Discrete particle simulations with MercuryDPM

Instructors: Anthony Thornton, Thomas Weinhart

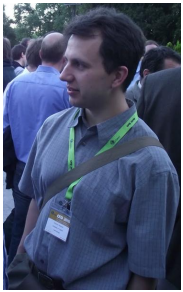
15 May 2025

A bit about the instructors

Co-founded MercuryDPM in 2009

Main Developer/Manager

A bit about the instructors



Co-founded MercuryDPM in 2009

Main Developer/Manager

Particle simulations with MercuryDPM

Particle simulations with MercuryDPM

MERCURYDPM

Open-source software for simulating granular materials

- Developed since 2009
- Modern C++ code base, easily extendable
- Funded by many research projects (>€10 mil.)
- Worldwide user/developer base
- Available at `mercurydpm.org`

Stanford University

UNIVERSITY OF CAMBRIDGE

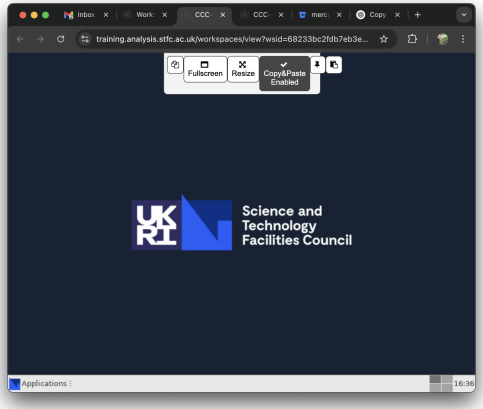
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

THE UNIVERSITY OF SYDNEY

FAU

But, how?

1. Go to `https://training.analysis.stfc.ac.uk/workspaces/`
2. Open workspace CCC-ParaSolS Training (CPU Only)
3. Hover on top of the window to get the menu below, and enable Copy&Paste



Exercise 0: Let's install - In 5 easy steps

1. Open a terminal
2. Create a folder
   ```
   mkdir MercuryDPM
   cd MercuryDPM
   ```
3. Download the code:
   ```
   git clone https://bitbucket.org/mercurydpm/mercurydpm.git MercurySource
   ```
4. Create the build system:
   ```
   mkdir MercuryBuild
   cd MercuryBuild
   cmake ../MercurySource
   ```
5. Build and test the code: `make fullTest`

Full instructions here: https://www.mercurydpm.org/downloads

Meanwhile while installing ...

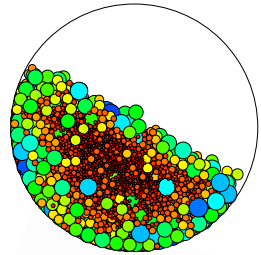
Directory Structure

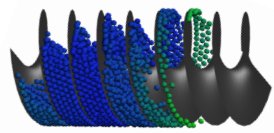
`MercurySource` contains many subdirectories:

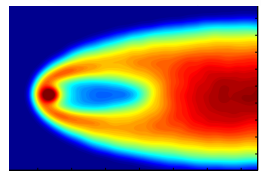
- `Kernel` contains all core functionality of MercuryDPM:
  Classes to store and manipulate particles, wall, boundaries, contact laws (species), and the DPM algorithm (`Mercury3D`).
- `Drivers` contains codes to simulate many different processes, tests, and demos. These codes use the classes defined in the Kernel.
- `Documentation` Text files and figures required to build the documentation
- `Configuration` Config files, e.g. for doxygen
- `Tools` Scripts for postprocessing and visualisation
- `Scripts` Matlab/Python scripts for postprocessing and testing
- `XBalls` A simple visualisation software

