Discrete Particle Modeling of granular materials using MercuryDPM

Discrete Particle Modeling of granular materials using MercuryDPM

Part 1

Dec 9, 2013

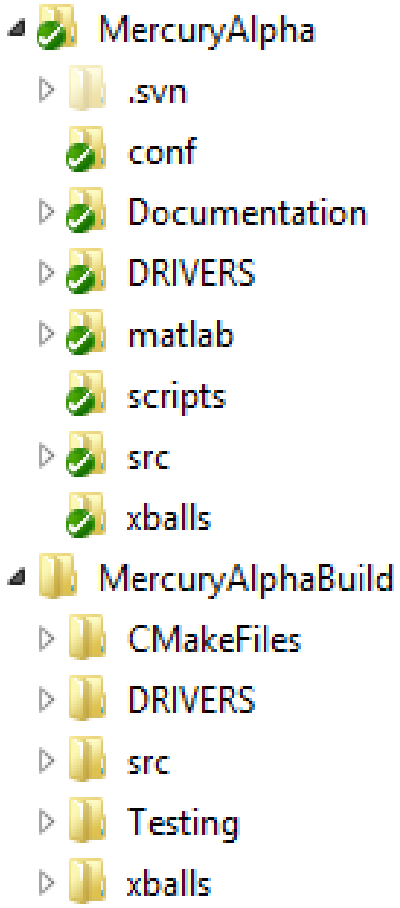
Installation instructions

- see Installation Instructions for MercuryDPM.pdf on Blackboard (will be on the website soon)

Website

- see website http://mercurydpm.org/
- you also find the documentation there (or use `make doc`)

Directory structure



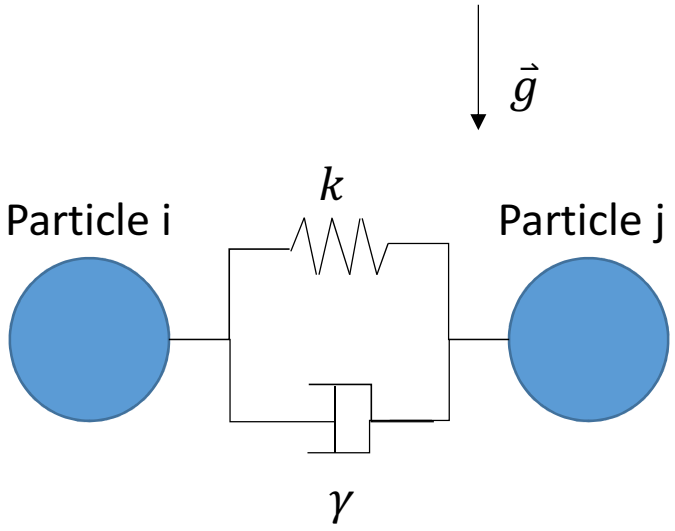
Source directory (e.g., MercuryAlpha)

- src: do not change, contains the MercuryDPM source code (in particular class MD)
- DRIVERS: implement your simulations here

Build directory (e.g., MercuryAlphaBuild)

- go there to compile and run code (type `make help` in the build dir. to find out more)

Basic contact dynamics [Luding 2008]



Example:

- glass, steel: r>0.9
- table tennis ball: 0.8<r<0.9
- rubber: r<0.2

- Particles are spherical with mass m_i, diameter d_i, position $\vec{r}_i$, velocity $\vec{v}_i$
- Contacts are elastic-dissipative with stiffness k, dissipation γ, overlap δ_{ij}^n, rel. normal vel. v_{ij}^n

$$f_{ij} = -k^n \delta_{ij}^n - \gamma^n v_{ij}^n$$

- stiffness is always underestimated to improve speed; choose k s.t. overlaps are small even for max. pressure/velocity and check for k –indep.
- this yields contact time $t_c = \frac{\pi}{\omega}$ and restitution coefficient $r = \frac{v_{after}}{v_{before}} = \exp\left(-\frac{\pi\eta_0}{\omega}\right)$ with $\omega = \sqrt{\frac{k}{m_{ij}} - \eta_0^2}, \eta_0 = \frac{\gamma_0}{2m_{ij}}, m_{ij} = \frac{m_i m_j}{m_i + m\_j}$
  - body forces are active on all particles
- walls are particles of inf. mass and radius

The MD class and the solve routine

The solve routine is the work horse of the code.

- runs setup_particles_initial_conditions();
- runs actions_before_time_loop();
- runs time loop: while (t<dt<tmax)
  - runs do_integration_before_force_computation(*it);
  - runs output_xballs_data();
  - runs actions_before_time_step();
  - runs compute_all_forces();
  - runs actions_after_time_step()
  - runs do_integration_after_force_computation(*it);
  - runs output_ene();
  - increments in time t+=dt;

Blue functions can be modified by the user.

