left| \frac{x}{r_x} \right|^{\frac{2}{\epsilon_1}} + \left| \frac{y}{r_y} \right|^{\frac{2}{\epsilon_1}} \right)^{\frac{\epsilon_1}{\epsilon_2}} + \left| \frac{z}{r_z} \right|^{\frac{2}{\epsilon_2}} = 1 \quad (3.11)$$

with

$$r_x = r_x^+ \quad \text{if } x \geq 0 \quad \text{else } r_x^- \quad (3.12a)$$

$$r_y = r_y^+ \quad \text{if } y \geq 0 \quad \text{else } r_y^- \quad (3.12b)$$

$$r_z = r_z^+ \quad \text{if } z \geq 0 \quad \text{else } r_z^- \quad (3.12c)$$

where  $r_x^+, r_y^+, r_z^+$  and  $r_x^-, r_y^-, r_z^-$  are the principal elongation along positive and negative directions of x, y, z axes, respectively;  $\epsilon_1, \epsilon_2$  control the squareness or blockiness of particle surface, and their possible values are in (0, 2) for convex shapes. Fig. 3.18 intuitively shows how  $\epsilon_1$  and  $\epsilon_2$  affect particle surface. The module `_superquadrics_utils` provides two functions to generate a poly-superellipsoid (see Sec. 5.1.2):

- NewPolySuperellipsoid(eps, rxyz, mat, rotate, isSphere, inertiaScale = 1.0)
- NewPolySuperellipsoid\_rot(eps, rxyz, mat,  $q_w, q_x, q_y, q_z$ , isSphere, inertiaScale = 1.0)

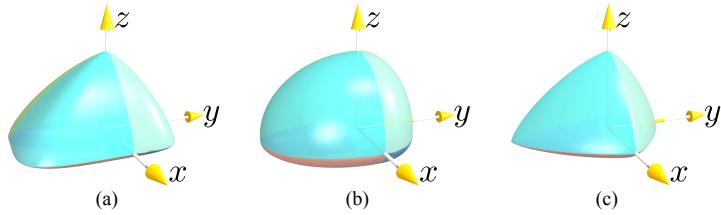


Figure 3.18: Poly-superellipsoids with  $r_x^+ = 1.0, r_x^- = 0.5, r_y^+ = 0.8, r_y^- = 0.9, r_z^+ = 0.4, r_z^- = 0.6$  and (a)  $\epsilon_1 = 0.4, \epsilon_2 = 1.5$ , (b)  $\epsilon_1 = \epsilon_2 = 1.0$ , (c)  $\epsilon_1 = \epsilon_2 = 1.5$ .

Here we give an exemplified script for packing of poly-superellipsoids as follows:

```

1  #
2  from sudodem._superquadrics_utils import *
3  import math
4  import random
5  import numpy as np
6
7  a=1.1734e-2
8  isSphere=False
9  ap=0.4
10 eps=0.5
11 num_s=200
12 num_t=8000
13 trials=0
14
15
16 wallboxsize=[10e-1,0,0] #size of the layer for the particles
   generation
17 boxsize=np.array([7e-1,7e-1,20e-1])
18
19 def down():
20     for b in O.bodies:
21         if(isinstance(b.shape,PolySuperellipsoid)):
22             if(len(b.intrs())==0):
23                 b.state.vel=[0,0,-2]
24                 b.state.angVel=[0,0,0]
25

```

```

26 def gettop():
27     zmax=0
28     for b in O.bodies:
29         if(isinstance(b.shape, PolySuperellipsoid)):
30             if(b.state.pos[2]>=zmax):
31                 zmax=b.state.pos[2]
32     return zmax
33
34
35 def Genparticles(a, eps, ap, mat, boxsize, num_s, num_t, bottom):
36     global trials
37     gap=max(a, a*ap)
38     #print(gap)
39     num=0
40     coor=[]
41     width=(boxsize[0]-2.*gap)/2.
42     length=(boxsize[1]-2.*gap)/2.
43     height=(boxsize[2]-2.*gap)
44     #iteration=0
45
46     while num<num_s and trials<num_t:
47         isOK=True
48         pos=[0]*3
49         pos[0]=random.uniform(-width, width)
50         pos[1]=random.uniform(gap-length, length-gap)
51         pos[2]=random.uniform(bottom+gap, bottom+gap+height)
52
53         for i in range(0, len(coor)):
54             distance=sum([(coor[i][j]-pos[j])**2.] for j in
55 range(0,3)])
56             if(distance<(4.*gap*gap)):
57
58                 isOK=False
59                 break
60
61             if(isOK==True):
62                 coor.append(pos)
63                 num+=1
64                 trials+=1
65                 #print(num)
66
67     return coor
68
69
70 def Addlayer(a, eps, ap, mat, boxsize, num_s, num_t):
71     bottom=gettop()
72     coor=Genparticles(a, eps, ap, mat, boxsize, num_s, num_t, bottom)
73     for b in coor:
74         #xyz = np.array([1.0, 0.5, 0.8, 1.5, 1.2, 0.4]) *0.1
75         xyz = np.array([1.0, 0.5, 0.8, 1.5, 0.2, 0.4]) *0.1

```

```

74     bb=NewPolySuperellipsoid ([1.0 ,1.0] ,xyz ,mat ,True ,isSphere )#
75     bb . state . pos=[b [0] ,b [1] ,b [2]]
76     bb . state . vel=[0,0,-1]
77     O. bodies . append(bb)
78     down()
79
80
81
82 p_mat = PolySuperellipsoidMat (label="mat1" ,Kn=1e5 ,Ks=7e4 ,
83     frictionAngle=math. atan (0.3) ,density=2650,betan=0,betas=0) #
84     define Material with default values
83 wall_mat = PolySuperellipsoidMat (label="wallmat" ,Kn=1e6 ,Ks=7e5 ,
84     frictionAngle=0.0,betan=0,betas=0) #define Material with
84     default values
84 wallmat_b = PolySuperellipsoidMat (label="wallmat" ,Kn=1e6 ,Ks=7e5 ,
85     frictionAngle=math. atan (1) ,betan=0,betas=0) #define Material
85     with default values
85
86 O. materials . append(p_mat)
87 O. materials . append(wall_mat)
88 O. materials . append(wallmat_b)
89
90
91 O. bodies . append( utils . wall (-0.5*wallboxsize [0] ,axis=0,sense=1,
91     material = wall_mat))#left x
92 O. bodies . append( utils . wall (0.5*wallboxsize [0] ,axis=0,sense=-1,
92     material = wall_mat))#right x
93 O. bodies . append( utils . wall (-0.5*boxsize [1] ,axis=1,sense=1,
93     material = wall_mat))#front y
94 O. bodies . append( utils . wall (0.5*boxsize [1] ,axis=1,sense=-1,
94     material = wall_mat))#back y
95
96 O. bodies . append( utils . wall (0 ,axis=2,sense=1, material =wallmat_b)
96     )#bottom z
97 #O. bodies . append( utils . wall (boxsize [0] ,axis=2,sense=-1, material
97     = wallmat_b))#top z
98
99 newton=NewtonIntegrator (damping = 0.3 ,gravity=(0.,0.,-9.8) ,label=
99     "newton" ,isSuperquadrics=2)#isSuperquadrics=2 for Poly-
100     superellipsoids
100
101 O. engines=[
102     ForceResetter () ,
103     InsertionSortCollider ([Bo1_PolySuperellipsoid_Aabb() ,
103         Bo1_Wall_Aabb()] ,verletDist=0.2*0.1) ,
104     InteractionLoop (
105         [Ig2_Wall_PolySuperellipsoid_PolySuperellipsoidGeom() ,
105             Ig2_PolySuperellipsoid_PolySuperellipsoid_PolySuperellipsoidGeom
105             () ] ,