Particle simulations with MercuryDPM

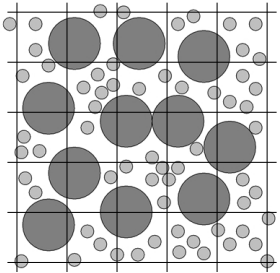
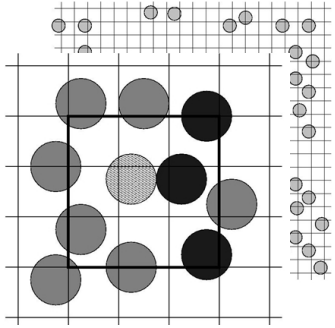
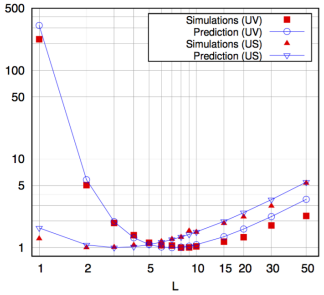
- Fast
  Contact detection algorithm
  allows polydisperse simulations
- Flexible
  Complex walls and
  boundary conditions
- Accurate
  Coarse-graining technique
  to extract continuum fields
- Open-source
  Available at mercurydpm.org



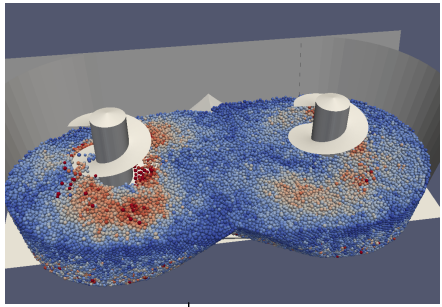


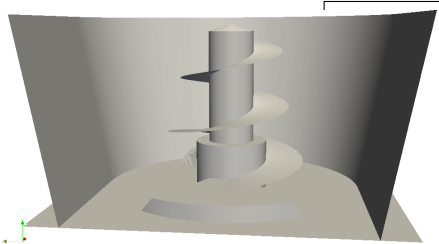


Fast computations (optimal contact detection)

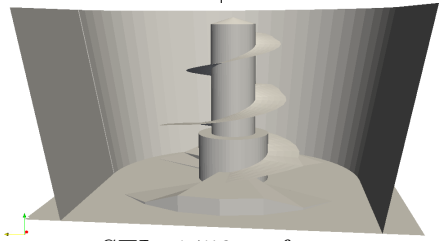
Traditional Linked Cell	MercuryDPM Hierarchical Grid	Speed-Up for high polydispersity
		
#checks: 1000	#checks: 89	up to 200x quicker!

