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

Shiwei Zhao[1,2] and Jidong Zhao[1]

[1] *The Hong Kong University of Science and Technology*
[2] *South China University of Technology*

Updated on June 13, 2020, version 1.2.1

Contents

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[1]. 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.

[1] https://yade-dev.gitlab.io/trunk/

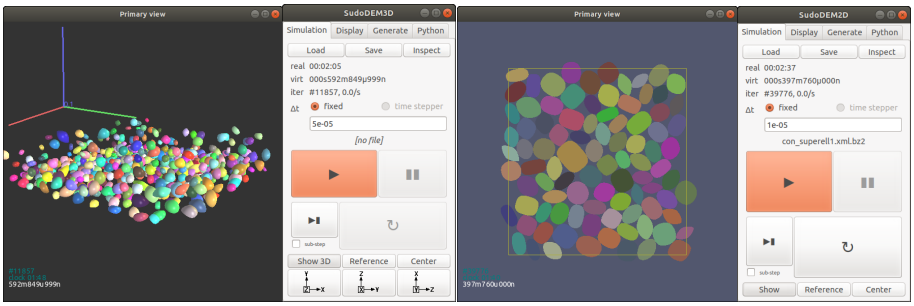


Figure 1.1: GUIs of *SudoDEM*^{3D} (*left*) and *SudoDEM*^{2D} (*right*).

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. *Gétechnique*, 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 *package*[1] 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*.

[1] or using sudo apt install *package*

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

```
$> sudo apt-get install python-numpy
$> pwd
$> cd /home/
$> 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:

```
import math # import the module math which includes some basic
    math functions

# we have numbers
a = 2 # integer number
b = 5
c = 2.0 # float number
d = a/b # the value is an integer
e = c/b # float number
print a, d, e # print the values of a, d and e to the terminal

#we have strings
a = 'abc' # we can assign any type of values to any type of
    variables
b = "this is a 'Python' script."
print a, b

#we have list, tuple and dict to store and manage the above basic
     primitives
c = [1,2,3,'ss'] # a list
d = (1,2,3,'ss') # a tuple
c[0] = 5 # change the first item in the list c, and the index
    starts from 0.
d[0] = 5 # failed! a tuple is constant, which is the
    distinguishing property from a list
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.
print d[1], d[4], d['w']

# we initialize two objects c and d for storing data at a for
    loop later
c = list() # an empty list
```

```
d = dict() # an empty dict
#test expression
if 1>2:
  print "1 is greater than 2, really?"
else:
  if 2 in [1, 2, 3]: # if 2 is in the list [1, 2, 3]
    #ok, let's start a for loop
    for i in range(5): # equal to for i in [0, 1, 2, 3, 4]
      print i # you can see what's going on in the for loop
      # we append some data to a list c and a dict d
      c.append(math.cos(i)*10 + 2.0**5) # we append the value of 10*cos(i) + 2.0^5 to the list c. We use the math function cos from the module math.
      if i not in d.keys(): # if i is not a key or index of the dict d.
        d[i] = str(i)

#we can capsule our commands by a function
def GoodJob(input_number, keyword1 = 1, keyword2 = True):
  if keyword2: # that is if keyword2 == True
    print "the input number is", input
  input_number += keyword1 # i.e., the value of input_number is updated to input_number + keyword1
  return input_number # we can return the value

# call the function
a = GoodJob(2) # a = 3, with print info 'the input number is 3'
b = GoodJob(3, keyword1 = 5, keyword2 = False) # b = 8, without 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[2]

[2]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':

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

2.3.2 Non-root Compilation

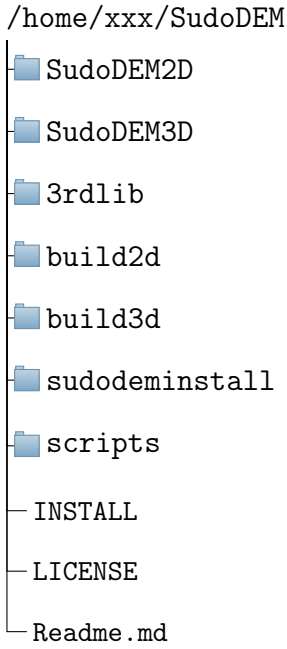
(1) Directory trees

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

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

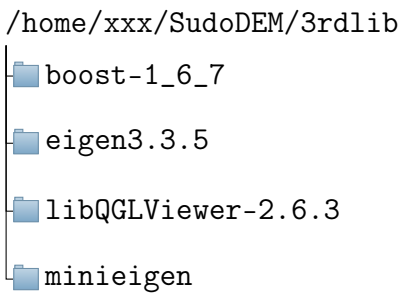
You may add subfolders, e.g.,'3rdlib', 'build2d', 'build3d', and 'sudodeminstall', inside the parent folder 'SudoDEM'. Thus, the directory tree

looks like



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



Four major libraries:

- Boost-1.67
  Download the source from its official page (https://www.boost.org/users/history/version_1_67_0.html) or the github repo (https://github.com/SwaySZ/boost-1_6_7).

```
cd boost-1\_6\_7
./bootstrap.sh --prefix=$PWD/../boost167 --with-libraries=python,thread,filesystem,iostreams,regex,serialization,system,date_time link=shared runtime-link=shared --without-icu
./b2 -j3
./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 'CMakeLists.txt':

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

  Then,

```
mkdir build
cd build
cmake ../
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)

```
cd QGLViewer
qmake
make
```

Tools and dependencies:

- biuld-essential, cmake

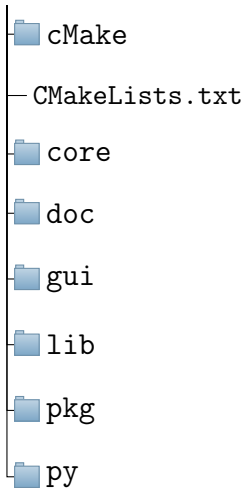
- freeglut3-dev

- 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
```



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

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

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

```
  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:

```
set(QGLVIEWER_INCLUDE_DIR "${CMAKE_CURRENT_SOURCE_DIR}/../3rdlib/
    libQGLViewer-2.6.3/")
set(QGLVIEWER_LIBRARIES "${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':

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

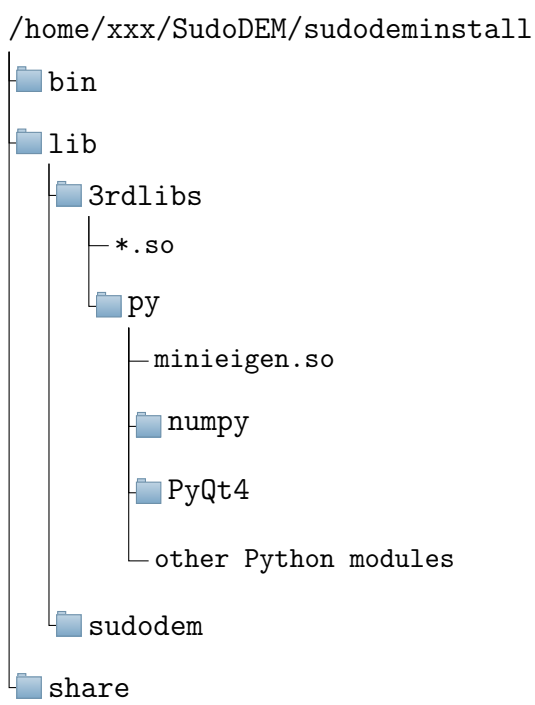
Compile and install *SudoDEM3D*:

```
cd build3d
cmake ../SudoDEM3D
make -j3
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':

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

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



Note: the installation procedure of *SudoDEM* does not write 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'.

```
cd 3rdlibs
cp /home/xxx/SudoDEM/scripts/changerpath.sh .
chmod +x changerpath.sh
./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':

```
# A sphere free falling on a ground under gravity

from sudodem import utils # module utils has some auxiliary functions

#1. we define materials for particles
mat = RolFrictMat(label="mat1",Kn=1e8,Ks=7e7,frictionAngle=math.atan(0.0),density=2650) # material 1 for particles
wallmat1 = RolFrictMat(label="wallmat1",Kn=1e8,Ks=0,frictionAngle=0.) # material 2 for walls

# we add the materials to the container for materials in the simulation.
# O can be regarded as an object for the whole simulation.
# We can see that materials is a property of O, so we use a dot operation (O.materials) to get the property (materials).
O.materials.append(mat) # materials is a list, i.e., a container for all materials involving in the simulations
```

```
O.materials.append(wallmat1)
# adding a material to the container (materials) so that we can easily access it by its label. See the definition of a wall below.

#create a spherical particle
sp = sphere((0,0,10.0),1.0, material = mat)
O.bodies.append(sp)
# create the ground as an infinite plane wall
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.
O.bodies.append(ground) # add the ground to the body container


# define a function invoked periodically by a PyRunner
def record_data():
    #open a file
    fout = open('data.dat','a') #create/open a file named 'data.dat' in append mode
    #we want to record the position of the sphere
    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.
    # p is the object of the sphere
    # as a body object, p has several properties, e.g., state, material, which can be listed by p.dict() in the commandline.
    pos = p.state.pos # get the position of the sphere
    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.
    # then, close the file and release resource
    fout.close()

# create a Newton engine by which the particles move under Newton
```

```
    's Law
newton=NewtonIntegrator(damping = 0.1,gravity=(0.,0.0,-9.8),label
    ="newton")
# we set a local damping of 0.1 and gravitational acceleration
    -9.8 m/s2 along z axis.