Run your first code

- Look at the source code
  gedit
  ~/MercuryAlpha/DRIVERS/UnitTests/free_fall_demo.cpp
- Goto your build directory, update build if necessary, and compile
  cd ~/MercuryAlphaBuild
  cmake .
  make free_fall_demo
- Run your code
  cd DRIVERS/UnitTests
  ./free_fall_demo
- View the output (editor, gnuplot, matlab)

Output file format *.data

Format of *.data: This file is used for plotting particles. For each time step, the following format is used:

First Line: N, time, xmin, ymin, zmin, xmax, ymax, zmax

N Lines: x, y, z, vx, vy, vz, rad, q1, q2, q3, omex, omey, omez, xi

where N is the number of particles,

time denotes the time step,

xmin, ymin, zmin, xmax, ymax, zmax denote the domain size,

x, y, z are the coordinates,

vx, vy, vz are the volocities,

rad is the radius,

q1, q2, q3 is the angular position,

omex, omey, omez is the angular velocity,

and xi is an additional variable the user can specify (default 0)

This is the (standard) output required for 3D data; for 2D data, only seven columns of particle information is written: x, y, z, vx, vy, vz, rad, xi

Output file format *.data

Format of *.data: This file is used for plotting particles. For each time step, the following format is used:

First Line: N, time, xmin, ymin, zmin, xmax, ymax, zmax

N Lines: x, y, z, vx, vy, vz, rad, q1, q2, q3, omex, omey, omez, xi

where N is the number of particles,

Example: free_fall.data

```
1 0 0 0 0.01 0.1
0.005 0.049998775 0 -0.00245 0.0005 -0 -0 0
1 0.25 0 0 0.01 0.1
0.005 0.07253368046208 0 -0.242797898369 0.0005 -0 -0 0
...
```

Output file format *.fstat

Format of *.fstat: This file is mainly used for calculating stresses. For each time step, the following format is used:

 # time, info
 # info
 # info
1 line per contact: time, i, j, x, y, z, delta, deltat, fn, ft, nx, ny, nz, tx, ty, tz

with time step time,
particle number i
contact partner j (particles >= 0, walls < 0)
the contact point x, y, z
delta = overlap at the contact
deltat = length of the tangential spring
absolute normal force |f^n|
absolute tangential force |f^t|=|f-f^n|
normal unit vector nx, ny, nz
tangential unit vector tx, ty, tz.

Output file format *.fstat

Format of *.fstat: This file is mainly used for calculating stresses. For each time step, the following format is used:

 # time, info

 # info

 # info

1 line per contact: time, i, j, x, y, z, delta, deltat, fn, ft, nx, ny, nz, tx, ty, tz

Example: free_fall.fstat

```
# 0.1010000000001 0
# 0 0 0 0.01 0.1 0
# 0.0005 0.0005 0 0 0 0
0.1010000000001 0 -3 0.005 -2.710505431214e-20 0 0.0003704634180495 0 3.704634180495 0 0 -1 0 -0 -0 -0
# 0.1015000000001 0
# 0 0 0 0.01 0.1 0
# 0.0005 0.0005 0 0 0 0
0.1015000000001 0 -3 0.005 0 0 0.0002354139557222 0 2.354139557222 0 0 -1 0 -0 -0 -0
...
```

Output file format *.ene

Format of *.ene: This file is mainly used for interpreting the time evolution. For each time step, the following format is used:

time ene_gra ene_kin ene_rot ene_ela X_COM Y_COM Z_COM

with

ene_gra = sum_i m_i Dot([x,y,z],-[gx,gy,gz])
the gravitational potential energy (with gravity g=[gx,gy,gz])

ene_kin = sum_i m_i v_i^2 / 2
the translational kinetic energy

ene_rot = sum_i I_i ome_i^2 / 2
the rotational kinetic energy (with inertia I)

ene_ela = sum_i (k_i delta_i^2 + kt_i deltat_i^2) / 2
the potential energy from contact forces

X_COM, Y_COM, Z_COM the center of mass

Output file format *.ene

Format of *.ene: This file is mainly used for interpreting the time evolution. For each time step, the following format is used:

time ene_gra ene_kin ene_rot ene_ela X_COM Y_COM Z_COM

with

Example: free_fall.ene

t	ene_gra	ene_kin	ene_rot	ene_ela	X_COM	Y_COM	Z_COM
0	0.00076967	1.885740-08	0	0	0.005	0.04999	0
0.5	0.000771285	0.000128745	0	0	0.005	0.05010	0
...							

Output file format *.restart

Format of *.restart: This file contains all information required to restart the code from a given timestep:

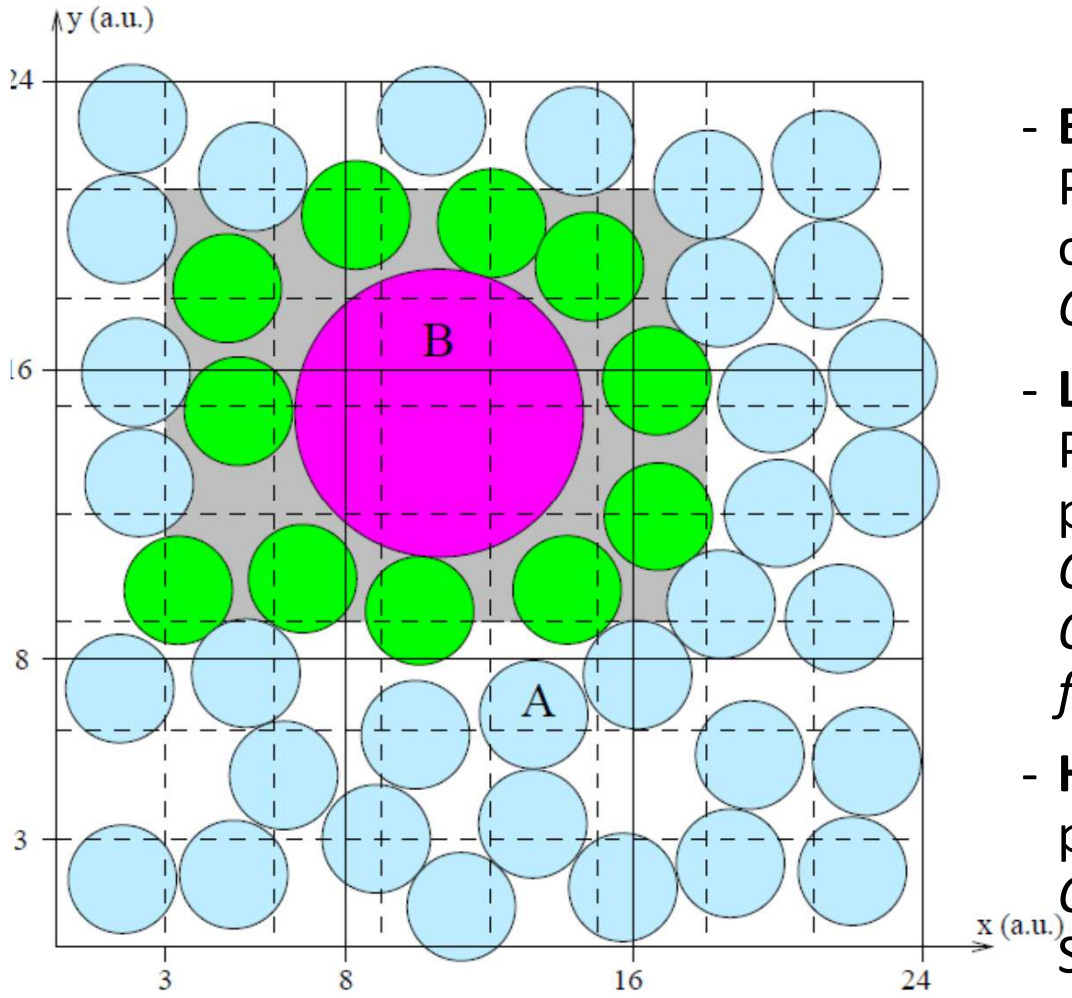
Example: free_fall.restart

```
restart_version 3
name free_fall
xmin 0 xmax 0.01 ymin 0 ymax 0.1 zmin 0 zmax 0
dt 1e-06 t 1 tmax 1 save_count_data 500 save_count_ene 500 save_count_stat 500
save_count_fstat 500
dim 2 gravity 0 -9.8 0
options_fstat 1 options_data 1 options_ene 1 options_restart 1
Species 1
k 10000 disp 0 kt 0 dispt 0 mu 0 rho 2000 dim_particle 2
Walls 4
InfiniteWall normal -1 0 0 position -0
InfiniteWall normal 1 0 0 position 0.01
InfiniteWall normal 0 -1 0 position -0
InfiniteWall normal 0 1 0 position 0.1
Boundaries 0
Particles 1
BP 0.005 0.049370932 0 0 0.11104265 0 0.0005 0 0 0 0 0 0 636.61977 5.0929582e+09 0
```

More examples

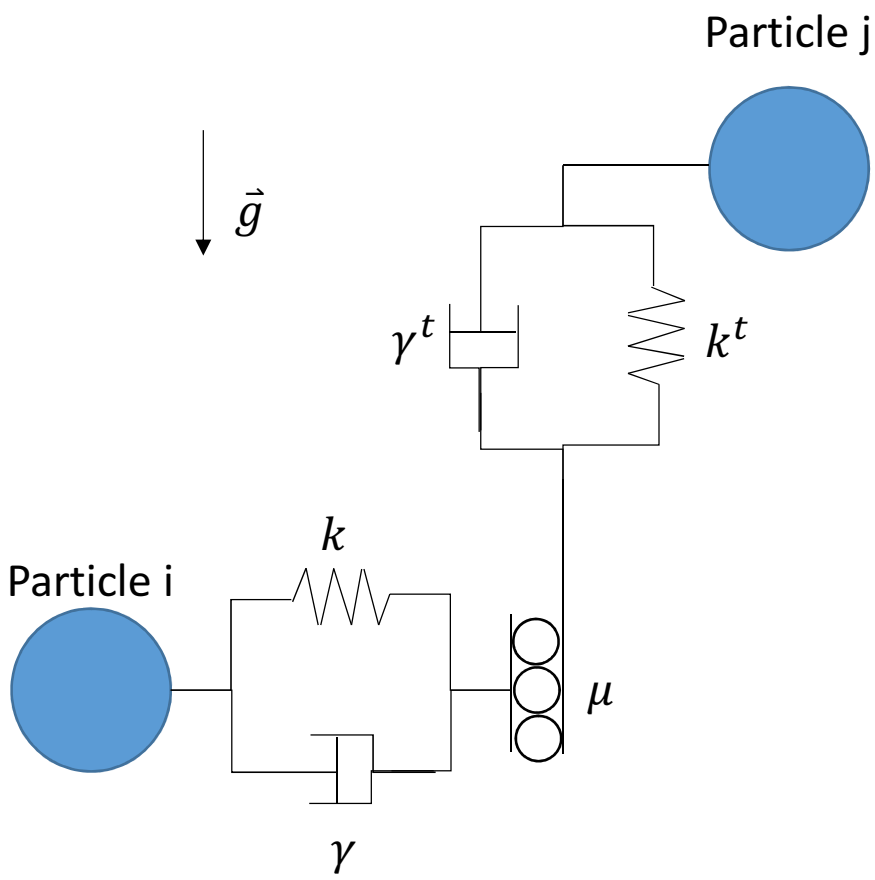
- elasticCollision (two particles)
- freeCooling (multiparticles, neighbourhood search, estimate of computing time)
- inclined plane (frictional contact forces)