```

```

106 [
107     Ip2_PolySuperellipsoidMat_PolySuperellipsoidMat_PolySuperellipsoidPhys
108     () , # collision "physics"
109     [PolySuperellipsoidLaw()] # contact law — apply forces
110 ),
111 newton
112 ]
113 O. dt=5e-5
114 #adding particles
115 Addlayer(a,eps,ap,p_mat,boxsize,num_s,num_t)#Note: these
116     particles may intersect with each other when we generate them
117 .
118 #setting the Display properties. You can go to the Display panel
119     to set them directly.
120 sudodem.qt.G11_PolySuperellipsoid.wire=False
121 sudodem.qt.G11_PolySuperellipsoid.slices=10
122 sudodem.qt.G11_Wall.div=0 #hide the walls

```

After running the script above, the user is expected to see a packing as shown in Fig. 3.19.

### 3.6 Example 5: packing of superellipses

The surface function of a superellipse in the local Cartesian coordinates can be defined as

$$\left| \frac{x}{r_x} \right|^{\frac{2}{\epsilon}} + \left| \frac{y}{r_y} \right|^{\frac{2}{\epsilon}} = 1 \quad (3.13)$$

where  $r_x$  and  $r_y$  are referred to as the semi-major axis lengths in the direction of x, and y axes, respectively; and  $\epsilon \in (0, 2)$  is the shape parameters determining the sharpness of particle edges or squareness of particle surface. Here we simulate packing of superellipses under gravity with the following script:

```

1 #####
2 from sudodem import utils
3 from sudodem._superellipse_utils import *
4 import numpy as np
5 import math
6 import random
7 random.seed(11245642)
8 #####
9 #####constants#####
10 #####
11 isSphere = False
12
13 r = 0.5 #particle size

```

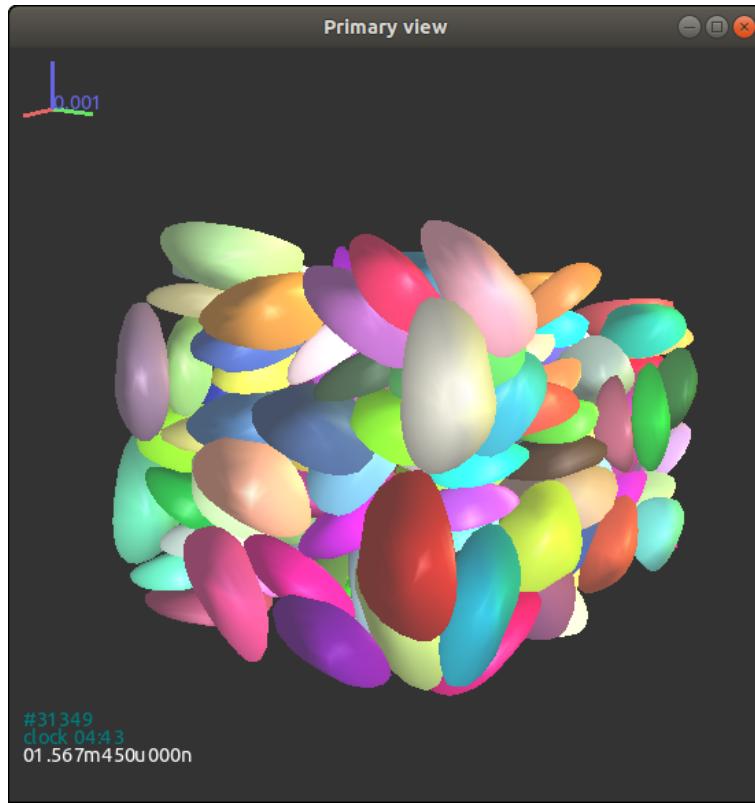


Figure 3.19: Packing of poly-superellipsoids.

```

14 num_s=100
15 num_t=8000
16 trials=0
17 boxsize=[50.0 ,10.0]
18
19 #####
20 mat =SuperellipseMat(label="mat1",Kn=1e8,Ks=7e7,frictionAngle=
    math.atan(0.225),density=2650)
21 O.materials.append(mat)
22
23 O.bodies.append(utils.wall(-0.5*boxsize[1],axis=1,sense=1,
    material = 'mat1'))#ground
24 O.bodies.append(utils.wall(-0.5*boxsize[0],axis=0,sense=1,
    material = 'mat1'))#left
25 O.bodies.append(utils.wall(0.5*boxsize[0],axis=0,sense=-1,
    material = 'mat1'))#right
26
27 ####
28 def gettop():
29     ymax=0