# the engine container is the kernel part of a simulation
# During each time step, each engine will run one by one for
    completing a DEM cycle.
O.engines=[
    ForceResetter(), # first, we need to reset the force container
     to store upcoming contact forces
    # next, we conduct the broad phase of contact detection by
    comparing axis-aligned bounding boxs (AABBs) to rule out
    those pairs that are definitely not contacting.
    InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Wall_Aabb()],
    verletDist=0.2*0.01),
    # then, we execute the narrow phase of contact detection
    InteractionLoop( # we will loop all potentially contacting
    pairs of particles
     [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.
     [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.
     [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.
    ),
    newton, # finally, we compute acceleration and velocity of
    each particle and update their positions.
    #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.
    PyRunner(command='record_data()',iterPeriod=10000,label='
    record',dead = False)
]

# we need to set a time step
O.dt = 1e-5

# clean the file data.dat and write a file head
fout = open('data.dat','w') # we open the file in a write mode so
```

```
    that the file will be clean
print>>fout, 'iter, x, y, z' # write a file head
fout.close() # close the file

# run the simulation
# you have two choices:
# 1. give a command here, like,
# O.run()
# you can set how many iterations to run
O.run(1600000)
# 2. clike the run button at the GUI controler
```

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

```
sudodem3d example0.py
```

For multi-threads, e.g., using 2 threads [1]

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

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

```
iter, x, y, z
10000 0.0 0.0 9.95588676912
20000 0.0 0.0 9.82357353912
30000 0.0 0.0 9.60306030912
40000 0.0 0.0 9.29434707912
50000 0.0 0.0 8.89743384912
60000 0.0 0.0 8.41232061912
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:

```
sudo apt install gnuplot
```

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

```
$> gnuplot
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

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

[1] Note: it is not acceptable to use multi-threads for the exemplified case.

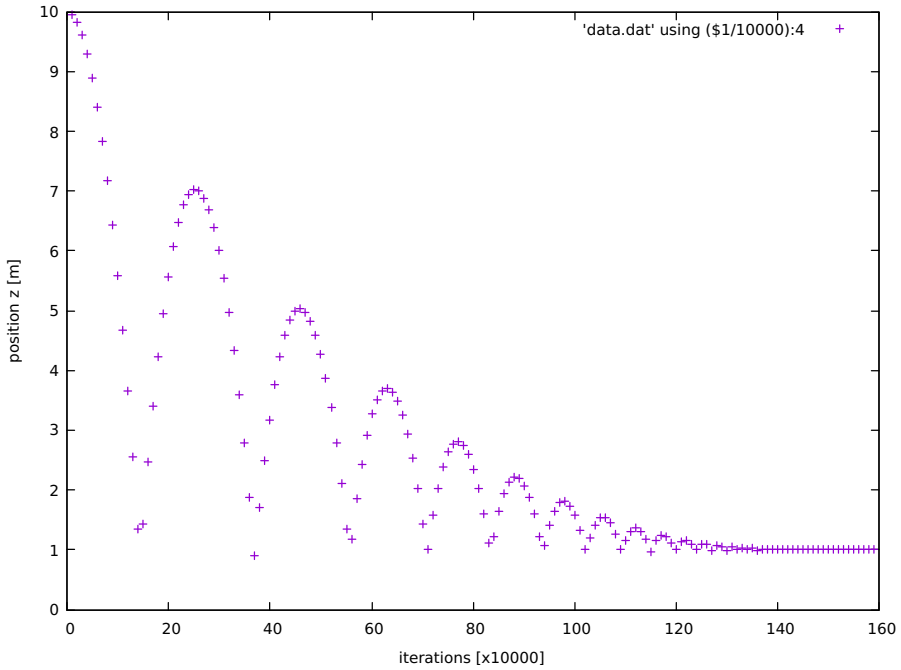


Figure 3.1: The z-position of a sphere free falling onto a ground.

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 axies, respectively; and $\epsilon_i (i = 1, 2)$ are the shape parameters determining the sharpness of particle edges or squareness of particle surface. Varying ϵ_i between 0 and 2 yields a wide range of convex-shaped superellipsoid. Two functions for generating a superellipsoid are highlighted below (see Sec. 5.1.2):

- NewSuperquadrics2(r_x, r_y, r_z, ϵ_1, ϵ_2, mat, rotate, isSphere)
- NewSuperquadrics_rot2(r_x, r_y, r_z, ϵ_1, ϵ_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.

```
############################################################
# granular packing of super-ellipsoids
# We position particles at a lattice grid,
# then particles free fall into a cubic box.
############################################################
# import some modules
from sudodem import _superquadrics_utils

from sudodem._superquadrics_utils import *
```

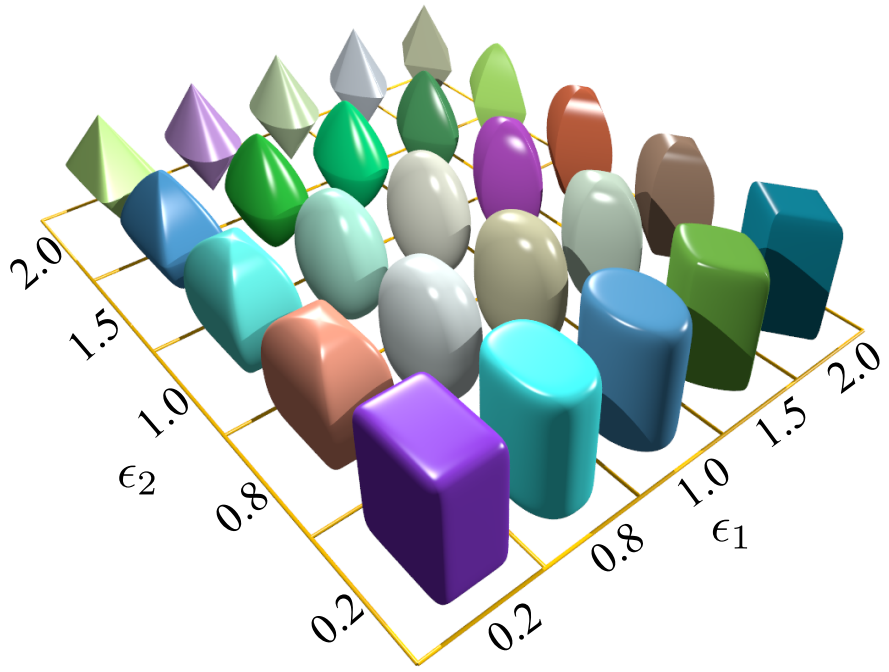


Figure 3.2: Superellipsoids.

```
import math
import random as rand
import numpy as np

########define some parameters###################
isSphere=False

num_x = 5
num_y = 5
num_z = 20
R = 0.1

#######define some auxiliary functions#########

#define a latice grid
#R: distance between two neighboring nodes
#num_x: number of nodes along x axis
#num_y: number of nodes along y axis
#num_z: number of nodes along z axis
#return a list of the postions of all nodes
def GridInitial(R,num_x=10,num_y=10,num_z=20):
    pos = list()
    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,
       isSphere)
51     body.state.pos=p
52     O.bodies.append(body)
53
54 ##########setup a simulation#########################
55 # material for particles
56 p_mat = SuperquadricsMat(label="mat1",Kn=1e5,Ks=7e4,frictionAngle
       =math.atan(0.3),density=2650,betan=0,betas=0)
57 # betan and betas are coefficients of viscous damping at contact,
        no viscous damping with 0 by default.
58 # material for the side walls
59 wall_mat = SuperquadricsMat(label="wallmat",Kn=1e6,Ks=7e5,
       frictionAngle=0.0,betan=0,betas=0)
60 # material for the bottom wall
61 wallmat_b = SuperquadricsMat(label="wallmat",Kn=1e6,Ks=7e5,
       frictionAngle=math.atan(1),betan=0,betas=0)
62 # add materials to O.materials
63 O.materials.append(p_mat)
64 O.materials.append(wall_mat)
65 O.materials.append(wallmat_b)
66
67 # create the box and add it to O.bodies
68 O.bodies.append(utils.wall(-R,axis=0,sense=1, material = wall_mat
       ))#left wall along x axis
69 O.bodies.append(utils.wall(2.0*R*num_x-R,axis=0,sense=-1,
       material = wall_mat))#right wall along x axis
70 O.bodies.append(utils.wall(-R,axis=1,sense=1, material = wall_mat
       ))#front wall along y axis
71 O.bodies.append(utils.wall(2.0*R*num_y-R,axis=1,sense=-1,
       material = wall_mat))#back wall along y axis
72 O.bodies.append(utils.wall(-R,axis=2,sense=1, material =wallmat_b
```

```
    ))#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(),
    Ig2_Superquadrics_Superquadrics_SuperquadricsGeom()],
87         [Ip2_SuperquadricsMat_SuperquadricsMat_SuperquadricsPhys()
    ], # collision "physics"
88         [SuperquadricsLaw()]    # contact law
89     ),
90     newton
91
92 ]
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 axies.
  - ϵ_1, ϵ_2: float, shape parameters.
  - isSphere: bool, particle is spherical or not.
- Methods:
  - getVolume(): float, return particle's volume.