Neighborhood search algorithms



- **Basic neighborhood search:** Particles check against all other particles
  Complexity $O(N^2)$
- **Linked Cell algorithm:** Particles check only against part. in neighbouring cells
  Complexity:
  O(N), but with high prefactor for polydispersed flows
- **Hierarchical grid:** Large particles have their own grid:
  Complexity O(N) for polydisp.
  See [OgarkoLuding2012]

Tangential contact dynamics



- Contacts are elastic-dissipative in normal direction,

$$f^n = -k^n \delta^n - \gamma^n v^n$$

- sliding friction works against movement in tang. direction

$$f^t = \min(\mu f^n, |\boldsymbol{f}_0^t|),$$

$$\boldsymbol{f}_0^t = -k^t \boldsymbol{\delta}^t - \gamma^t \boldsymbol{v}^t \ .$$

- additional to sliding, rolling and torsional friction torques are implemented.
- See [Luding 2008]

Discrete Particle Modeling of granular materials using MercuryDPM

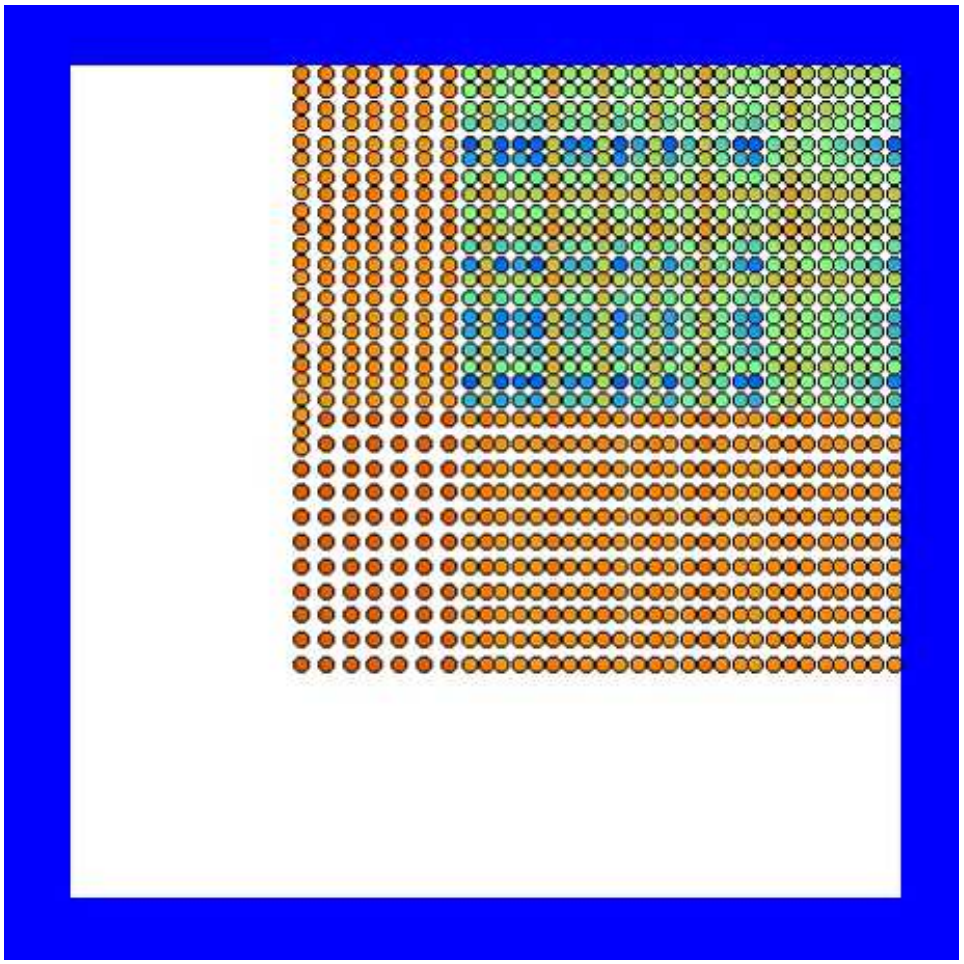
Part 2

Dec 16, 2013

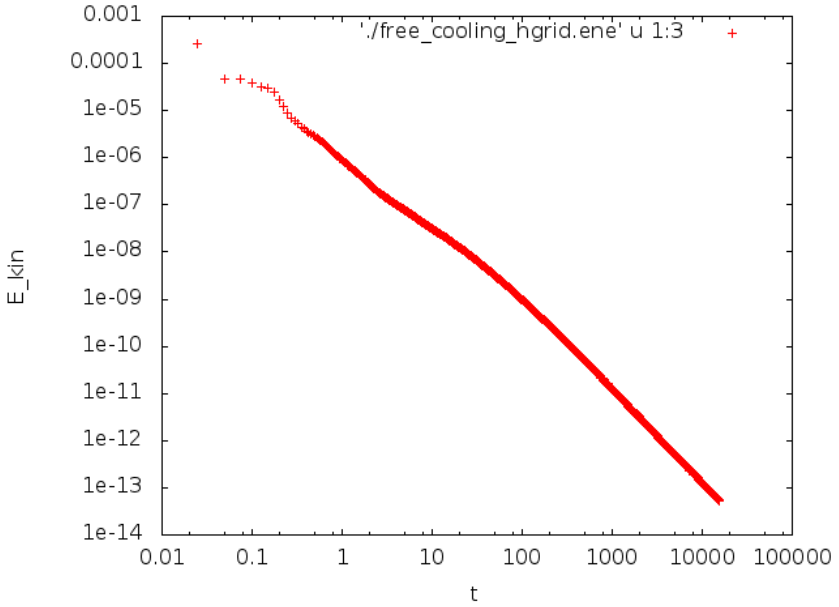
Managing a multi-developer code

- **svn** is used to manage distribution and development of code
  - **svn checkout http://mercurydpm.org/svn/MercuryDPM/alpha/ MercuryAlpha** to distribute the code
  - **svn update** to update you existing installation
  - **svn commit** to check in your updates (only for developers!)
  - **svn status** to see differences between your local installation and repository
- **cmake** is used to install the code
  - Takes care of compiler instructions.
  - Use **cmake** for default installallation.
  - For more options, choose **cmake-gui** or **ccmake**
- **doxygen** is used to document the code
  - Comments beginning with **///** become part of the documentation
  - create a local documentation using **make doc** (if doxygen is enabled in cmake-gui)

Example from last week: free_cooling_demo



Energy dissipates according to Haff's law ($E_{kin} = C\ t^{-\alpha}$)



Why particle simulations?

Continuum equations can be modeled at a much larger scale than particles and have therefore less DOF, i.e. are more efficient.