Curved walls: flexible, efficient and accurate



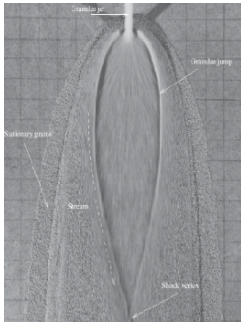


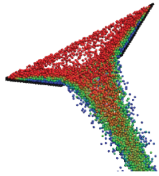
Curved walls: 5 surfaces

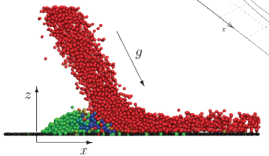


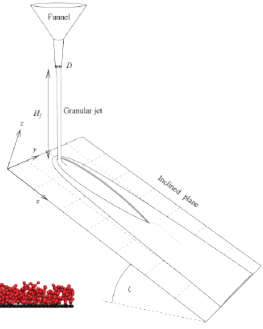
STL: 1456 surfaces

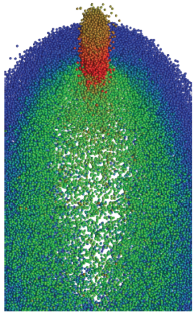
Accurate analysis via MercuryCG

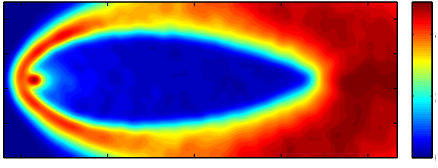




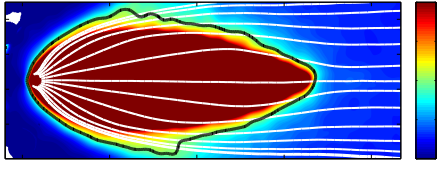








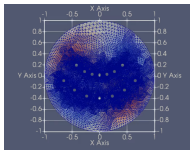
flow height

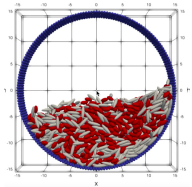


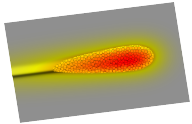
shocks & streamlines

Other key features

- Integration with fluid/solid solvers
- Fully parallelised
- Non-spherical particles
- Model complex physical interactions
- Model many complex processes

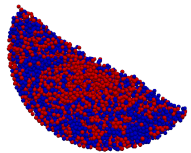


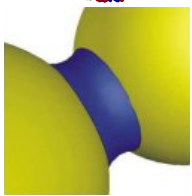


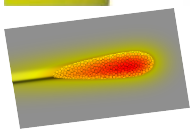


Other key features

- Integration with fluid/solid solvers
- Fully parallelised
- Non-spherical particles
- Model complex physical interactions
  - Cohesion/adhesion
  - Liquid bridges
  - Charged particles
  - Sintering/melting
  - Surface wear
  - Breakage
- Model many complex processes

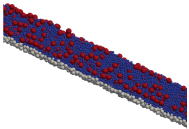


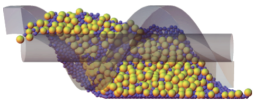


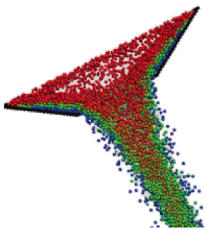


Other key features

- Integration with fluid/solid solvers
- Fully parallelised
- Non-spherical particles
- Model complex physical interactions
- Model many complex processes
  - Tabletting/agglomeration
  - Selective laser sintering
  - Cohesive mixing, granulation communition and deagglomeration
  - Conveying/transportation
  - Spray-drying







External Packages

- Git is used to download the code and get updates.
- CMake for compiling, CTest for self-testing.
- Python for self-testing and pre/postprocessing.
- Doxygen for documentation.
- X11 for xballs visualisation.
- Paraview for 3D visualisation.
- MPI for running parallelised simulations.

Git

MercuryDPM is stored in a git repository.

- Download the code:
  `git clone https://bitbucket.org/mercurydpm/mercurydpm.git`
- Update to the latest version: `git pull`
- Check for local modifications: `git status`
- Get other branches of the code: `git checkout 1.0.Alpha`

Why Git/Bitbucket?

Git gives version history, allows parallel development.
Bitbucket allows us to share the code open-source.

CMake

- All subfolders containing source files also contain a CMakeLists.txt file to define how the files should be compiled and linked together.
- Running `cmake-gui`, `ccmake` or `cmake` will create a build directory, that contains Makefiles to build the executables.
- Important: Rerun `cmake` each time you create a new source file.

Why CMake?

Using CMake, we tell the compiler how to create executables from the source files.

CMake

- CMake creates a build directory (`MercuryBuild`),
  which mirrors the structure of the source directory (`MercurySource`).
  - Source directory
    - Drivers
    - Kernel
    - CMakeLists.txt
    - ...
  - Build directory
    - Drivers
    - Kernel
    - CMakeCache.txt
    - ...
- Use `cmake .` to update the build instructions, e.g. when you add new files.
- Use `make name` to build the executable belonging to the driver `name.cpp`.
- Special make commands : `make fullTest` runs the test suite,
  `make clean` removes all compiled files, `make doc` builds the documentation.