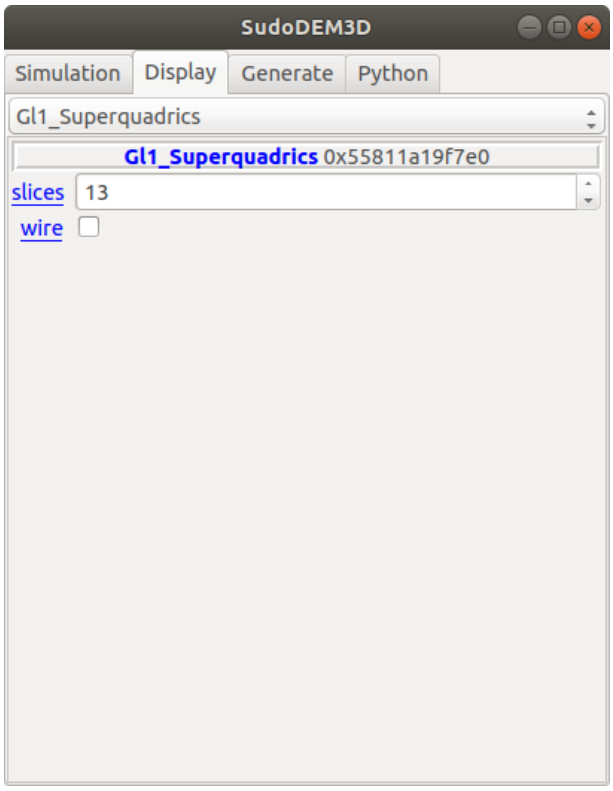


Figure 3.3: GUI for display configuration of superellipsoids.

- 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:

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

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

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

The output is like follows:

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

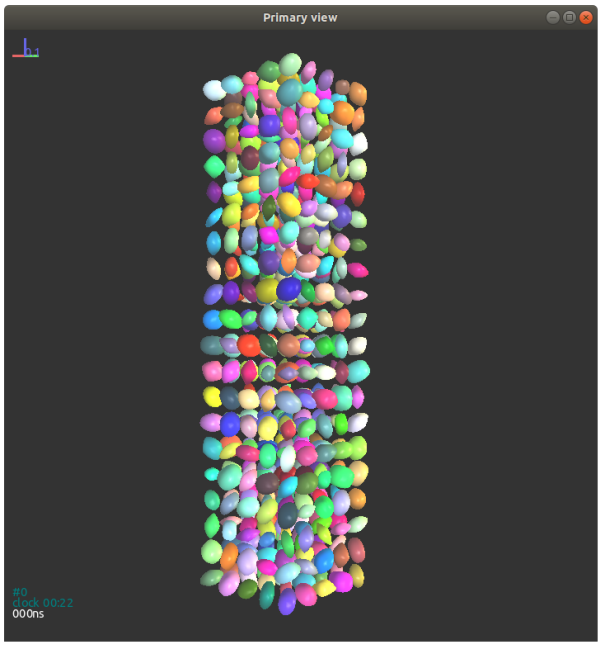


Figure 3.4: Initial packing of superellipsoids.

```
'shape': <Superquadrics instance at 0x56072ec83b60>,
'state': <State instance at 0x56072ec839e0>,
'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:

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

The output is as follows:

```
{'angMom': Vector3(0,0,0),
 'angVel': Vector3(0,0,0),
 'blockedDOFs': 0,
 'densityScaling': 1.0,
 'inertia': Vector3(0.0045389863901528615,0.007287422614611933,
                    0.0067053718092146865),
 'isDamped': True,
 'mass': 3.225715855242557,
 'refOri': Quaternion((1,0,0),0),
 'refPos': Vector3(0,0,0),
```

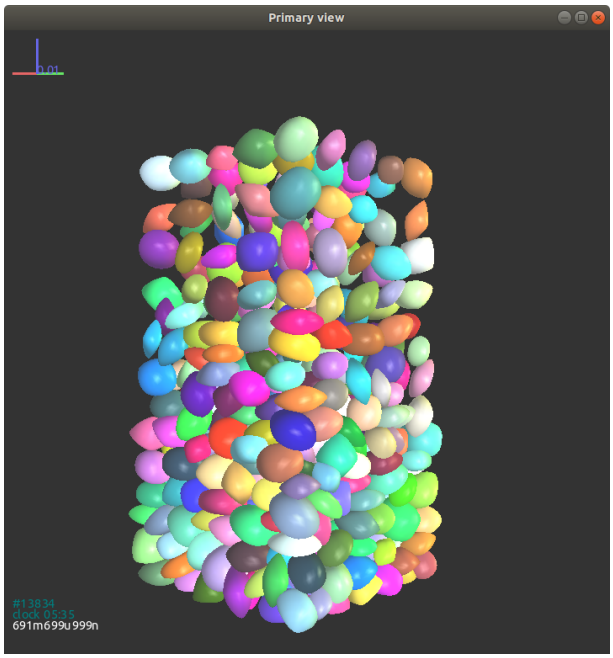


Figure 3.5: Packing of superellipsoids during falling.

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

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

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

The output is as follows:

```
{'color': Vector3(0.6407663308273844,0.07426042997942327,
                        0.05872324344642611),
 'eps': Vector2(1.3975848801318045,1.5376309088540607),
 'eps1': 1.3975848801318045,
 'eps2': 1.5376309088540607,
 'highlight': False,
 'isSphere': False,
 'rx': 0.1,
 'rxyz': Vector3(0.1,0.06469580193313924,0.07572423080016706),
 'ry': 0.06469580193313924,
 'rz': 0.07572423080016706,
 'wire': False}
```

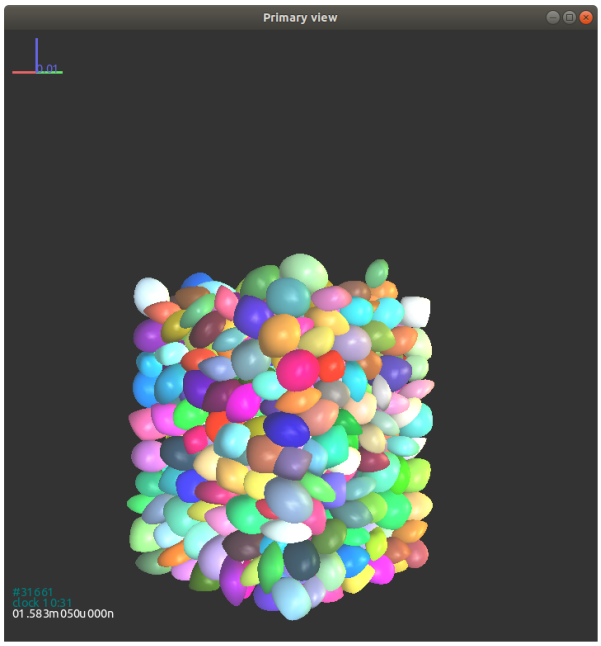


Figure 3.6: Final packing of superellipsoids.

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

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

```
triax=CompressionEngine(
    goalx = 1.0e5, # confining stress 100 kPa
    goaly = 1.0e5,
```

```
    goalz = 1.0e5,
  savedata_interval = 10000,
  echo_interval = 2000,
  continueFlag = False,
  max_vel = 100.0,
  gain_alpha = 0.5,
    f_threshold = 0.01, #1-f/goal
    fmin_threshold = 0.01, #the ratio of deviatoric stress
   to mean stress
    unbf_tol = 0.01, #unblanced force ratio
  file='record-con'
)
```

- *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* α: guiding the movement of the confining walls.

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

  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; Δt 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* α_f: a target ratio of the deviation of the present stress from the goal to the goal, i.e.,

$$\alpha_f = \frac{|f_w - f_g|}{f_g} \tag{3.3}$$

- *fmin_ threshold* β_f: a target ratio of the deviatoric stress q to the mean stress p, i.e.,

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

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

where σ_{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 δ_{ij} is the Kronecker delta.

- *unbf_tol* γ_f: a target unbalanced force ratio defined as

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

Note: the consolidation ends up with that all α_f, β_f and γ_f reach the targets.

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

- Compression

```
triax=CompressionEngine(
  z_servo = False,
    goalx = 1.0e5,
    goaly = 1.0e5,
    goalz = 0.01,
    ramp_interval = 10,
    ramp_chunks = 2,
  savedata_interval = 1000,
  echo_interval = 2000,
  max_vel = 10.0,
  gain_alpha = 0.5,
  file='sheardata',
  target_strain = 0.4,
  gain_alpha = 0.5
)
```

- *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}

$$I = \dot{\epsilon_1} \frac{\hat{d}}{\sqrt{\sigma_0/\rho}} \tag{3.7}$$

  where $\dot{\epsilon_1} = \mathrm{d}\epsilon_1/\mathrm{d}t$ is the major principal strain rate; $\hat{d}$ and ρ are the average diameter and material density of particles respectively; σ_0 is the confining stress.

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