```

```

30     for b in O.bodies:
31         if (isinstance(b.shape, Superellipse)):
32             if (b.state.pos[1]>=ymax):
33                 ymax=b.state.pos[1]
34     return ymax
35
36
37 def Genparticles(r,mat,boxsize,num_s,num_t,bottom):
38     global trials
39     gap=r
40     #print(gap)
41     num=0
42     coor=[]
43     width=(boxsize[0]-2.*gap)/2.
44     height=(boxsize[1]-2.*gap)
45     #iteration=0
46
47     while num<num_s and trials<num_t:
48         isOK=True
49         pos=[0]*2
50         pos[0]=random.uniform(-width,width)
51         pos[1]=random.uniform(bottom+gap,bottom+gap+height)
52
53         for i in range(0,len(coor)):
54             distance=sum([(coor[i][j]-pos[j])**2. for j in
55 range(0,2)])
56             if (distance<(4.*gap*gap)):
57
58                 isOK=False
59                 break
60
61             if (isOK==True):
62                 coor.append(pos)
63                 num+=1
64                 trials+=1
65                 #print(num)
66     return coor
67
68 def Addlayer(r,mat,boxsize,num_s,num_t):
69     bottom=gettop()
70     coor=Genparticles(r,mat,boxsize,num_s,num_t,bottom)
71     for b in coor:
72         rr = r*random.uniform(0.5,1.0)
73         bb = NewSuperellipse(1.5*r,r,1.0,mat,True,isSphere)
74         bb.state.pos=b
75         bb.state.vel=[0,-1]
76         O.bodies.append(bb)
77     if len(O.bodies) - 3 > 2000:

```

```

78     O.engines[-1].dead=True
79
80 O.dt = 1e-3
81
82 newton=NewtonIntegrator(damping = 0.1, gravity=(0., -10.0), label="newton", isSuperellipse=True)
83
84 O.engines=[
85     ForceResetter(),
86     InsertionSortCollider([Bo1_Superellipse_Aabb(), Bo1_Wall_Aabb()],
87     ], verletDist=0.1),
88     InteractionLoop(
89         [Ig2_Wall_Superellipse_SuperellipseGeom(),
90         Ig2_Superellipse_Superellipse_SuperellipseGeom()],
91         [Ip2_SuperellipseMat_SuperellipseMat_SuperellipsePhys()], # collision "physics"
92         [SuperellipseLaw()] # contact law — apply forces
93     ),
94     newton,
95     PyRunner(command='Addlayer(r,mat,boxsize,num_s,num_t)', virtPeriod=0.1, label='check', dead = False)
96 ]

```

After running the above script with *sudodem2d*, the user can get a packing as shown in Fig. 3.20.

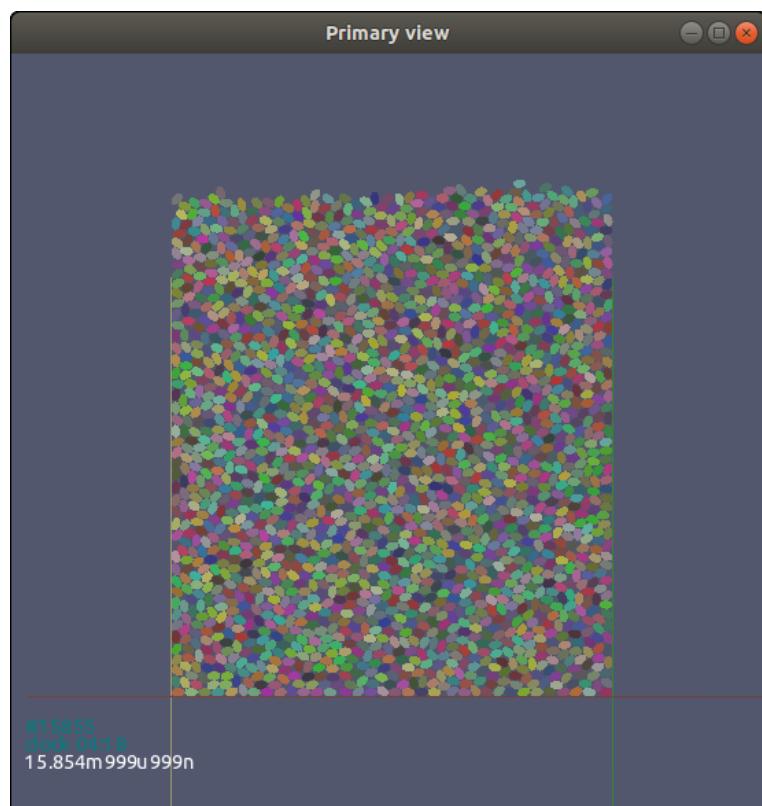


Figure 3.20: Packing of superellipses under gravity.

# Chapter 4

## Post-processing

### 4.1 Data

It is preferable to save a series of states during a simulation, i.e., periodically saving data using a PyRunner:

```
1 def savedata():
2     global saving_num
3     saving_num += 1
4     #print "testing , no data saved"
5     O.save('shear'+str(saving_num)+'.xml.bz2')
6
7 O.engines = O.engines + [PyRunner( command = 'savedata()',
8     iterPeriod = 1000 )] # if we load the consolidation data,
9     then we do not need this line because the PyRunner calling '
10    'savedata()' has been defined.
```

The Python module *utilspost* provides plentiful examples of post-processing, including some basic functions of stress-force-fabric computation and other pre-processing functions for visualization (e.g., writing vtk files of particles and 3D histograms of fabric). Based on this module, users can copy and modify for a self purpose or just call the module functions in a post-processing script, e.g.,

```
1 import sudodem.utilspost as utilspost
2 #the following functions will output the average friction
3     mobilization index of each state to a single file.
4 for i in ['1.0','1.2','1.4','1.6']:
5     utilspost.calc_avgFricMobilIndex(i,steps_num)
```

The Python module *\_superquadrics\_utils* also provides some auxiliary functions, e.g.,

- *outputParticles(filename)* : *filename*, the file name for writing; output info of each particle, i.e., shape parameters  $r_x$ ,  $r_y$ ,  $r_z$ ,  $\epsilon_1$ ,  $\epsilon_2$ , position

$x, y, z$ , and orientation  $q_0, q_1, q_2, q_3$  (equivalent to  $q_w, q_x, q_y, q_z$  line by line. respectively).

- `outputParticlesByIds(filename, ids)` : *filename*, the file name for writing; *ids*, a list of ids for particles needed outputting; output info of each particle in the list *ids*, i.e., shape parameters  $r_x, r_y, r_z, \epsilon_1, \epsilon_2$ , position  $x, y, z$ , and orientation  $q_0, q_1, q_2, q_3$  (equivalent to  $q_w, q_x, q_y, q_z$  line by line. respectively).
- `outputWalls(filename)` : *filename*, the file name for writing; output positions of all walls.
- `outputPOV(filename)` : *filename*, the file name for writing; output a POV-Ray file of particles for render using POV-Ray.
- `outputVTK(filename, slices)` : *filename*, the file name for writing; *slices*, number of slices a particle needed slicing in longitude; output a vtk file of particles for render using Paraview.