CTest

- All drivers labelled `*UnitTest.cpp`, `*SelfTest.cpp`, `*MPI*Test.cpp` are part of the test suite. Every time the code is updated, we test whether these codes still produce the desired output.
- `make fullTest` runs the full test suite (clears, compiles, and runs all tests), `make test` runs all tests in a directory.
- If a test fails, it produces an errorlog (e.g. `STLReaderUnitTest.errorlog`).

Why CTest?

CTest allows us to check whether the current development has affected the existing code. This ensures that modifications don't break existing features.

Doxygen Documentation

- Uses doxygen to document all functions and classes in MercuryDPM
- Includes tutorials and demos, output files, visualisation, how to develop a new driver, how to use MPI, etc.
- Can be built locally with `make doc` (if turned ON in cmake)
- Web version: http://docs.mercurydpm.org/Trunk

Editors

IDEs

- Not just an editor, but a full-scale "development environment"
- Many features that usually operates on a "whole project"
  - Loads the project
  - Allows navigation between files
  - Provides autocompletion based on the whole project
  - Integrates with a version management system (like git)
  - A complete testing environment
- CLion, Eclipse, Visual Studio

Lightweight Editors

- Not as powerful as IDEs, but they're fast, elegant and simple
- They are mainly used to open and edit a file instantly
- However, in practice, lightweight editors may have a lot of plugins similar to IDEs, so there's no strict border between a lightweight editor and an IDE
- Sublime Text, NotePad++, Vim and Emacs

Let's crack on with real exercises!!!

Exercise 1: Build the documentation

- Go to `MercuryBuild`.
- Use cmake (or ccmake) to turn on the option of building the documentation:
  `cmake ../MercurySource -DMercuryDPM_BUILD_DOCUMENTATION=ON`
- Type `make doc` to create a local documentation
- Open the documentation in a web browser
  `firefox Documentation/html/index.html &`

Exercises: Tutorials 1-3

- Goto `MercuryBuild` and open the documentation in a web browser `firefox Documentation/html/index.html &`
- Click on "Fun with MercuryDPM (Tutorials)".
- Then, click on "Beginner tutorials".
- Read, execute and visualise the first three tutorials.

Deep dive into Tutorial 3

Output files

- `Tutorial3.data`
  contains particle positions, velocities, radii for multiple time steps
- `Tutorial3.fstat`
  contains contact points, forces, overlaps for multiple time steps
- `Tutorial3.ene`
  contains kinetic and potential energy, center of mass for multiple time steps
- `Tutorial3.restart`
  contains all parameters of the last saved time step.


fileName.data

- First line: N, Time, XMin, YMin, ZMin, XMax, YMax, ZMax
- N_p lines: x, y, z, vx, vy, vz, rad, q1, q2, q3, ox, oy, oz, species
- ... then repeats

```
1 0 0 0 0.01 0.1
0.005 0.095 0 0 0.005 -0 -0 0
1 0.0005 0 0 0.01 0.1
0.005 0.094998775 0 -0.0049 0.005 -0 -0 0
...
```

Used for visualisation and analysis.

fileName.fstat

- 3 lines: # time, info; # info; # info
- N_c lines: t, i, j, x, y, z, δ, δ^t, $f^{c,n}$, $f^{c,t}$, n_x, n_y, n_z, t_x, t_y, t_z
- ... then repeats

```
# 0 1
# 0 0 0 169.9 169.9 169.9
# 1 40 0 0 0 0
0 5 6 43.93 6.894 134.1 1.360 0 0 0 0.9995 −0.009 −0.0274 0 0 0
0 6 5 43.93 6.894 134.1 1.360 0 0 0 −0.9995 0.009 0.0274 0 0 0
...
```

Used for analysis.

fileName.ene

- Stores energy statistics.

t	ene_{gra}	ene_{kin}	ene_{rot}	ene_{ela}	X_{COM}	Y_{COM}	Z_{COM}
0	0.0009..	0	0	0	0.005	0.095	0
0.0005	0.0009..	$1.2e-08$	0	0	0.005	0.094	0
...							