```
##########################################################
from sudodem import plot, _superquadrics_utils
from sudodem._superquadrics_utils import *

import math
import random as rand
rand.seed(11245642)
##########################################################
epsilon = 1.4
isSphere = False
boxsize=[16.2e-3,16.2e-3,16.2e-3]
################################
#some auxiliary functions
#generate a sample of monodisperse superellipsoids with random orientaions and positions
def gen_sample(width, depth, height, r_max = 1.0e-3):
    b_num_tot = 5000
    for j in range(b_num_tot):
        rb = r_max*0.5
        x = rand.uniform(rb, width-rb)
        y = rand.uniform(rb, depth - rb)
        z = rand.uniform(rb, height - rb)
        r = r_max*0.5
        b = NewSuperquadrics2(r*1.25,r,r,epsilon,epsilon,mat,True,isSphere)
        b.state.pos=(x, y, z)
        O.bodies.append(b)
#Materials definition
```

```
mat = SuperquadricsMat(label="mat1",Kn=3e4,Ks=3e4,frictionAngle=
    math.atan(0.1),density=2650e4) #define Material with default
    values
wallmat1 = SuperquadricsMat(label="wallmat1",Kn=1e6,Ks=0.,
    frictionAngle=0.) #define Material with default values
wallmat2 = SuperquadricsMat(label="wallmat2",Kn=1e6,Ks=0.,
    frictionAngle=0.) #define Material with default values
O.materials.append(mat)
O.materials.append(wallmat1)
O.materials.append(wallmat2)
#################################################
#####create confining walls
#################################################
O.bodies.append(utils.wall(0,axis=0,sense=1, material = 'wallmat1
    '))#left x
O.bodies.append(utils.wall(boxsize[0],axis=0,sense=-1, material =
     'wallmat1'))#right x
O.bodies.append(utils.wall(0,axis=1,sense=1, material = 'wallmat1
    '))#front y
O.bodies.append(utils.wall(boxsize[1],axis=1,sense=-1, material =
     'wallmat1'))#back y

O.bodies.append(utils.wall(0,axis=2,sense=1, material = 'wallmat2
    '))#bottom z
O.bodies.append(utils.wall(boxsize[2],axis=2,sense=-1, material =
     'wallmat2'))#top z
#####create particles

gen_sample(boxsize[0], boxsize[1], boxsize[2])

####function for saving data (e.g., simulation states here)
def savedata():
        O.save(str(O.time)+'.xml.bz2')

#################################################
####triax engine
#CAUTION: the CompressionEngine regards the first six bodies as
    confining walls by default. So the walls should be generated
    first.
triax=CompressionEngine(
   goalx = 1.0e5, # confining stress 100 kPa
   goaly = 1.0e5,
   goalz = 1.0e5,
  savedata_interval = 10000,
  echo_interval = 2000,
  continueFlag = False,
  max_vel = 10.0,
   gain_alpha = 0.5,
   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(),
     Ig2_Superquadrics_Superquadrics_SuperquadricsGeom()],
79         [Ip2_SuperquadricsMat_SuperquadricsMat_SuperquadricsPhys()
     ],
80         [SuperquadricsLaw()]
81     ),
82     triax,
83     newton,
84     PyRunner(command='quiet_ball()',iterPeriod=2,iterLast = 50000,
     label='calm',dead = False),
85     PyRunner(command='savedata()',realPeriod=7200.0,label='
     savedata',dead = False)#saving data every two hours
86 ]
87
88
89 #delete the balls outside the box
90 def del_balls():
91         num_del = 0
92         #wall position
93         left = O.bodies[0].state.pos[0]#x
94         right = O.bodies[1].state.pos[0]
95         front = O.bodies[2].state.pos[1]#y
96         back = O.bodies[3].state.pos[1]
97         bottom = O.bodies[4].state.pos[2]#z
98         top = O.bodies[5].state.pos[2]
99         for b in O.bodies:
100                 if isinstance(b.shape,Superquadrics):
101                         (x,y,z) = b.state.pos
102                         if x > right or x < left or y < front or
     y > back or z < bottom or z > top:
103                                 O.bodies.erase(b.id)
104                                 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
```

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

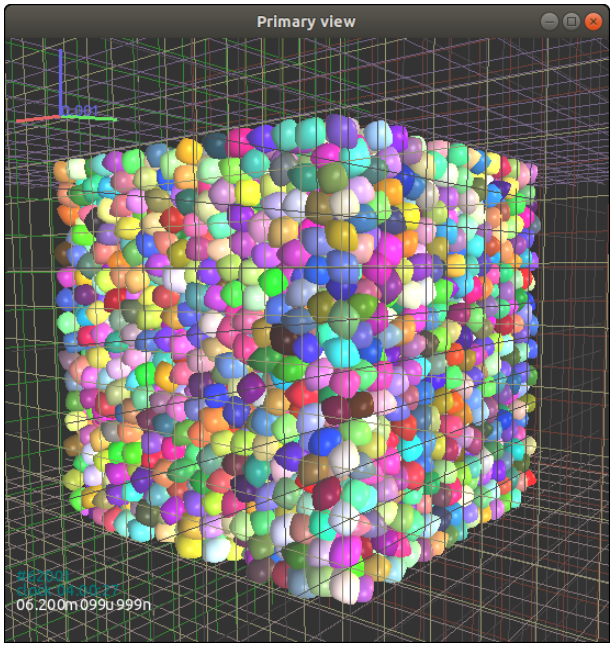


Figure 3.7: Packing of superellipsoids after consolidation.

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

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

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

```
#####################################################################
from sudodem import plot, _superquadrics_utils

from sudodem._superquadrics_utils import *

import math

saving_num = 0
#####################################################################
def savedata():
        global saving_num
        saving_num += 1
        #print "testing, no data saved"
```

```
        O.save('shear'+str(saving_num)+'.xml.bz2')

def quiet_system():
    for i in O.bodies:
        i.state.vel=(0.,0.,0.)
        i.state.angVel=(0.,0.,0.)
suffix=""
O.load("final_con"+suffix+".xml.bz2")
triax = O.engines[3]
O.dt = 5e-5
friction = 0.5
O.materials[0].frictionAngle=math.atan(friction)
if 1:
   for itr in O.interactions:
      id = min(itr.id1,itr.id2)
      if id > 5:
         itr.phys.tangensOfFrictionAngle = friction
         #itr.phys.betan = 0.5

##############################
#shear begins
##############################
def shear():
    quiet_system()
    triax.z_servo = False
    triax.ramp_interval = 10
    triax.ramp_chunks = 2
    triax.file="sheardata"+suffix
    triax.goalz = 0.01
    triax.goalx = 1e5
    triax.goaly = 1e5
    triax.savedata_interval = 2500
    triax.echo_interval = 2000
    triax.target_strain = 0.4

shear()

O.engines[-1].iterPeriod = 2500
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:

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

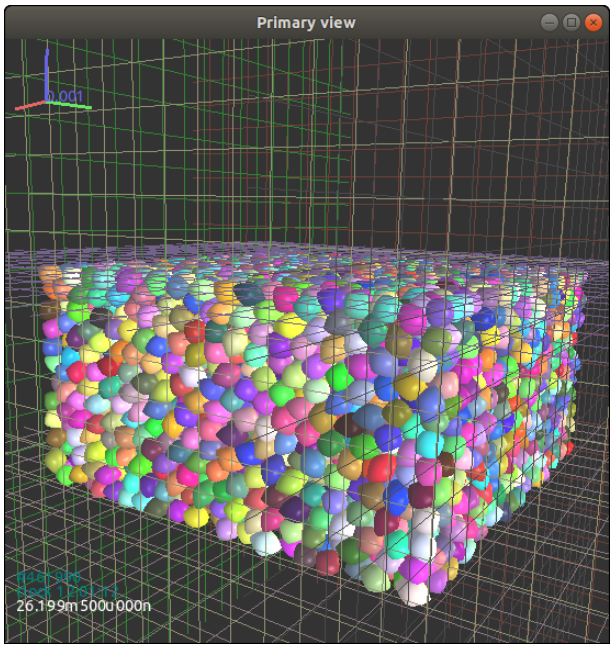


Figure 3.8: Packing of superellipsoids after shear.

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:

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

where σ'_{ij} is the deviatoric part of stress tensor σ_{ij}.

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

$$\epsilon_z = \int_{H_0}^{H} \frac{\mathrm{d}h}{h} = -\ln\frac{H_0}{H}, \quad \epsilon_v = \epsilon_x + \epsilon_y + \epsilon_z = \int_{V_0}^{V} \frac{\mathrm{d}v}{v} = -\ln\frac{V_0}{V} \tag{3.10}$$

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
```

```
0.005511002766  0.003867756447  148076.65  131980.95
0.008011004016  0.004825115794  159924.28  163706.90
0.010511005266  0.005409561745  168606.27  186520.14
0.013011006516  0.005673777079  175131.53  203243.28
0.015511007766  0.005650846281  180293.19  216248.17
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.