## 4.2 Scene Visualization

### 4.2.1 SudoDEM3D

For visualization, two free, open-source post-processing softwares, *Paraview*<sup>1</sup> and *POV-Ray*<sup>2</sup>, are strongly recommended.

It is preferable to reproduce a high-resolution figure of particle configuration using post-processing softwares instead of a screenshot. As a demonstration, we load a state file named "shearend.xml.bz2" and output a vtk file or a pov file for post visualization.

```

1 O.load("shearend.xml.bz2")
2 import sudodem._superquadrics_utils as superutils
3 superutils.outputVTK("shearend.vtk", 15)

```

Open *Paraview*, then import the vtk file "shearend.vtk" and click the button "Apply", so that the configuration of particles is visualized as shown in Fig. 4.2. It is noteworthy that the function *outputVTK* processes a particle surface as a combination of many (determined by the argument *slices*, i.e., 15 herein) polygons, which would yield a large file (around 180 Mb in the presented example). The users may find ways to reduce the vtk file for a faster visualization in *Paraview*. One possible way is to show only the particles

<sup>1</sup><https://www.paraview.org/>

<sup>2</sup>[www.povray.org/](http://www.povray.org/)

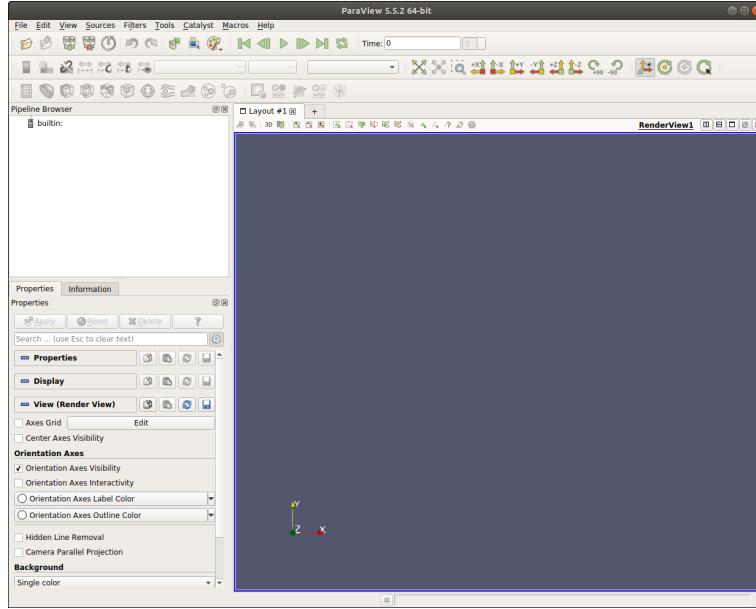


Figure 4.1: The main window of Paraview.

exposed in the view. Another possible way is to use the interface of built-in *Superquadric* source in *Paraview*. Another alternative approach is rendering the scene utilizing *POV-Ray*.

```

1 O.load("shearend.xml.bz2")
2 import sudodem._superquadrics_utils as superutils
3 superutils.outputPOV("shearend.pov")

```

The users might revise the parameters of the camera in the POV file, i.e., "shearend.pov" here,

```

1 camera{ //orthographic angle 45
2   location <0.6,0.6,0.5>*50e-3
3   sky z
4   look_at <0,0,0.05>*10e-3
5   right x*image_width/image_height
6   translate <-0.04,0.01,-0.1>*50e-3
7 }

```

Light sources should be slightly adjusted as well.

```

1 // White background
2 background{rgb 1}
3 // Two lights with slightly different colors
4 light_source{<4,8,5>*10e-3 color rgb <1,1,1>}
5 light_source{<12,-6>*10e-3 color rgb <1,1,1>}

```

In a terminal, run the following command to render the scene:

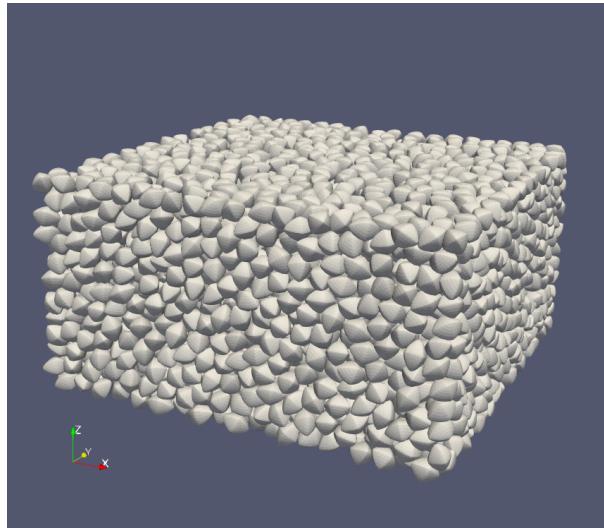


Figure 4.2: Configuration of particles rendered by *Paraview*.

---

```
1 povray shearend.pov
```

---

For a high-resolution render, additional arguments are appended to the command,

---

```
1 povray shearend.pov +A0.01 +W1600 +H1200
```

---

Image processing softwares, e.g., *GIMP*<sup>3</sup>, can be used to change image size or make a gif animation, etc. Fig. 4.3 shows configuration of particles rendered by *POV-Ray* and cropped using *GIMP*.

The module *snapshot* provides functions for visualizing poly-superellipsoids. As an example, we load the state of final packing, and import the module and call the *outPov* function as follows:

---

```
1 from sudodem import snapshot
2 snapshot.outPov("packingpolysuper", withbox=False)
```

---

we then get two files *packingpolysuper.inc* and *packingpolysuper.pov*. The first file *packingpolysuper.inc* includes macros for the definition of poly-superellipsoid and the particle information such as shape parameters, positions and orientations, like this:

---

```
1 #macro myremain(sign1, sign2, sign3)
2   clipped_by{plane{-sign1*x, 0}}
3   clipped_by{plane{-sign2*y, 0}}
4   clipped_by{plane{-sign3*z, 0}}
```