So why do particle simulations?

- Particles differ from fluids due to their finite size and their high energy dissipation
- Elasto-dissipative particles in gas state can relatively easily be described by continuum equations (kinetic theory)
- Frictional/ adhesive/ plastic forces make this task hard.
- For dense states (fluids and solids), continuum theory needs to be adjusted (shear behaviour) or might even require particle-sized resolution (jamming, cracks)

Creating a new driver code

- Create a new driver file **filename.cpp**
- Update CMake by running **cmake .** in your build directory.
  This allows you to compile the new file using **make filename**
- create a class inherited from MD (or a derived class, e.g. HGRID_3D):
  class newClass : public MD{ … };
- In **main()**, instantiate the class, set all parameters, and call solve()

```
int main () {
    newClass md;
    md.set…
    md.solve();
}
```

- Overwrite **setup_particles_initial_conditions()** to create walls and particles
  (if this requires parameters, define member variables, set them in **main()**)
- Run **cmake .** in your build directory.
  This allows you to compile the new file using **make filename**
- Example: **free_cooling_demo.cpp**

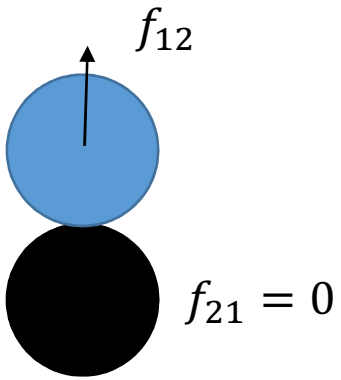
Creating a new driver code: Example

Let's create a new code where the bottom wall in free_fall_demo is replaced by a fixed particle.

Copy free_fall_demo.cpp into free_fall_with_fixed_particle.cpp and replace the wall definition by

```
void setup_particles_initial_conditions()
{
    BaseParticle p1;
    p1.set_Position(Vec3D(get_xmax()/2,0.0,0.0));
    p1.set_Velocity(Vec3D(0.0,0.0,0.0));
    p1.set_Radius(0.0005);
    p1.fixParticle();
    getParticleHandler().copyAndAddObject(p1);
    …
}
```

Fixed Particles

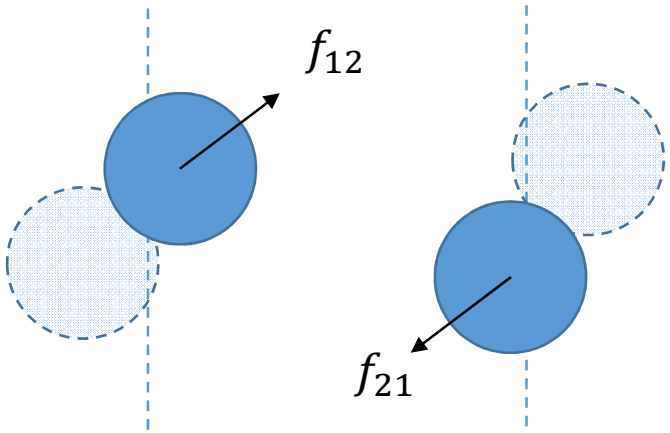


- Fixed particles have infinite mass and feel no forces / do not move (similar to walls)
- Used to create rough boundaries
- see hopper_demo.cpp