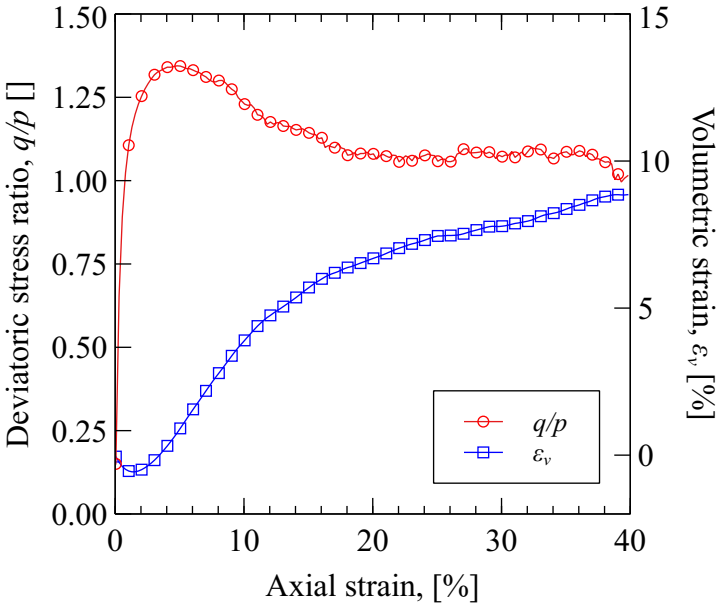


Figure 3.9: Variation of deviatoric stress ratio q/p and volumetric strain ϵ_v with axial strain during shearing.

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 [2], 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:

[2]GJKparticle is just experimental, and has not been rigorously tested yet.

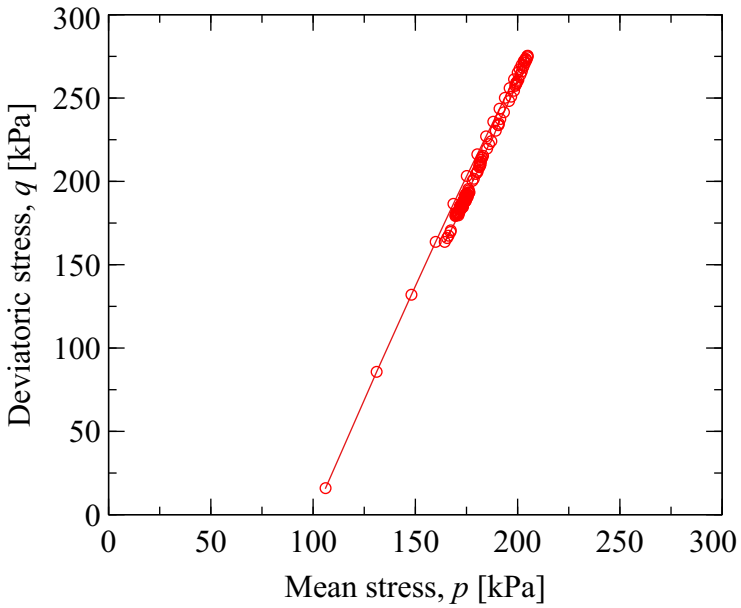


Figure 3.10: Loading path during shearing.

- 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:

```
from sudodem import plot, _gjkparticle_utils
from sudodem import *

from sudodem._gjkparticle_utils import *
import random as rand
import math
```

Next, define some constants:

```
cube_v0 =
    [[0,0,0],[1,0,0],[1,1,0],[0,1,0],[0,0,1],[1,0,1],[1,1,1],[0,1,1]]

cube_v = [[i*0.01 for i in j] for j in cube_v0]

isSphere=False

num_x = 5
num_y = 5
num_z = 20
R = 0.01
```

Define a lattice grid:

```
#R: distance between two neighboring nodes
#num_x: number of nodes along x axis
#num_y: number of nodes along y axis
#num_z: number of nodes along z axis
#return a list of the postions of all nodes
def GridInitial(R,num_x=10,num_y=10,num_z=20):
    pos = list()
    for i in range(num_x):
        for j in range(num_y):
            for k in range(num_z):
                x = i*R*2.0
                y = j*R*2.0
                z = k*R*2.0
                pos.append([x,y,z])
    return pos
```

Set properties of materials:

```
p_mat = GJKParticleMat(label="mat1",Kn=1e4,Ks=7e3,frictionAngle=
    math.atan(0.5),density=2650,betan=0,betas=0) #define Material
     with default values
#mat = GJKParticleMat(label="mat1",Kn=1e6,Ks=1e5,frictionAngle
    =0.5,density=2650) #define Material with default values
wall_mat = GJKParticleMat(label="wallmat",Kn=1e4,Ks=7e3,
    frictionAngle=0.0,betan=0,betas=0) #define Material with
    default values
wallmat_b = GJKParticleMat(label="wallmat",Kn=1e4,Ks=7e3,
    frictionAngle=math.atan(1),betan=0,betas=0) #define Material
    with default values

O.materials.append(p_mat)
O.materials.append(wall_mat)
O.materials.append(wallmat_b)
```

Create the cubic box:

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

```
O.bodies.append(utils.wall(-R,axis=0,sense=1, material = wall_mat))#left wall along x axis
O.bodies.append(utils.wall(2.0*R*num_x-R,axis=0,sense=-1, material = wall_mat))#right wall along x axis
O.bodies.append(utils.wall(-R,axis=1,sense=1, material = wall_mat))#front wall along y axis
O.bodies.append(utils.wall(2.0*R*num_y-R,axis=1,sense=-1, material = wall_mat))#back wall along y axis
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:

```
newton=NewtonIntegrator(damping = 0.3,gravity=(0.,0.,-9.8),label="newton",isSuperquadrics=4)

O.engines=[
    ForceResetter(),
    InsertionSortCollider([Bo1_GJKParticle_Aabb(),Bo1_Wall_Aabb()],verletDist=0.2*0.01),
    InteractionLoop(
        [Ig2_Wall_GJKParticle_GJKParticleGeom(),
    Ig2_GJKParticle_GJKParticle_GJKParticleGeom()],
        [Ip2_GJKParticleMat_GJKParticleMat_GJKParticlePhys()], # collision "physics"
        [GJKParticleLaw()]    # contact law -- apply forces
    ),
    newton
]
O.dt=1e-5
```

Polyhedral particles are generated as follows:

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

# create a lattice grid
pos = GridInitial(R,num_x=num_x,num_y=num_y,num_z=num_z) # get positions of all nodes
# create particles at each nodes
GenSample(R,pos) #
```

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

```
#generate a sample
```

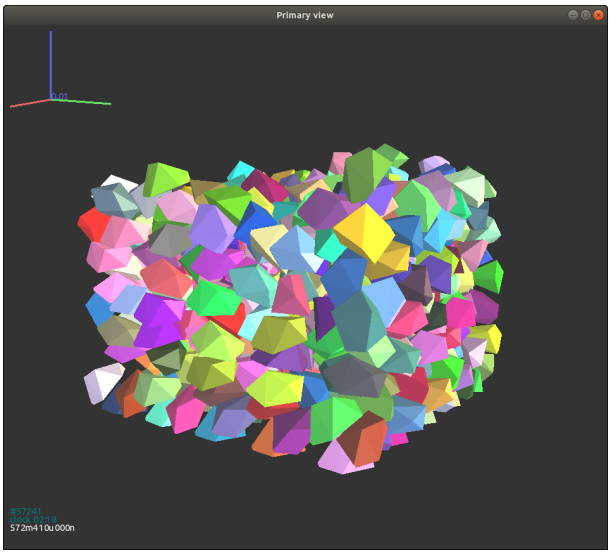


Figure 3.11: Final packing of polyhedrons without random initial orientations (the box not shown).

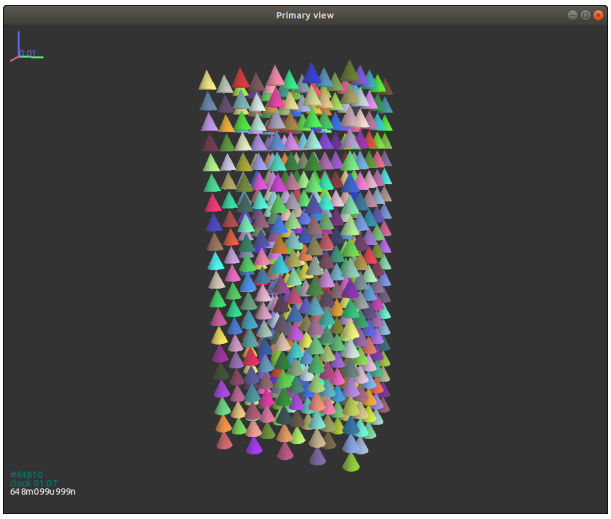


Figure 3.12: Final packing of cones without random initial orientations (the box not shown).

```
def GenSample(r,pos):
    for p in pos:
     body = GJKCone(0.005,0.01,0.05*1e-2,p_mat, False)
     body.state.pos=p
     O.bodies.append(body)
     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:

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

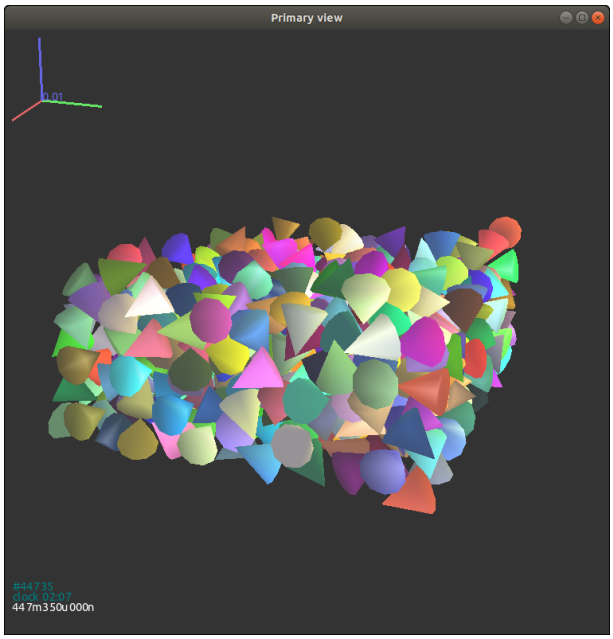


Figure 3.13: Final packing of cones with random initial orientations (the box not shown).

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:

```
def GenSample(r,pos):
    for p in pos:
     body = GJKCylinder(0.005,0.01,0.05*1e-2,p_mat,False)
     body.state.pos=p
     O.bodies.append(body)
     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

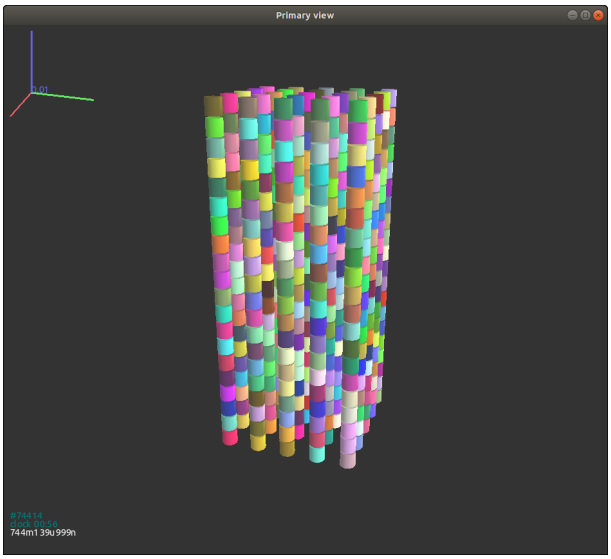


Figure 3.14: Final packing of cylinders without random initial orientations (the box not shown).

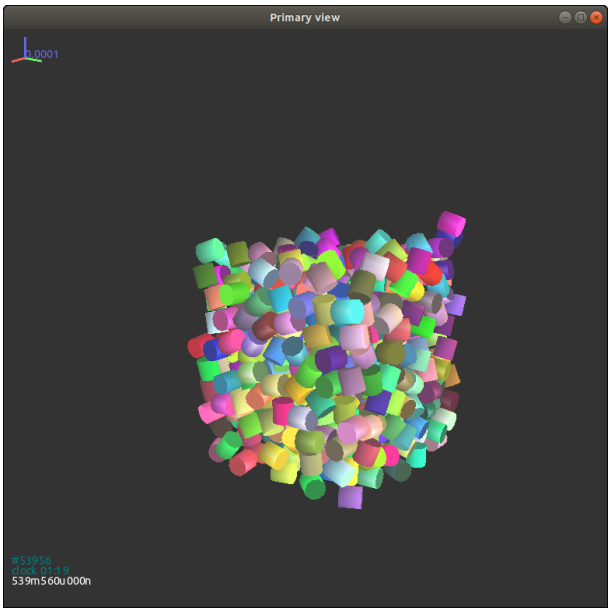


Figure 3.15: Final packing of cylinders with random initial orientations (the box not shown).

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

```
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.

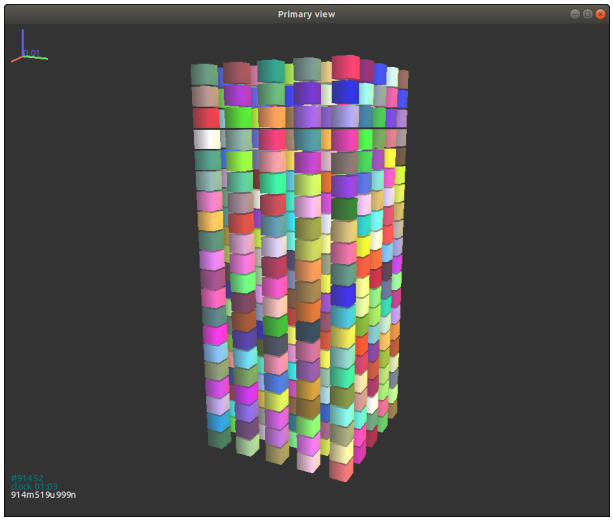


Figure 3.16: Final packing of cuboids without random initial orientations (the box not shown).

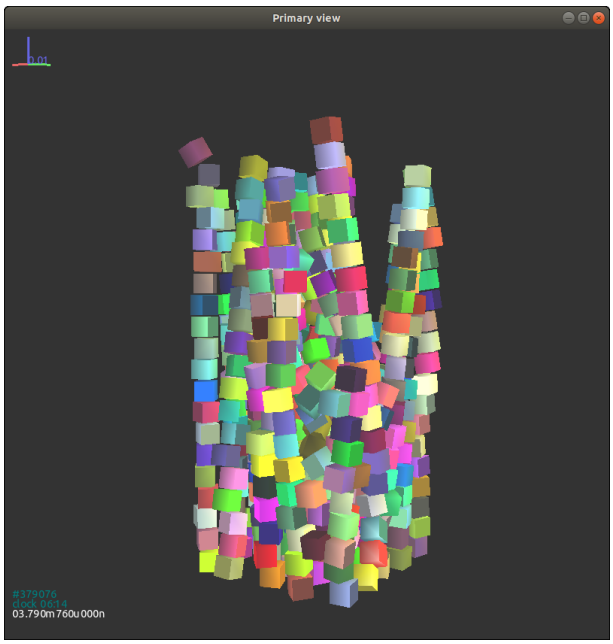


Figure 3.17: Final packing of cuboids with random initial orientations (the box not shown).

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

```
#generate a sample
def GenSample(r,pos):
    for p in pos:
     body = GJKCuboid([0.005,0.005,0.005],0.05*1e-2,p_mat,False)
     body.state.pos=p
     O.bodies.append(body)
     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:

```
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:

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

with

$$r_x = r_x^+ \quad \text{if} \quad x \geq 0 \quad \text{else} \quad r_x^- \tag{3.12a}$$
$$r_y = r_y^+ \quad \text{if} \quad y \geq 0 \quad \text{else} \quad r_y^- \tag{3.12b}$$
$$r_z = r_z^+ \quad \text{if} \quad z \geq 0 \quad \text{else} \quad r_z^- \tag{3.12c}$$

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; ϵ_1, ϵ_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 ϵ_1 and ϵ_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)

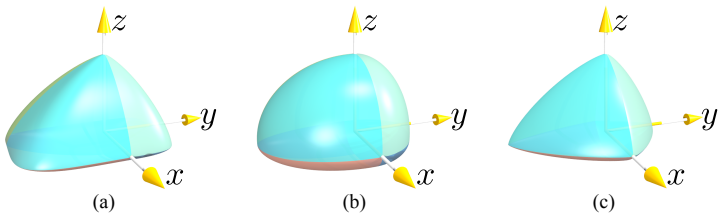


Figure 3.18: Poly-superellipsoids with $r_x^+ = 1.0, r_x^- = 0.5, r_y^+ = 0.8, r_y^- = 0.9, r_z^+ = 0.4, r_z^- = 0.6$ and (a) $\epsilon_1 = 0.4, \epsilon_2 = 1.5$, (b) $\epsilon_1 = \epsilon_2 = 1.0$, (c) $\epsilon_1 = \epsilon_2 = 1.5$.

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

```
#
from sudodem._superquadrics_utils import *
import math
import random
import numpy as np

a=1.1734e-2
isSphere=False
ap=0.4
eps=0.5
num_s=200
num_t=8000
trials=0


wallboxsize=[10e-1,0,0] #size of the layer for the particles
    generation
boxsize=np.array([7e-1,7e-1,20e-1])

def down():
    for b in O.bodies:
        if(isinstance(b.shape,PolySuperellipsoid)):
            if(len(b.intrs())==0):
                b.state.vel=[0,0,-2]
                b.state.angVel=[0,0,0]

```

```
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
    range(0,3)])
55             if(distance<(4.*gap*gap)):
56 
57                 isOK=False
58                 break
59 
60         if(isOK==True):
61             coor.append(pos)
62             num+=1
63             trials+=1
64             #print(num)
65     return coor
66 
67 
68 def Addlayer(a,eps,ap,mat,boxsize,num_s,num_t):
69    bottom=gettop()
70    coor=Genparticles(a,eps,ap,mat,boxsize,num_s,num_t,bottom)
71    for b in coor:
72       #xyz = np.array([1.0,0.5,0.8,1.5,1.2,0.4])*0.1
73       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,
        frictionAngle=math.atan(0.3),density=2650,betan=0,betas=0) #
        define Material with default values
83  wall_mat = PolySuperellipsoidMat(label="wallmat",Kn=1e6,Ks=7e5,
        frictionAngle=0.0,betan=0,betas=0) #define Material with
        default values
84  wallmat_b = PolySuperellipsoidMat(label="wallmat",Kn=1e6,Ks=7e5,
        frictionAngle=math.atan(1),betan=0,betas=0) #define Material
        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,
        material = wall_mat))#left x
92  O.bodies.append(utils.wall(0.5*wallboxsize[0],axis=0,sense=-1,
        material = wall_mat))#right x
93  O.bodies.append(utils.wall(-0.5*boxsize[1],axis=1,sense=1,
        material = wall_mat))#front y
94  O.bodies.append(utils.wall(0.5*boxsize[1],axis=1,sense=-1,
        material = wall_mat))#back y
95
96  O.bodies.append(utils.wall(0,axis=2,sense=1, material =wallmat_b)
        )#bottom z
97  #O.bodies.append(utils.wall(boxsize[0],axis=2,sense=-1, material
        = wallmat_b))#top z
98
99  newton=NewtonIntegrator(damping = 0.3,gravity=(0.,0.,-9.8),label=
        "newton",isSuperquadrics=2)#isSuperquadrics=2 for Poly-
        superellipsoids
100
101 O.engines=[
102     ForceResetter(),
103     InsertionSortCollider([Bo1_PolySuperellipsoid_Aabb(),
        Bo1_Wall_Aabb()],verletDist=0.2*0.1),
104     InteractionLoop(
105         [Ig2_Wall_PolySuperellipsoid_PolySuperellipsoidGeom(),
        Ig2_PolySuperellipsoid_PolySuperellipsoid_PolySuperellipsoidGeom
        ()],
```