Used for quickly checking whether the program runs as expected.

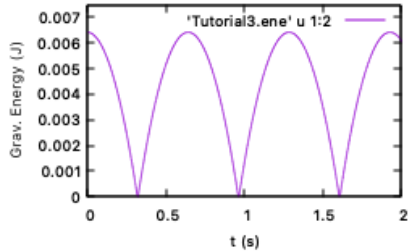
fileName.restart

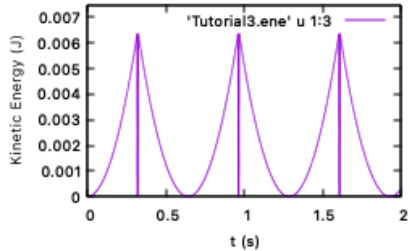
- Stores the full state of the simulation.
- Used for restarting, checking parameters.

```
MercuryDPM 0.14 name Tutorial3 revision 4946 repository https://mercurydpm.
dataFile     fileType MULTIPLE_FILES saveCount 10 counter 2000
fStatFile    fileType NO_FILE saveCount 10 counter 0
eneFile      fileType ONE_FILE saveCount 10 counter 2000
restartFile fileType ONE_FILE saveCount 10 counter 2000
xMin 0 xMax 1 yMin 0 yMax 1 zMin 0 zMax 1
timeStep 0.0001 time 2 ntimeSteps 20000 timeMax 2
systemDimensions 3 particleDimensions 3 gravity 0 0 0 writeVTK 0
Species 1
LinearViscoelasticSpecies id 0 density 2500 stiffness 258.5 dissipation 0
Walls 0
Boundaries 0
Particles 1
BaseParticle id 0 indSpecies 0 position 1 0.3 0.3 orientation 1 0 0 0 veloc
Interactions 0
```

T3 exercise solution

2. Open `gnuplot`
3. Plot gravitational and kinetic energy:
   `plot 'Tutorial3.ene' using 1:2 with lines`
   `plot 'Tutorial3.ene' using 1:3 with lines` [1]





We observe a small collision time (5ms), and no loss of velocity (r=1.0).

[1] You need to reduce savecount to 10 to get a detailed kinetic energy plot

Typical driver code

A typical driver code

```
#include "Mercury3D.h"

class Demo : public Mercury3D {
  void setupInitialConditions() override {
    //define geometric setup here
  }
};

int main() {
  Demo problem;
  //define process parameters here
  problem.solve();
}
```

- `Mercury3D` contains the DPM algorithm.
- The derived class `Demo` contains the initial conditions.
- The main function instantiates the class and calls the solve routine.

A typical driver code: the `main` function

Define parameters in the main function using set-functions:

```
...
int main() {
  Demo problem;
  problem.setName("Demo");
  problem.setMin(Vec3D(-1.0,-1.0,-1.0));
  problem.setMax(Vec3D( 1.0, 1.0, 1.0));
  problem.setTimeStep(1e-4);
  problem.setTimeMax(2.0);
  problem.setGravity(Vec3D(0,0,-9.81));
  problem.setSaveCount(200);
  problem.solve();
}
```

A typical driver code: `setupInitialConditions`

- Define the initial state in `setupInitialConditions`.
- You can declare four main types of objects:
  - Particles
  - Walls
  - Species (material and contact properties)
  - Boundaries (boundary conditions like periodic walls, insertion and deletion regions)
- Include the appropriate class, declare a new object, set its parameters and add it to the handler.

Let's look at a few examples ...