---

<sup>3</sup><https://www.gimp.org/>

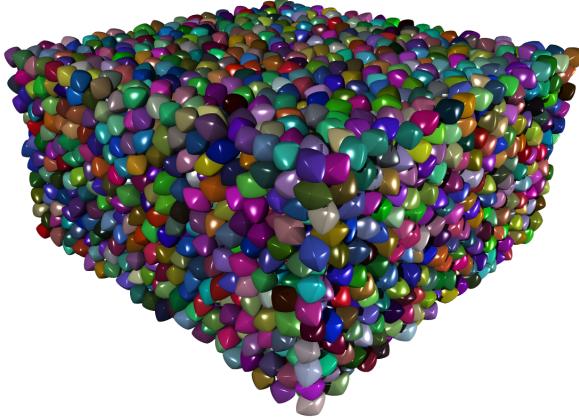


Figure 4.3: Configuration of particles rendered by *POV-Ray*.

```

5 #end
6 #macro randomColor(r,g,b)
7 texture{ pigment{ color rgb<r,g,b> transmit para_trans} finish {
8     phong para_phong }}
9 #end
10 #macro octantSuper(ep1,ep2,rx,ry,rz,sign1,sign2,sign3)
11     superellipsoid{ <ep1,ep2>
12 #if (singleColor = false)
13     randomColor(rand(Random_r),rand(Random_g),rand(Random_b))
14 #end
15 myremain(sign1,sign2,sign3) scale <rx,ry,rz>
16 #end
17 #macro polySuperEllipsoid(ep1,ep2,a1,a2,b1,b2,c1,c2,t1,t2,t3,r1,
18     r2,r3,r,g,b)
19 union{
20     octantSuper(ep1,ep2,a2,b2,c2,-1,-1,-1)
21     octantSuper(ep1,ep2,a1,b2,c2,1,-1,-1)
22     octantSuper(ep1,ep2,a2,b1,c2,-1,1,-1)
23     octantSuper(ep1,ep2,a1,b1,c2,1,1,-1)
24     octantSuper(ep1,ep2,a2,b2,c1,-1,-1,1)
25     octantSuper(ep1,ep2,a1,b2,c1,1,-1,1)
26     octantSuper(ep1,ep2,a2,b1,c1,-1,1,1)
27     rotate<r1,r2,r3>
28     translate<t1,t2,t3>
29 #if (singleColor = true)
30     randomColor(r,g,b)
31 #end
32 }
33 polySuperEllipsoid(1.000000e+00,1.000000e+00,1.000000e-01,
```

```

5.0000000e-02,8.0000000e-02,1.5000000e-01,2.0000000e-02,
4.0000000e-02,-3.2218276e-01,1.8390308e-01,3.1674861e-01,
-167.727379,16.417442,-104.719219,0.783099,0.394383,0.840188)
34 ...
35 ...

```

The users are not suggested to edit this file though. But the second file *packingpolysuper.pov* includes some editable parameters as introduced above, and looks like this:

```

1 //using the command: povray **.pov +A0.01 +W1600 +H1200
2 #include "colors.inc"
3 #declare singleColor = true;
4 #declare Random_r = seed (1432);
5 #declare Random_g = seed (7242);
6 #declare Random_b = seed (9912);
7 #declare para_trans = 0 ;
8 #declare para_phong = 1.0 ;
9 camera {
10 location < 1.39244459003 , 0.950562440499 , 0.605226784362 >
11 sky z
12 right -x*image_width/image_height
13 look_at < 0.612872382451 , 0.349255160637 , 0.430021966053 >
14 }
15 #include "packingellipses.inc"
16 light_source{<4,8,5>*10 color rgb <1,1,1>}
17 light_source{<12,-6>*10 color rgb <1,1,1>}
18 light_source { <0, 2, 10> White }
19 background{rgb 1}
20 plane { z, -5 pigment { checker Green White }}//you may comment
      out this line to remove the plane

```

Then render it in a fresh terminal by typing

```
1 povray packingpolysuper.pov +A0.01 +W1600 +H1200
```

you will get a nice figure with higher quality as Fig. 4.4.

### 4.2.2 SudoDEM2D

Module \_superellipse\_utils provides a function *drawSVG* (see Sec. 5.2.2) to dump the particle profiles and contact forces chains to a SVG file.

```

1 from sudodem import _superellipse_utils
2 O.load("your data file")
3 drawSVG("test.svg", draw_forcechain=True,
4         force_line_color=(1.0,0,0),
5         solo_color=(93.0/255,152.0/255,208.0/255))

```

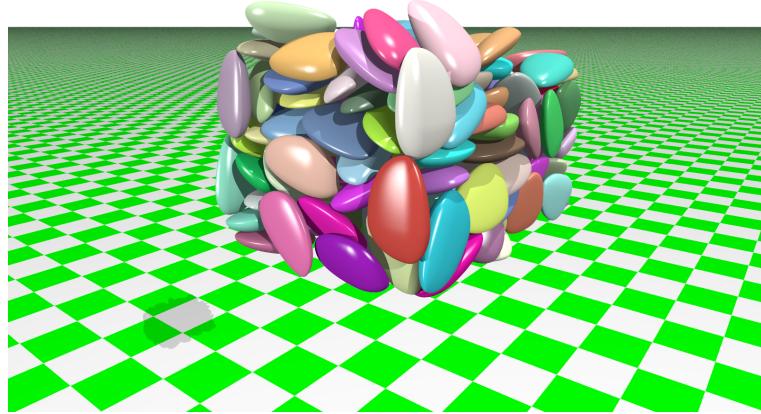


Figure 4.4: Packing of poly-superellipsoids in Example 4 rendered by *POV-Ray*.

The output file test.svg is editable by using a text editor and/or Inkscape (recommended). Fig. 4.5 shows an exemplified snapshot of a packing of elliptic particles with periodic boundary conditions. Refer to the keywords of *drawSVG* for configuring the output figure.

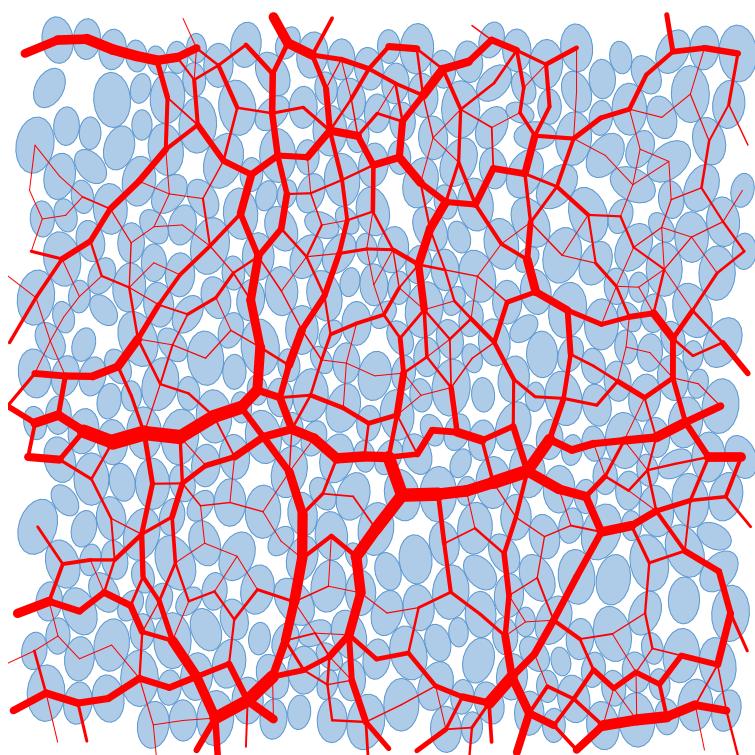


Figure 4.5: Particles and contact force chains dumped by *drawSVG*.