Ex: To fix particle P2, use:

```
BaseParticle P1, P2;
P2.fixParticle();
```

Periodic Boundaries 1/2

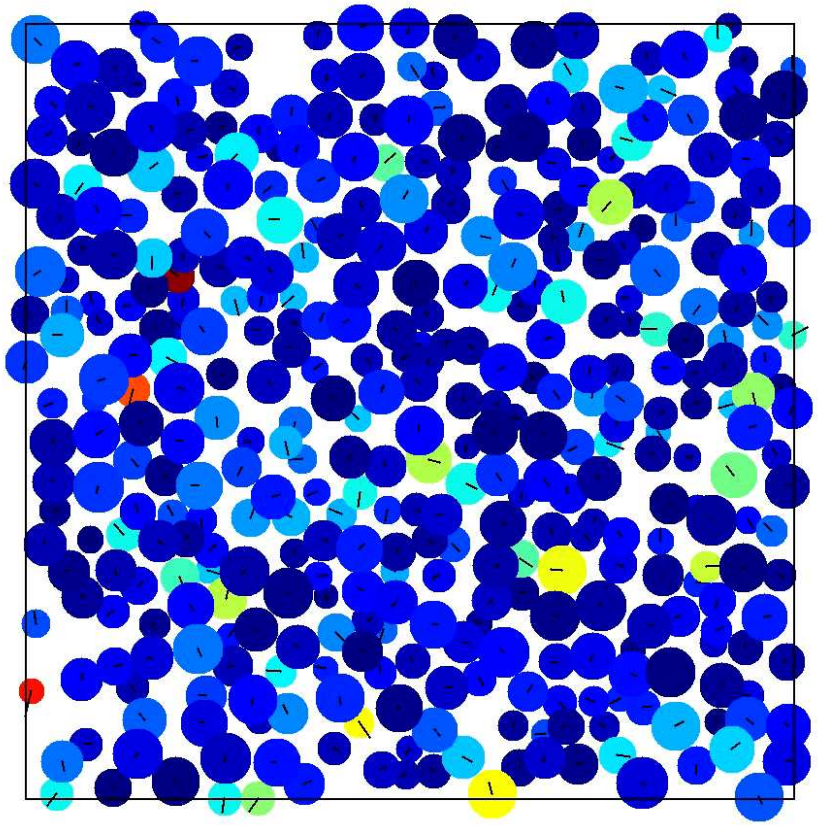


- Used to represent a volume element of a system without boundaries.
- Particles near the periodic boundary have ghost particles, allowing for contacts through the periodic boundary.
- See **periodic_walls_demo.cpp**
- See also **ShearCell2D.cpp** for more complex periodic boundaries

Ex: To implement a periodic boundary along the x-axis, use:

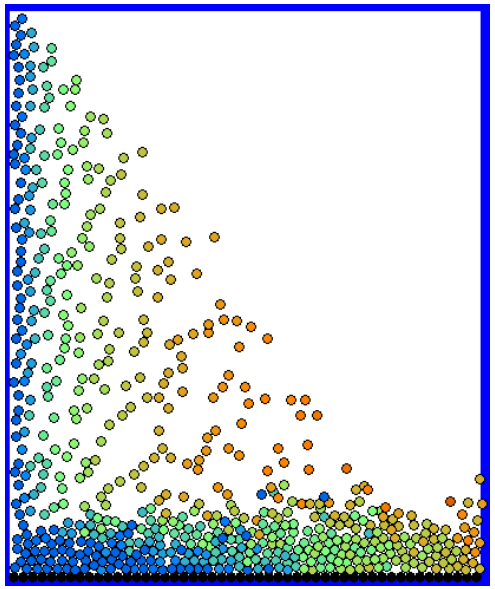
```
PeriodicBoundary b0;
b0.set(Vec3D(1,0,0), get_xmin(), get_xmax());
getBoundaryHandler().copyAndAddObject(b0);
```

Periodic Boundaries 2/2

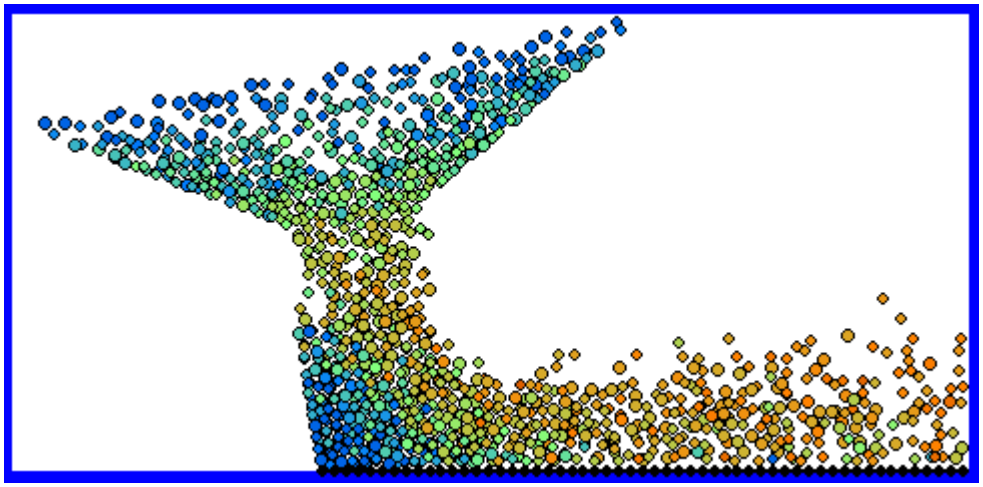


Also see Videos\Shear_100P_4cycles.avi

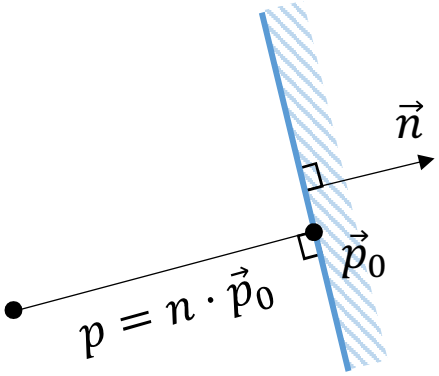
Inflow/Outflow conditions



- Creates a region where particles are continuously inserted, and a region where particles are deleted.
- Implemented in the Chute and Chute WithHopper class.
- See **chute_demo.cpp** (left) and **hopper_demo.cpp** (bottom)