A typical driver code

First, define a species (material and contact properties):

```
#include "Species/LinearViscoelasticSpecies.h"
...
  void setupInitialConditions()
  {
    ...
    LinearViscoelasticSpecies species;
    species.setDensity(2000);
    species.setStiffness(1e3);
    species.setDissipation(0.1);
    speciesHandler.copyAndAddObject(species);
    ...
  }
...
```

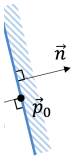
A typical driver code

Then, define particles:

```
#include "Particles/SphericalParticle.h"
...
  void setupInitialConditions()
  {
    ...
    SphericalParticle p;
    p.setSpecies(speciesHandler.getObject(0));
    p.setRadius(1e-3);
    p.setPosition(Vec3D(0.1,0.1,0.0));
    particleHandler.copyAndAddObject(p);
    ...
  }
...
```

A typical driver code

Now add walls by defining a wall's normal and position.




```
#include "Walls/InfiniteWall.h"
...
  void setupInitialConditions()
  {
    ...
    InfiniteWall w0;
    w0.setSpecies(speciesHandler.getObject(0));
    w0.set(Vec3D(0.0, 0.0, -1.0), Vec3D(0, 0, getZMin()));
    wallHandler.copyAndAddObject(w0);
    ...
  }
...
```

A typical driver code

Note: for MPI (parallel) simulations, you have to define the species in the main:

```
#include "Species/LinearViscoelasticSpecies.h"
...
int main(){
  ...
  LinearViscoelasticSpecies species;
  species.setDensity(2000);
  species.setStiffness(1e3);
  species.setDissipation(0.1);
  problem.speciesHandler.copyAndAddObject(species);
  ...
}
```

For an instructive full example, see `Tutorial3_BouncingBallElastic.cpp`.

Tutorials T4 -T9 exercises

Come join the team :)

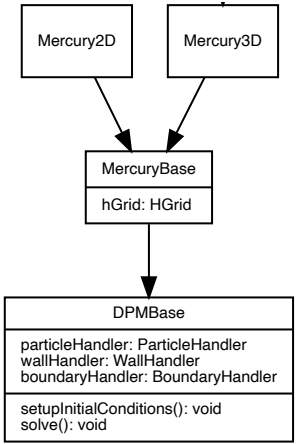
https://www.mercurydpm.org/about/team

Behind the scenes:
The MercuryDPM kernel

MercuryDPM's class structure

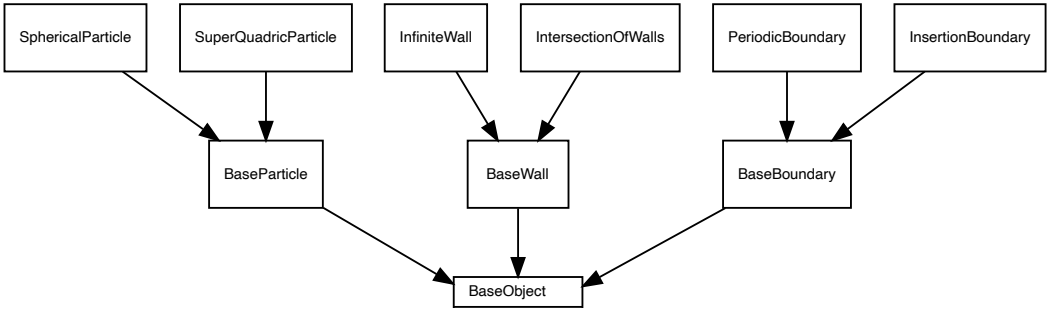
The main class of MercuryDPM is `Mercury3D`.

We use a cascade of classes to successively add complexity to the DPM algorithm:



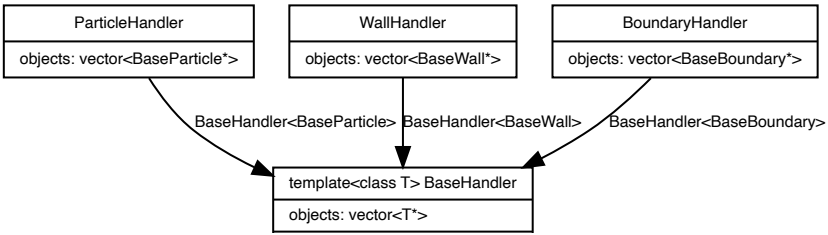
MercuryDPM's class structure

All other objects are also sucessively derived from simpler classes:



MercuryDPM's class structure

All objects are stored in handlers:



With that information, the driver codes are easier to understand.