```
        [
    Ip2_PolySuperellipsoidMat_PolySuperellipsoidMat_PolySuperellipsoidPhys
    ()], # collision "physics"
        [PolySuperellipsoidLaw()]   # contact law -- apply forces
    ),
    newton
]
O.dt=5e-5
#adding particles
Addlayer(a,eps,ap,p_mat,boxsize,num_s,num_t)#Note: these
    particles may intersect with each other when we generate them
    .

#setting the Display properties. You can go to the Display panel
    to set them directly.
sudodem.qt.Gl1_PolySuperellipsoid.wire=False
sudodem.qt.Gl1_PolySuperellipsoid.slices=10
sudodem.qt.Gl1_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

$$|\frac{x}{r_x}|^{\frac{2}{\epsilon}} + |\frac{y}{r_y}|^{\frac{2}{\epsilon}} = 1 \tag{3.13}$$

where r_x and r_y are referred to as the semi-major axis lengths in the direction of x, and y axies, 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:

```
############################################################
from sudodem import utils
from sudodem._superellipse_utils import *
import numpy as np
import math
import random
random.seed(11245642)
############################################################
##########constants
############################################################
isSphere = False

r = 0.5 #particle size
```

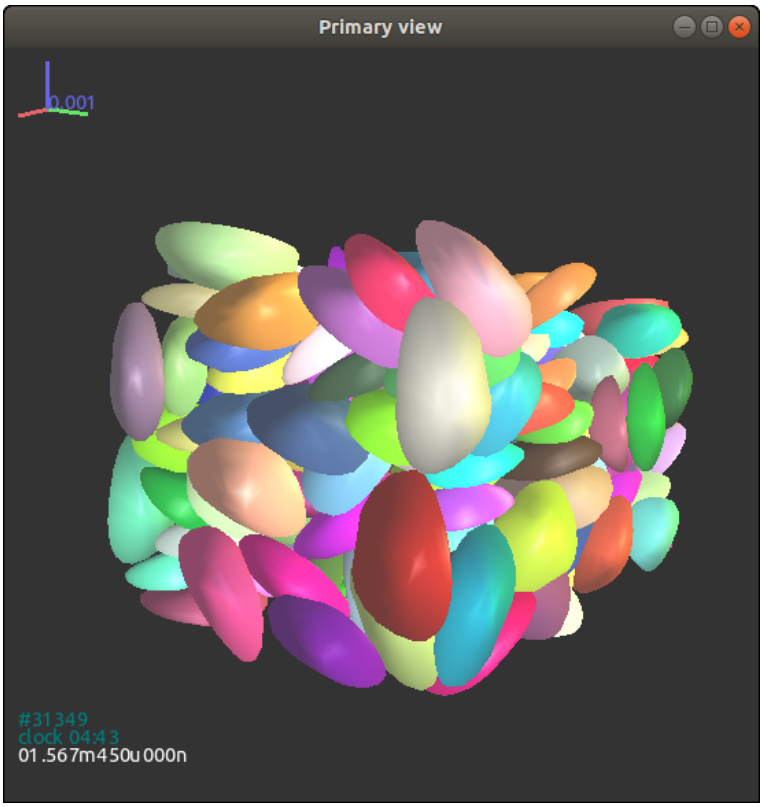


Figure 3.19: Packing of poly-superellipsoids.

```
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
    range(0,2)])
55              if(distance<(4.*gap*gap)):
56
57                  isOK=False
58                  break
59
60          if(isOK==True):
61              coor.append(pos)
62              num+=1
63              trials+=1
64              #print(num)
65      return coor
66
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()
       ],verletDist=0.1),
87     InteractionLoop(
88         [Ig2_Wall_Superellipse_SuperellipseGeom(),
       Ig2_Superellipse_Superellipse_SuperellipseGeom()],
89         [Ip2_SuperellipseMat_SuperellipseMat_SuperellipsePhys()], #
        collision "physics"
90         [SuperellipseLaw()]    # contact law -- apply forces
91     ),
92     newton,
93     PyRunner(command='Addlayer(r,mat,boxsize,num_s,num_t)',
       virtPeriod=0.1,label='check',dead = False)
94 ]
```

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

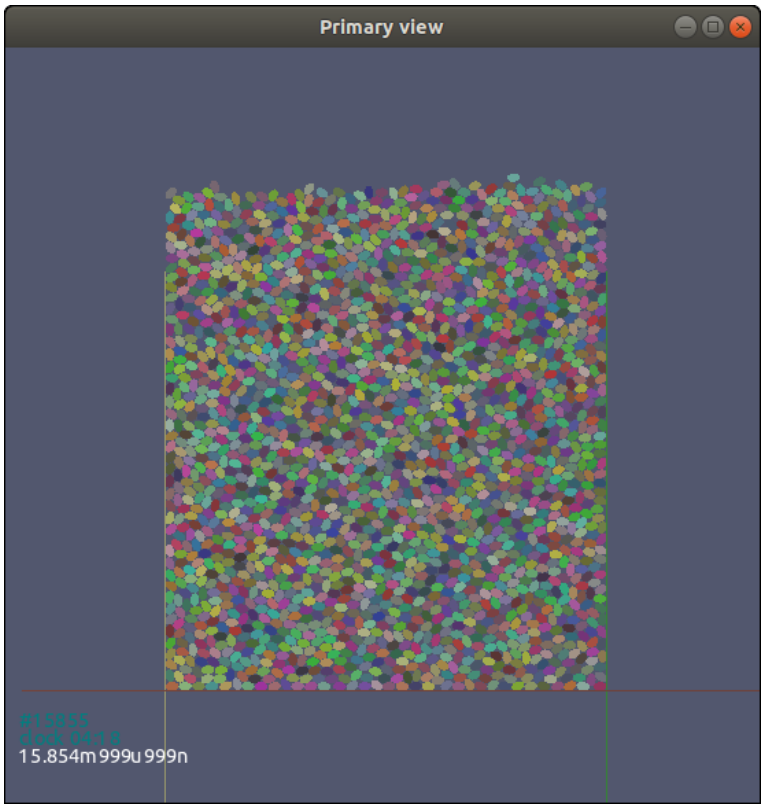


Figure 3.20: Packing of superellipses under gravity.

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:

```
def savedata():
        global saving_num
        saving_num += 1
        #print "testing, no data saved"
        O.save('shear'+str(saving_num)+'.xml.bz2')

O.engines = O.engines + [PyRunner( command = 'savedata()',
    iterPeriod = 1000 )] # if we load the consolidation data,
    then we do not need this line because the PyRunner calling '
    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.,

```
import sudodem.utilspost as utilspost
#the following functions will output the average friction
    mobilization index of each state to a single file.
for i in ['1.0','1.2','1.4','1.6']:
  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, ϵ_1, ϵ_2, position