# Chapter 5

## Python Class Reference

### 5.1 SudoDEM3D

#### 5.1.1 Basic classes

```
class State(inherits Serializable)
State of a body (spatial configuration, internal variables).
```

- `angMom(=Vector3r::Zero())`  
Current angular momentum
- `angVel(=Vector3r::Zero())`  
Current angular velocity
- `blockedDOFs`  
Degrees of freedom where linear/angular velocity will be always constant (equal to zero, or to an user-defined value), regardless of applied force/torque. String that may contain ‘xyzXYZ’ (translations and rotations).
- `dict() → dict`  
Return dictionary of attributes.
- `inertia(=Vector3r::Zero())`  
Inertia of associated body, in local coordinate system.
- `isDamped(=true)`  
Damping in Newtonintegrator can be deactivated for individual particles by setting this variable to FALSE. E.g. damping is inappropriate for particles in free flight under gravity but it might still be applicable to other particles in the same simulation.

- `mass(=0)`  
Mass of this body
- `ori`  
Current orientation.
- `pos`  
Current position.
- `refOri(=Quaternionr::Identity())`  
Reference orientation
- `refPos(=Vector3r::Zero())`  
Reference position
- `se3(=Se3r(Vector3r::Zero(), Quaternionr::Identity()))`  
Position and orientation as one object.

### 5.1.2 Module `_superquadrics_utils`

- `NewSuperquadrics2( $r_x$ ,  $r_y$ ,  $r_z$ ,  $\epsilon_1$ ,  $\epsilon_2$ , mat, rotate, isSphere)`  
Generate a Superquadric with isSphere option.
  - $r_x$ ,  $r_y$ ,  $r_z$ : float, semi-major axis lengths in x, y, and z axes.
  - $\epsilon_1$ ,  $\epsilon_2$ : float, shape parameters.
  - mat: Material, a material attached to the particle.
  - rotate: bool, with a random orientation or not.
  - isSphere: bool, particle is spherical or not.
- `NewSuperquadrics_rot2( $r_x$ ,  $r_y$ ,  $r_z$ ,  $\epsilon_1$ ,  $\epsilon_2$ , mat,  $q_w$ ,  $q_x$ ,  $q_y$ ,  $q_z$ , isSphere)`  
Generate a Super-ellipsoid with specified quaternion components:  $q_w$ ,  $q_x$ ,  $q_y$ ,  $q_z$  with isSphere option.
  - $r_x$ ,  $r_y$ ,  $r_z$ : float, semi-major axis lengths in x, y, and z axes.
  - $\epsilon_1$ ,  $\epsilon_2$ : float, shape parameters.
  - mat: Material, a material attached to the particle.
  - isSphere: bool, particle is spherical or not.
  - $q_w$ ,  $q_x$ ,  $q_y$ ,  $q_z$ : float, components of a quaternion ( $q_w$ ,  $q_x$ ,  $q_y$ ,  $q_z$ ) representing the orientation of a particle.
- `outputWalls(filename)`  
Output positions of all walls in the scene.

- filename: string, the name (with path) of the output file.
- getSurfArea(id, w, h)
 

Get the surface area of a super-ellipsoid by a given particle id with resolution w and h (e.g., w=10,h=10). Larger w and h will yield more accurate results.

  - id: int, the id of the given particle.
  - w, h: int, w×h slices the particle surface will be discretized into.
- outputParticles(filename)
 

Output information including id, shape parameters, position and orientation of all particles into a file.

  - filename: string, the name (with path) of the output file.
- outputParticlesByIds(filename, ids)
 

Output particles by a list of ids, referring to the last function. [Only for superellipsoids]

  - filename: string, the name (with path) of the output file.
  - ids: list of int, a list of particle ids, e.g., [10, 24, 22].
- NewPolySuperellipsoid(eps, rxyz, mat, rotate, isSphere, inertiaScale = 1.0)
 

Generate a PolySuperellipsoid.

  - eps: Vector2, shape parameters, i.e.,  $[\epsilon_1, \epsilon_2]$ .
  - rxyz: Vector6, semi-major axis lengths in x, y, and z axies, i.e.,  $[r_x^-, r_x^+, r_y^-, r_y^+, r_z^-, r_z^+]$ .
  - mat: Material, a material attached to the particle.
  - rotate: bool, with a random orientation or not.
  - isSphere: bool, particle is spherical or not.
  - inertialScale: float, scale the moment of inertia only. Do not set it if you are not sure the side effect.
- NewPolySuperellipsoid\_rot(eps, rxyz, mat, qw, qx, qy, qz, isSphere, inertiaScale = 1.0)
 

Generate a PolySuperellipsoid with specified quaternion components:  $q_w, q_x, q_y$  and  $q_z$ .

  - eps: Vector2, shape parameters, i.e.,  $[\epsilon_1, \epsilon_2]$ .

- *rxyz*: Vector6, semi-major axis lengths in x, y, and z axes, i.e., [ $r_x^-$ ,  $r_x^+$ ,  $r_y^-$ ,  $r_y^+$ ,  $r_z^-$ ,  $r_z^+$ ].
- *mat*: Material, a material attached to the particle.
- *isSphere*: bool, particle is spherical or not.
- $q_w$ ,  $q_x$ ,  $q_y$  and  $q_z$ : float, components of a quaternion ( $q_w$ ,  $q_x$ ,  $q_y$ ,  $q_z$ ) representing the orientation of a particle.
- *inertialScale*: float, scale the moment of inertia only. Do not set it if you are not sure the side effect.
- **outputPOV(filename)**  
Output Superellipsoids (including spheres) into a POV file for post-processing using POV-ray. [Only for superellipsoids and spheres]
  - *filename*: string, the name (with path) of the output file.
- **PolySuperellipsoidPOV(filename, ids = [], scale = 1.0)**  
Output PolySuperellipsoids into a POV file for post-processing using POV-ray. This function only output the particles' information to a POV file like \*.inc, and the user may need to add some other info such as camera etc. See the module *snapshot*.
  - *filename*: string, the name (with path) of the output file.
  - *ids*: list of int, a list of particle ids. A NULL list will output all particles by default.
  - *scale*: float, scale the length unit.
- **outputVTK(filename, slices)**  
Output Superellipsoids into a VTK file for post-processing using Paraview. [Only for superellipsoids]
  - *filename*: string, the name (with path) of the output file.
  - *slices*: int, into how many slices the particle surface will be discretized.