The function solve()

The solve routine in `DPMBase.cc` is the work horse of the code. A simplified version:

```
void DPMBase::solve()
{
    if (isRestarted()) {
        actionsOnRestart();          // user-defined actions
    } else {
        setupInitialConditions(); // user-defined actions
    }

    while (getTime() < getTimeMax()) {
        computeOneTimeStep();
    }
}
```

The function computeOneTimeStep()

`computeOneTimeStep` evolves the system by one timestep. It also provides action-functions that can be specified by the user. A simplified version:

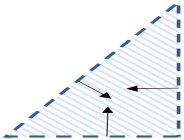
```
void DPMBase::computeOneTimeStep()
{
    integrateBeforeForceComputation(); // velocity-verlet part 1
    actionsBeforeTimeStep();           // user-defined actions
    computeAllForces();                // force computation
    actionsAfterTimeStep();            // user-defined actions
    integrateAfterForceComputation();  // velocity-verlet part 2
    time_ += timeStep_;
}
```

Material and contact properties

- `Species` contain material and contact properties.
- Material properties: density, etc
- Contact properties: stiffness, dissipation coefficient, etc
- Species can combine up to three different contact laws:

- The species name is a concatenation of the three force laws, e.g. LinearViscoelasticFrictionLiquidBridgeWilletSpecies
- For more about the various force laws implemented in MercuryDPM, visit `https://doi.org/10.1016/j.cpc.2019.107129`

Polygonal wall

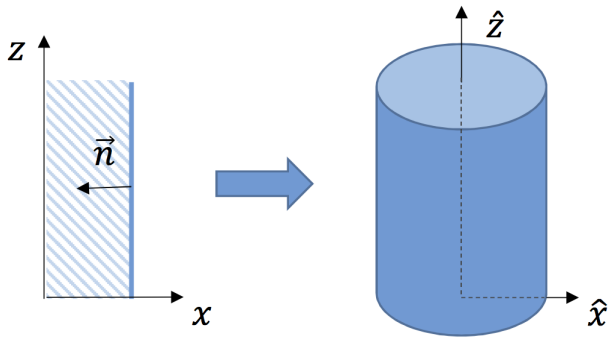


- Constructed by intersection of infinite walls.

```
#include "Walls/IntersectionOfWalls.h"
...
  void setupInitialConditions()
  {
    IntersectionOfWalls w;
    w.addObject(Vec3D(1,-1,0),Vec3D(0,0,0));
    w.addObject(Vec3D(1, 0,0),Vec3D(0,0,0));
    w.addObject(Vec3D(0,-1,0),Vec3D(1,0,0));
    wallHandler.copyAndAddObject(w);
  }
...
```

Axisymmetric walls

Obtained by rotating a 2D-polygonal wall around an axis:

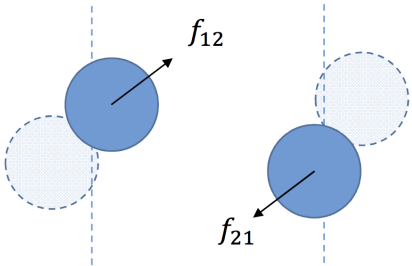



Axisymmetric walls

```
#include "Walls/IntersectionOfWalls.h"
...
  void setupInitialConditions()
  {
    AxisymmetricIntersectionOfWalls w;
    w.setPosition(0.,0.,0.);
    w.setOrientation(Vec3D(0.0,0.0,1.0));
    w.addObject(Vec3D(1,0,0), Vec3D(2.0,0.0,0.0));
    wallHandler.copyAndAddObject(w);
  }
...
```

See `HourGlass3DSelfTest` in `MercurySimpleDemos`.