x, y, z, and orientation $q0$,$q1$,$q2$,$q3$ (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, ϵ_1, ϵ_2, position x, y, z, and orientation $q0$,$q1$,$q2$,$q3$ (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*[1] and *POV-Ray*[2], 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.

```
O.load("shearend.xml.bz2")
import sudodem._superquadrics_utils as superutils
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

[1] https://www.paraview.org/
[2] www.povray.org/

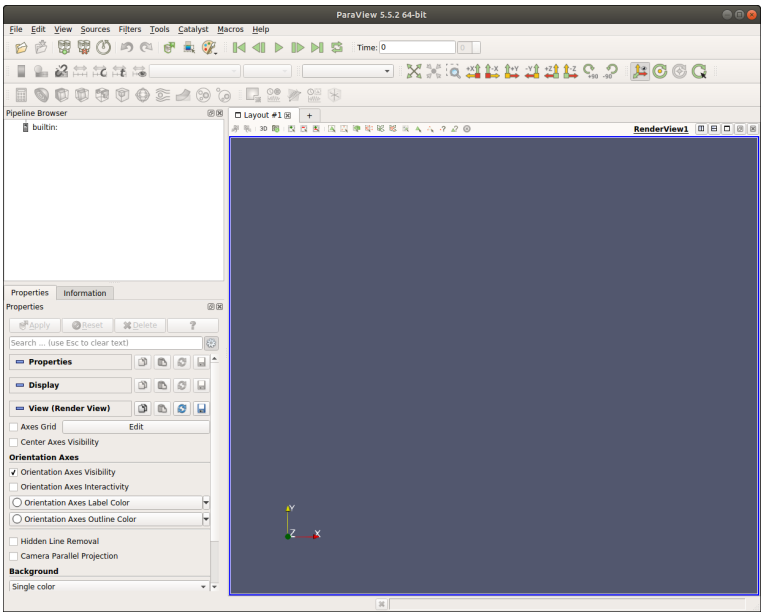


Figure 4.1: The main window of Paraview.

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*.

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

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

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

Light sources should be slightly adjusted as well.

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

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

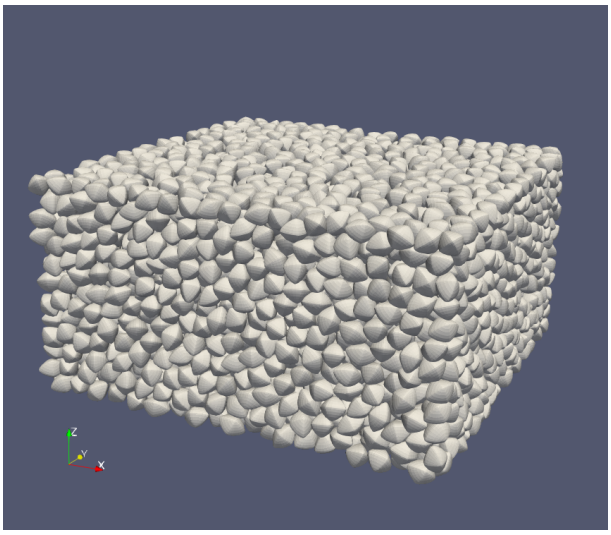


Figure 4.2: Configuration of particles rendered by *Paraview*.

```
povray shearend.pov
```

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

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

Image processing softwares, e.g., *GIMP* [3], 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:

```
from sudodem import snapshot
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:

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

[3] https://www.gimp.org/

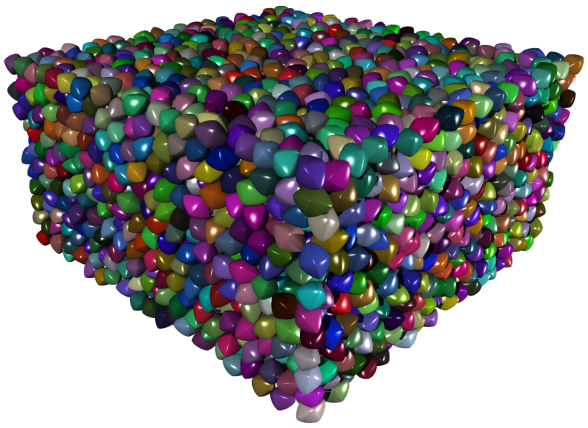


Figure 4.3: Configuration of particles rendered by *POV-Ray*.

```
#end
#macro randomColor(r,g,b)
texture{ pigment{ color rgb<r,g,b> transmit para_trans} finish { phong para_phong }}
#end
#macro octantSuper(ep1,ep2,rx,ry,rz,sign1,sign2,sign3)
    superellipsoid{ <ep1,ep2>
#if (singleColor = false)
 randomColor(rand(Random_r),rand(Random_g),rand(Random_b))
#end
myremain(sign1,sign2,sign3) scale <rx,ry,rz>}
#end
#macro polySuperEllipsoid(ep1,ep2,a1,a2,b1,b2,c1,c2,t1,t2,t3,r1,r2,r3,r,g,b)
union{
octantSuper(ep1,ep2,a2,b2,c2,-1,-1,-1)
octantSuper(ep1,ep2,a1,b2,c2,1,-1,-1)
octantSuper(ep1,ep2,a2,b1,c2,-1,1,-1)
octantSuper(ep1,ep2,a1,b1,c2,1,1,-1)
octantSuper(ep1,ep2,a2,b2,c1,-1,-1,1)
octantSuper(ep1,ep2,a1,b2,c1,1,-1,1)
octantSuper(ep1,ep2,a2,b1,c1,-1,1,1)
octantSuper(ep1,ep2,a1,b1,c1,1,1,1)
rotate<r1,r2,r3>
translate<t1,t2,t3>
#if (singleColor = true)
randomColor(r,g,b)
#end
}
#end
polySuperEllipsoid(1.0000000e+00,1.0000000e+00,1.0000000e-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)
...
...
```

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:

```
//using the command: povray **.pov +A0.01 +W1600 +H1200
#include "colors.inc"
#declare singleColor = true;
#declare Random_r = seed (1432);
#declare Random_g = seed (7242);
#declare Random_b = seed (9912);
#declare para_trans = 0 ;
#declare para_phong = 1.0 ;
camera {
location < 1.39244459003 , 0.950562440499 , 0.605226784362 >
sky z
right -x*image_width/image_height
look_at < 0.612872382451 , 0.349255160637 , 0.430021966053 >
}
#include "packingellipses.inc"
light_source{<4,8,5>*10 color rgb <1,1,1>}
light_source{<12,-6>*10 color rgb <1,1,1>}
light_source { <0, 2, 10> White }
background{rgb 1}
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

```
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.

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

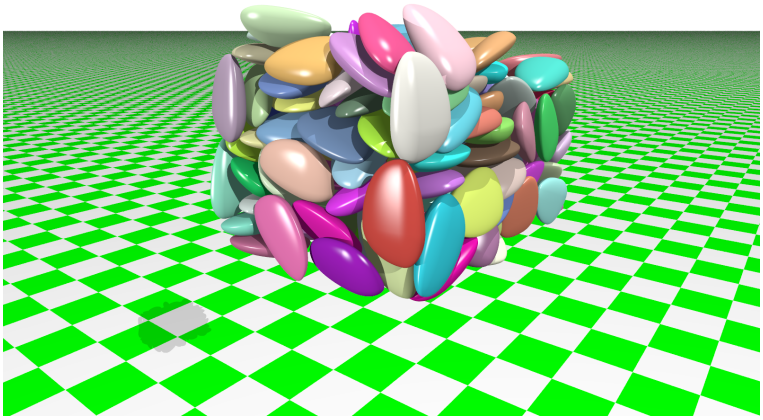


Figure 4.4: Packing of poly-superellipsoids in Example 4 rendered by *POV-Ray*.

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.

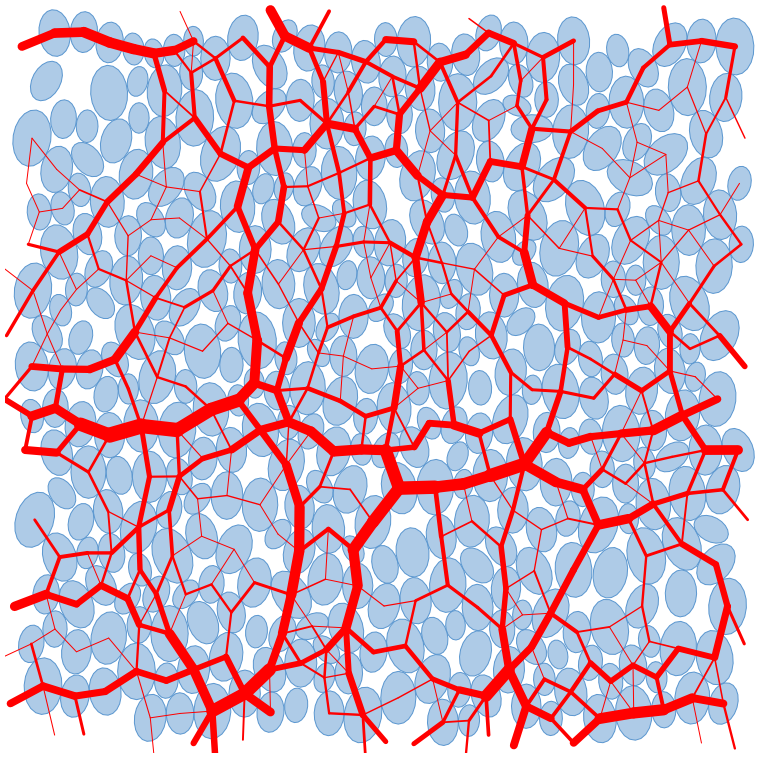


Figure 4.5: Particles and contact force chains dumped by *drawSVG*.

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
  Degress of freedom where linear/angular velocity will be always constant (equal to zero, or to an user-defined value), regardless of applied force/torque. String that may contain 'xyzXYZ' (translations and rotations).
- 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, ϵ_1, ϵ_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 axies.
  - ϵ_1, ϵ_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, ϵ_1, ϵ_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 axies.
  - ϵ_1, ϵ_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, q_w, q_x, q_y, q_z, 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 axies, 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 Superellpisoids 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.
- GJKCylinder(*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_min, x_max], y=[y_min, y_max], z=[z_min, z_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
  Degress of freedom where linear/angular velocity will be always constant (equal to zero, or to an user-defined value), regardless of applied force/torque. String that may contain '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 aperioidc 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: $S_{ijkl} = \frac{1}{V}\sum_c (k_n n_i l_j n_k l_l + k_t t_i l_j t_k l_l)$ float volume: same as in function getStress bool symmetry: make the tensors symmetric.
  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 formular 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.