Infinite Walls



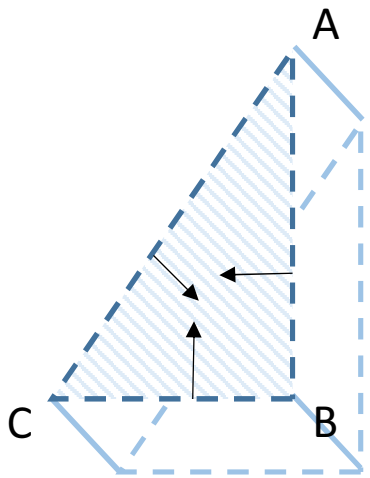
Walls are defined by a normal n that points into the wall and a position p:

$$\forall \vec{r}\colon \vec{n} \cdot \vec{r} = p$$

Ex: To implement walls at xmin and xmax, use:

```
InfiniteWall w;
w.set(Vec3D(-1,0,0), -get_xmin());
getWallHandler().copyAndAddWall(w);
w.set(Vec3D( 1,0,0),  get_xmax());
getWallHandler().copyAndAddWall(w);
```

Finite Walls

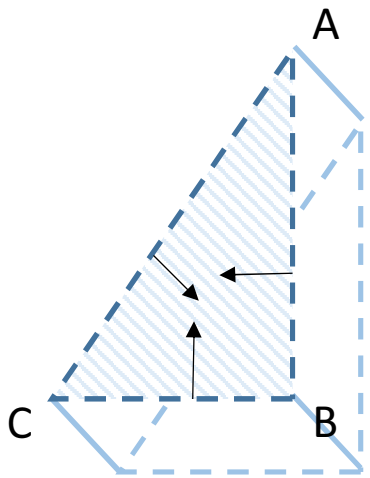


- Polyhedral walls can be created using finite walls
- See **HourGlass2D.cpp**

Ex: To implement a polyhedral wall between points A,B,C (points must be ordered in clockwise direction), use:

```
FiniteWall w1;
Vec3D D=Cross(B-C,A-B);
w1.add_finite_wall(Cross(A-B,D),A);
w1.add_finite_wall(Cross(B-C,D),B);
w1.add_finite_wall(Cross(C-A,D),C);
getWallHandler().copyAndAddWall(w1);
```

Finite Walls

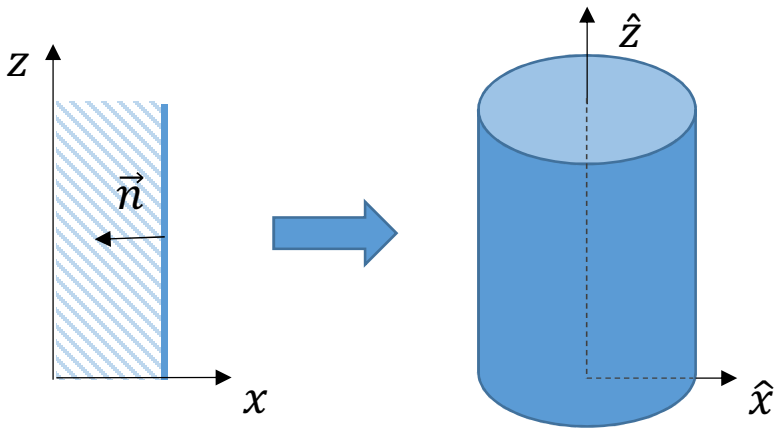


- Polyhedral walls can be created using finite walls
- See **HourGlass2D.cpp**

Alternative (works only for prismatic walls, i.e., all points in one plane)

```
FiniteWall w1;
std::vector<Vec3D> points;
points.push_back(A);
points.push_back(B);
points.push_back(C);
w1.create_prism_wall(Points);
```

Finite Axis-symmetric Walls

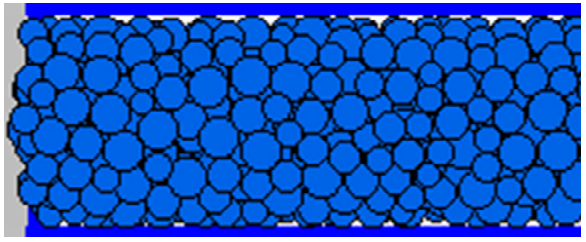


- Define a 2D FiniteWall in the xz-plane
- Define symmetry axis $\hat{z}$ and origin $\vec{O}$
- The axisymmetric wall will then be constructed by
  - translating the origin to $\vec{O}$,
  - rotating the z-axis to $\hat{z}$ and
  - rotating around $\hat{z}$
- See **HourGlass.cpp**