Periodic boundaries



- Found in Kernel/Boundaries/.
- See documentation.
- Similar: AngledPeriodicBoundary
- Exercise: Modify `FreeCoolingDemo.cpp` to use periodic boundaries in x- and y-direction.

```
#include "Boundaries/PeriodicBoundary.h"
...
  void setupInitialConditions()
  {
    PeriodicBoundary b;
    b.set(Vec3D(1,0,0), getXMin(), getXMax());
    boundaryHandler.copyAndAddObject(b);
  }
...
```

LeesEdwardsBoundary

Simulates a shear box, see LeesEdwardsSelfTest.cpp.

```
double velocity = 1.0;
auto posShift = [velocity] (double time) {return time*velocity;};
auto velShift = [velocity] (double time) {return velocity;};
LeesEdwardsBoundary b;
b.set(posShift, velShift, getXMin(), getXMax(), getYMin(), getYMax());
boundaryHandler.copyAndAddObject(b);
```

InsertionBoundaries

- Inserts particles in a volume, e.g. cube.
- By default, inserts until no more particles can be inserted (maxFailed criterium).
- Use `setPSD(..)` to set a particle size distribution.
- `setInitialVolume(..)`, `setVolumeFlowRate(..)` to insert a defined amount.
- Change the defaultParticle properties to determine what gets inserted.

```
SphericalParticle defaultParticle;
p.setSpecies(speciesHandler.getObject(0));

CubeInsertionBoundary b;
b.set(&defaultParticle,maxFailed,posMin,posMax,velMin,velMax,radMin,
    ↪ radMax);
boundaryHandler.copyAndAddObject(b);
```

DeletionBoundaries

- Deletes particles in a volume.

```
DeletionBoundary b;
b.set(Vec3D(0, 0, -1), -minHeight);
boundaryHandler.copyAndAddObject(b);
```

Fixed particles, prescribed position

- Particles and walls can have a prescribed position/velocity/angular velocity
- To create rough surfaces

```
SphericalParticle p;
p.fixParticle();
...
```

```
auto position = [] (double t)
  { return Vec3D(0, 0, std::sin(t)); };
InfiniteWall w;
w.setPrescribedPosition(position);
...
```

Parameter studies

- `autonumber()` saves files with an appended number.
- The number can be used to run parameter studies.
- Set up a 4-by-4 parameter study:

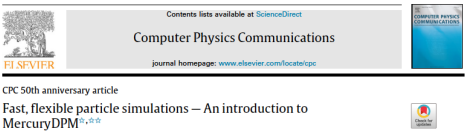
```
int main(){
  ...
  problem.autoNumber();
  std::vector<int> study_num=problem.get2DParametersFromRunNumber(4,4);
  double parameter1 = 2.0 + (study_num[1]-1.)*0.5;
  double parameter2 = (0.2 + 0.1*study_num[2]);
}
```

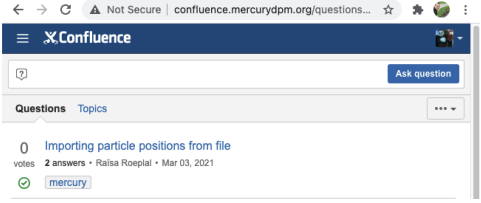
Further reading

- Lots of resources here: https://www.mercurydpm.org/documentation
- Including the text book: https://oercommons.org/courseware/lesson/113895
- Or the source for the must up-to version are here: https://bitbucket.org/mercurydpm/scientificcomputingcourse/src/master/
- This includes the version for this course

Further reading

- This journal article gives a good overview of the implementation and features: `https://doi.org/10.1016/j.cpc.2019.107129`
- If you have questions, please visit our Q&A: `https://mercurydpm.atlassian.net/wiki/spaces/CQ/overview/`





Come join the team :)

https://www.mercurydpm.org/about/team