### 5.1.3 Module `_gjkparticle_utils`

- **GJKSphere(radius, margin, mat)**
  - *radius*, float, radius of the sphere.
  - *margin*, float, a small margin added to the surface for assisting contact detection.

- *mat*, Material, a specified material attached to the particle.
- GJKPolyhedron(*vertices, extent, margin, mat, rotate*)
  - *vertices*, a list of vertices, and each vertex is a list, e.g.,  $[x, y, z]$  for coordinates.
  - *extent*, a list of three components to scale the x, y, z dimensions; valid only if *vertices* has no elements (i.e.,  $[]$ ); a randomly shaped polyhedron will be generated based on Voronoi tessellation.
  - *margin*, float, a small margin added to the surface for assisting contact detection.
  - *mat*, Material, a specified material attached to the particle.
  - *rotate*, bool, with random orientation or not.
- GJKCone(*radius, height, margin, mat, rotate*)
  - *radius*, float, base radius of the cone.
  - *height*, float, height of the cone.
  - *margin*, float, a small margin added to the surface for assisting contact detection.
  - *mat*, Material, a specified material attached to the particle.
  - *rotate*, bool, with random orientation or not.
- GJCKylinder(*radius, height, margin, mat, rotate*)
  - *radius*, float, radius of the cylinder.
  - *height*, float, height of the cylinder.
  - *margin*, float, a small margin added to the surface for assisting contact detection.
  - *mat*, Material, a specified material attached to the particle.
  - *rotate*, bool, with random orientation or not.
- GJKCuboid(*extent, margin, mat, rotate*)
  - *extent*, a list of three floats, extent of the cuboid in the x, y, z dimensions.
  - *margin*, float, a small margin added to the surface for assisting contact detection.
  - *mat*, Material, a specified material attached to the particle.

- *rotate*, bool, with random orientation or not.

Note: a margin is used to expand a particle for achieving a better computational efficiency, which however should be large enough for ensuring that collision occurs in this small region but small enough for shape fidelity.

#### 5.1.4 Module snapshot

- `outputBoxPov(filename, x, y, z, r=0.001, wallMask=[1,0,1,0,0,1])`  
Output \*.POV of a cubic box ( $x=[x_{\text{min}}, x_{\text{max}}]$ ,  $y=[y_{\text{min}}, y_{\text{max}}]$ ,  $z=[z_{\text{min}}, z_{\text{max}}]$ ) for post-processing in Pov-ray.
  - `filename`: string, the name (with path) of the output file.
  - `x, y, z`: Vector2, ranges of the cubic box along the three (x, y, z) directions.
  - `r`: float, thickness of the box edge.
  - `wallMask`, list of 6 int, corresponding to the six walls at (x-, x+, y-, y+, z-, z+). 1 to show and 0 to hide the wall.
- `outPov(fname,transmit=0,phong=1.0,singleColor=True,ids=[],scale=1.0, withbox = True)`
  - `filename`, string, the name (with path) of the output file.
  - `transmit`, float, parameter for transparency of particles, ranging between 0 and 1.
  - `phong`, float, parameter for highlighting the surface, ranging between 0 and 1.
  - `singleColor`, bool, whether to use a single color for all particles. If not, particles' colors would be the values defined in Shape.
  - `ids`, list of int, a list of particle ids that will be output. If no item are given, then all particles will be output by default.
  - `scale`, float, scale the unit.

The user is referred to POV-Ray's Doc for more details on the two parameters (transmit and phong) and others that will be written to the output file by this function.

## 5.2 SudoDEM2D

### 5.2.1 Basic classes

#### Class State

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

- `angMom(=Vector2r::Zero())`  
Current angular momentum
- `angVel(=Vector2r::Zero())`  
Current angular velocity
- `blockedDOFs`  
Degrees of freedom where linear/angular velocity will be always constant (equal to zero, or to an user-defined value), regardless of applied force/torque. String that may contain ‘xyZ’ (translations and rotations).
- `dict() → dict`  
Return dictionary of attributes.
- `inertia(=Vector2r::Zero())`  
Inertia of associated body, in local coordinate system.
- `isDamped(=true)`  
Damping in Newtonintegrator can be deactivated for individual particles by setting this variable to FALSE. E.g. damping is inappropriate for particles in free flight under gravity but it might still be applicable to other particles in the same simulation.
- `mass(=0)`  
Mass of this body
- `ori`  
Current orientation. The orientation and rotation of a particle is represented by a Rotation2d object (see [Rotation2D](#) in the Eigen library), which has the following members and functions.  
Class Rotation2d
  - `angle`: float, rotation represented by an angle in rad.
  - `toRotationMatrix()`: `Matrix2r`, returns an equivalent 2x2 rotation matrix.

- Identity(): Roation2d, returns an identity rotation.
- smallestAngle(): float, returns the rotation angle in  $[-\pi, \pi]$
- inverse(): Roation2d, returns the inverse rotation
- smallestPositiveAngle(): float, returns the rotation angle in  $[0, 2\pi]$
- pos  
Current position.
- refOri(=Rotation2D::Identity())  
Reference orientation
- refPos(=Vector2r::Zero())  
Reference position
- se2(=Se2r(Vector2r::Zero(), Rotation2d::Identity()))  
Position and orientation as one object.
- vel(=Vector2r::Zero()) Current linear velocity.