To define a cylindrical inner wall around the z-axis with radius r, use

```
FiniteAxisSymmetricWall w1;
Vec3D z = Vec3D(0,0,1);
Vec3D O = Vec3D(0,0,0);
w1.add_finite_wall(Vec3D(-1,0,0),-r);
w1.setSymmetryAxis(z,O);
getWallHandler().copyAndAddWall(w1);
```

Polydispersity/ random numbers



- Polydispersity avoids crystallization and thus unrealistic behaviour
- Random numbers can be accessed using random.get_RN(a,b)

For uniformly dispersed initial conditions with radii $r \in [r_0, r_1]$, use

```
BaseParticle p0;
p0.set_Velocity(0,0,0);
for (double x=get_xmin()+r1; x<get_xmax()-r1; x+=2.0*r1)
for (double y=get_ymin()+r1; y<get_ymax()-r1; y+=2.0*r1)
{
  p0.set_Radius(random.get_RN(r0,r1));
  p0.set_Position(Vec3D(x,y,0));
getParticleHandler().copyAndAddObject(p0);
}
```

The MD class and the solve routine

The solve routine is the work horse of the code.

- runs setup_particles_initial_conditions();
- runs actions_before_time_loop();
- runs time loop: while (t<dt<tmax)
  - runs do_integration_before_force_computation(*it);
  - runs output_xballs_data();
  - runs actions_before_time_step();
  - runs compute_all_forces();
  - runs actions_after_time_step()
  - runs do_integration_after_force_computation(*it);
  - runs output_ene();
  - increments in time t+=dt;

User interaction with the solve routine

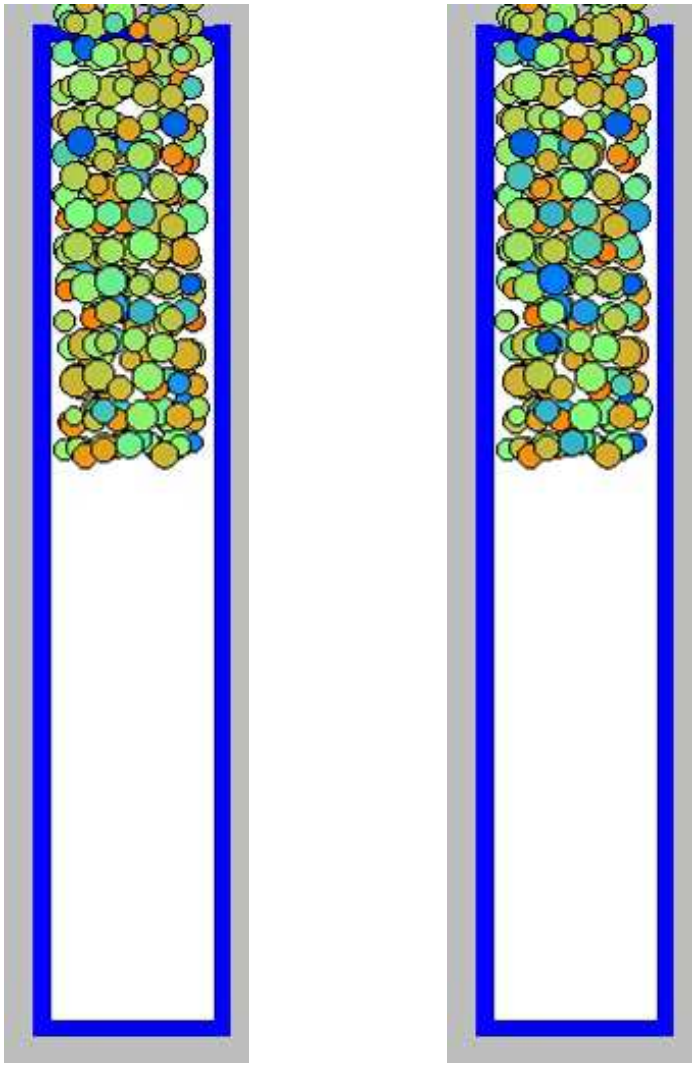
- One can easily time-dependent behavior by modifying actions_after_time_step()
- see, e.g., the HourGlass routine

To move a wall at t=0.9, use

```
void actions_after_time_step()
{
  if (get_t()<0.9 && get_t()+get_dt()>0.9)
  {
    ((InfiniteWall*)getWallHandler().getLastWall())
      ->set(Vec3D(0,0,-1), -get_zmin());
  }
}
```

Example: Hourglass.cpp

- Shows the effect of friction;
  the higher the friction, the less fluid the granular material behaves
- With high friction, even arching/ jamming is possible



Other advanced features

- Other contact models: Plastic, adhesive, Hertzian elastic, temperature-dependence, …
- Species: Define interactions between various types of particles, e.g. SpeciesDemo.cpp, wallSpecies_demo.cpp
- Restarting: Load previous state from a restart file, e.g. free_fall_restart_demo.cpp
- Special output: override output_ene() to modify the ene file output
- VMD output: we can show you how
- Self testing: Using make fulltest, the code automatically checks if you have made changes that affect the code
- …

Work in progress

MERCURYDPM

Version 1.0 expected end of March 2014

- Online tutorials/ full documentation
- Consistent naming convention
- Post-processing tool
- Binary output
- More visualization tools

Future:

- Parallelization
- Coupling with a continuum solver