### 5.2.2 Module `_superellipse_utils`

- NewSuperellipse(rx, ry, epsilon, material, rotate, isSphere, z\_dim = 1.0)  
Create a Superellipse
  - rx, ry: float, semi-axis length of the particle
  - epsilon: float, shape parameter of a superellipse
  - material: Material instance
  - rotate: bool, rotate the particle with random orientation?
  - isSphere: bool, is the superellipse a disk? A disk will speed up the computation.
  - z\_dim: float, the virtual length at the z dimension, and set to 1.0 by default.
- NewSuperellipse\_rot(x, y, epsilon, material, miniAngle, isSphere, z\_dim = 1.0)  
Create a superellipse with certain orientation
  - rx, ry: float, semi-axis length of the particle
  - epsilon: float, shape parameter of a superellipse

- material: Material instance
- miniAngle: orientation of the particle specified by the angle between its rx-axis and the global x axis
- isSphere: bool, is the superellipse a disk? A disk will speed up the computation.
- z\_dim: float, the virtual length at the z dimension, and set to 1.0 by default.
- outputParticles(filename)
 

output particle info(rx,ry,eps,x,y,rotation angle) to a text file
- drawSVG(filename, width\_cm = 10.0, slice = 20, line\_width = 0.1, fill\_opacity = 0.5, draw\_filled = true, draw\_lines = true, color\_po = false, po\_color = Vector3r(1.0,0.0,0.0), solo\_color = Vector3r(0,0,0), force\_line\_width = 0.001, force\_fill\_opacity = 0, force\_line\_color = Vector3r(1.0,1.0,1.0), draw\_forcechain = false)
 

Output particle profiles and contact force chains into a SVG file.

  - filename: string, the name of the output file
  - width\_cm: float, width (in centimeters) in the SVG header
  - slice: int, to how many slices a Superellipse will be discretized
  - line\_width: float, the line width of the edge of a Superellipse
  - fill\_opacity: float, the opacity of the fill color in a Superellipse
  - draw\_filled: bool, wheter to fill a Superellipse with a color
  - draw\_lines: bool, whether to draw the out profile of a Superellipse
  - color\_po: bool, whether to use a fill color to identify the orientation of a Superellipse
  - po\_color: Vector3r, the fill color to identify the orientation of a Superellipse
  - solo\_color: Vector3r, a solor color to fill a Superellipse. If its norm is zero, then use the color defined by po\_color or shape→color
  - force\_line\_width: float, the line width of the force chain with the average normal contact force
  - force\_fill\_opacity: float, the opacity of force chains
  - force\_line\_color: Vector3r, the color of the force chain
  - draw\_forcechain: bool, whether to draw the contact force chain that will be superposed with the particles

### 5.2.3 Module `_utils`

- `unbalancedForce(useMaxForce=false)`

Compute the ratio of mean (or maximum, if \*useMaxForce\*) summary force on bodies and mean force magnitude on interactions. For perfectly static equilibrium, summary force on all bodies is zero (since forces from interactions cancel out and induce no acceleration of particles); this ratio will tend to zero as simulation stabilizes, though zero is never reached because of finite precision computation. Sufficiently small value can be e.g. 1e-2 or smaller, depending on how much equilibrium it should be.

- `getParticleVolume2D()`

Compute the total volume (area for 2D) of particles in the scene.

- `getMeanCN()`

Get the mean coordination number of a packing

- `changeParticleSize2D(alpha)`

expand a particle by a given coefficient

- `getVoidRatio2D(cellArea=1)`

Compute 2D void ratio. Keyword cellArea is effective only for aperiodic cell

- `getStress2D(z_dim=1)`

Compute overall stress of periodic cell

- `getFabricTensorCN2D()`

Fabric tensor of contact normal

- `getFabricTensorPO2D()`

Fabric tensor of particle orientation along the rx axis

- `getStressAndTangent2D(z_dim=1,symmetry=true)`

Compute overall stress of periodic cell using the same equation as function `getStress`. In addition, the tangent operator is calculated using the equation:  $S_{ijkl} = \frac{1}{V} \sum_c (k_n n_{il} n_{kj} + k_t t_{il} t_{kj})$  float volume: same as in function `getStress` bool symmetry: make the tensors symmetric.

return: macroscopic stress tensor and tangent operator as py::tuple

- `getStressTangentThermal2D(z_dim=1, symmetry =true)`

Compute overall stress of periodic cell using the same equation as function `getStress`. In addition, the tangent operator is calculated using

the equation:  $S_{ijkl} = \frac{1}{V} \sum_c (k_n n_i l_j n_k l_l + k_t t_i l_j t_k l_l)$  float volume: same as in function getStress bool symmetry: make the tensors symmetric. Finally, the thermal conductivity tensor is calculated based on the formula in PFC. macroscopic stress tensor and tangent operator as py::tuple

### 5.2.4 Module utils

- **disk**(center, radius, z\_dim=1, dynamic=None, fixed=False, wire=False, color=None, highlight=False, material=-1, mask=1)  
 Create a disk with given parameters; mass and inertia computed automatically.
  - center: Vector2, center
  - radius: float, radius
  - dynamic: float, deprecated, see "fixed"
  - fixed: float, generate the body with all DOFs blocked?
  - material: specify ‘Body.material’; different types are accepted:
    - \* int: O.materials[material] will be used; as a special case, if material== -1 and there is no shared materials defined, utils.defaultMaterial() will be assigned to O.materials[0]
    - \* string: label of an existing material that will be used
    - \* ‘Material’ instance: this instance will be used
    - \* callable: will be called without arguments; returned Material value will be used (Material factory object, if you like)
  - int mask: ‘Body.mask’ for the body
  - wire: display as wire disk?
  - highlight: highlight this body in the viewer?
  - Vector2-or-None: body’s color, as normalized RGB; random color will be assigned if “None”.
- **wall**(position, axis, sense=0, color=None, material=-1, mask=1)  
 Return ready-made wall body.
  - 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} , orientation of the wall normal (0,1) for x,y

- sense: {-1,0,1} , sense in which to interact (0: both, -1: negative, +1: positive; see ‘Wall’)

See ‘utils.disk’’s documentation for meaning of other parameters.

- fwall(vertex1, vertex2, dynamic=None, fixed=True, color=None, highlight=False, noBound=False, material=-1, mask=1, chain=-1)  
Create fwall with given parameters.

- vertex1: Vector2, coordinates of vertex1 in the global coordinate system.
- vertex2: Vector2, coordinates of vertex2 in the global coordinate system.
- noBound: bool, set ‘Body.bounded’
- color: Vector3-or-None , color of the facet; random color will be assigned if ‘None’.

See ‘utils.disk’’s documentation for meaning of other parameters.

# Acknowledgments

The work was partially supported by the Hong Kong Scholars Program (2018), the National Natural Science Foundation of China (by Project No. 51679207, 51909095), Research Grants Council of Hong Kong (by GRF Project No. 16205418, TBRs Project No. T22-603/15N and CRF Project No. C6012-15G). The following open-source tools but not limited to are acknowledged: [Ubuntu](#), [YADE](#), [Python](#), [Boost](#), [Eigen](#), [Voro++](#), [Paraview](#), [Pov-Ray](#), [GIMP](#), [InkScape](#), [Veusz](#) and [LaTeX](#).