IMPETUS – Interactive MultiPhysics Simulation Environment

Tutorial File 1

Vi Q. Ha[1] and George Lykotrafitis[1,2]

[1]Mechanical Engineering, [2]Biomedical Engineering, University of Connecticut, Storrs, CT 06269

http://engr.uconn.edu/~gelyko/impetus.html

IMPETUS – Interactive MultiPhysics Simulation Environment

Tutorial 0: Using the Tutorial

The tutorial files are available at http://engr.uconn.edu/~gelyko/impetus.html. The provided zip file includes the library for the simulating engine, the tutorial simulation files, main.cpp, and a makefile. It is recommended that users compile the program using OpenMPI as other compilers are not guaranteed to work.

Selecting the Tutorial Files:

The tutorial demonstrates how to build three different types of simulations: a basic molecular dynamics simulation using "`CellSpace`" MID, diffusion in a continuum field using "`InteractiveField`" MID, and a chemotaxis model where particles obey to particle dynamics equations and at the same time can detect the concentration continuum field by coupling a "`CellSpace`" and an "`InteractiveField`". All tutorial files are located in the folder named "Tutorials". To access different tutorials, adjust the included header file in main.cpp. Additional instructions are shown in the main.cpp file. There are four tutorial files:

- 1a) Simple NVE molecular dynamics simulation:

  `Tutorials/tutorialSimpleMD/simple.h`

- 1b) NVE molecular dynamics simulation of a liquid:

  `Tutorials/tutorialSimpleMD/liquid.h`

- 2) Solving the diffusion equation with the continuum field

  `Tutorials/tutorialHeat/heat.h`

- 3) Simulating Chemotaxis

  `Tutorials/tutorialChemotaxis/chemotaxis.h`

Compiling the Simulation:

We provide a makefile to compile the simulation. Confirm that the programs 'make' and OpenMPI are installed in your system. The user is responsible for providing the path to the OpenMPI compiler in the Makefile. If the IMPETUS library is not in the same folder, the user is also required to change the path of the IMPETUS library to the correct location.

- To compile the program, use the command:

```
make
```

- Optionally, the user can clear up the object files by using `make clean` before `make`:

```
make clean

make
```

If the user is planning to use an MPI "machinefile", please modify the file named `nodefile` accordingly. After compilation, the command to run the program should look similar to:

```
/home/shared/openmpi/bin/mpirun -n 8 -machinefile nodefile bin/impetus-run.exe
```

Tutorial 1: Creating a simple Molecular Dynamics Simulation

1.1 Introduction

Here, we provide instructions on how to create a simple Molecular Dynamics simulation. Users will be running (i) an NVE simulation of 8000 particles interacting via a simple spring potential, and (ii) a liquid simulation using the LJ potential. In addition, the tutorial provides instructions on how to record the mean square displacement and compute the radial distribution function.

- The files for these examples are located in:

```
Tutorials/tutorialSimpleMD/
```

- To include the header for a simple molecular dynamics simulations insert the following line in main.cpp

```
#include "../Tutorials/tutorialSimpleMD/simple.h"
```

1.2 The Parameters Object

The C++ IMPETUS object "`Parameters`" is the core of the simulations. The purpose of the "`Parameters`" is to store all the parameter variables used in a simulation. All MIDs and many other functions will refer to the pointer of "`Parameters`" to obtain simulation variables. It can be constructed with or without using an input file. For the scope of this tutorial, we will construct it using an input file. The input argument for constructing a "`Parameters`" object is the `MPI_Comm` object, commonly named as the global variable: `MPI_COMM_WORLD`. For details, please refer to the online application program interface (API) http://engr.uconn.edu/~gelyko/impetus.html.

Some of the variables in the "`Parameters`" object are imperative to construct the simulation. Such mandatory parameters are:

- Processor division of the simulation. For example, to divide the simulation to 8 processes in a 2x2x2 partition, the user should assign the value of 2 to each of the following "`Parameters`" variables:

```
int gridinfo->proc.np[3]
```

- The user is required to assign the lower and upper coordinates of the total simulation box:

```
double gridinfo->world.lo[3]

double gridinfo->world.hi[3]
```

Some of the parameters are optional. However, the use is strongly encouraged to assign a value to them. These parameters are:

- Time scale:

```
double delta_t
```

- Number of steps for the simulation:

```
int end_step
```

The optional parameters that are required by some commonly used functions are:

- The desired temperature of the simulation which is required for thermostats:

```
double desired_temperature
```

- The atomeye cfg printing frequency for cell space, continuum space, and network space:

```
int cell_print_interval

int cont_print_interval

int net_print_interval
```

- The number of "`GlobalNetwork`", "`CellSpace`", "`InteractiveField`" that the user intend to use in the program:

```
int n_net

int n_cell

int n_cont
```

An example of how to create the "Parameters" object using an input file is as follows:

```
/// Initiating and declaring the param object:
vamde::Parameters * param = new vamde::Parameters(MPI_COMM_WORLD);


/// Read in the values from the param.input, please modify the path
accordingly.
param->readinput("Tutorials/tutorialSimpleMD/input/param.input" );
```

An example of param.input is as follows:

```
[IN] delta_t
0.01

[IN] end_step
1000

[IN] cell_print_interval
100

[IN] cont_print_interval
100

[IN] proc.np
2 2 2

[IN] world.lo
0 0 0

[IN] world.hi
20 20 20

[IN] desired_temperature
1

[IN] number of networks
0

[IN] number of cell list space
1

[IN] number of continuum space
0
```

Common "Parameters" Functions:

In addition, the "Parameters" object provides functions that are commonly used in simulations.

- Getting the rank number of current simulation process:

```
int rank()
```

- Getting the simulation time:

```
double time()
```

- Getting the simulation time step

```
int step()
```

- Advancing the simulation clock by one step

```
void clock->advance();
```

1.3 The "CellSpace" Object

The "CellSpace" object is the short-range particle dynamic MID built using LCM (refer to Online Methods). It handles all particle dynamics functions. The "Parameters" object is required to create the "CellSpace". An example on how to create the "Parameters" object using an input file is given below:

```
vamde::CellSpace * s0;
s0 = new vamde::CellSpace("path/to/cell0.input",param);
```

Mandatory parameters required for "CellSpace" to be constructed:

- The minimum_cell_size variable also referred to as rcut in the input file is the minimum size of a cell in the LCM. Note that the space is automatically divided up evenly to all cells so the resulting size of the cells will be at least as large as the provided minimum_cell_size.

```
double minimum_cell_size
```

- The variables "sigma" and "mass" represent the size and mass of the particles in the "CellSpace", respectively. They are used in the molecular dynamics potentials which are provided with the library. However, they are not absolutely necessary if users plan to build their own potential functions:

```
double sigma;
```

```
double mass;
```

- Some functions, such as printing of cfg files requires the following output path variable:

```
char *output_directory_name;
```

"CellSpace" has functions that automatically generate simple initial configurations for particles but users can build their own initial configurations by using the "createParticle()" function. This is discussed in the online API:

- "CellSpace" automatically provides a uniformly distributed set of particles as an initial configuration. The following variables describe the distribution:

```
int particle_distribution[3];
```

- Users can assign two types of initial velocity distribution. Set "velocity_distribution_type" to 0 to assign the same initial velocity to all particles and to 1 to assign random uniformly distributed particle velocities ranging between a provided minimum and maximum value:

```
int velocity_distribution_type ;
```

- The 3 components of the constant initial velocity distribution are assigned to:

```
double orderly_velocity_distribution [3];
```

- The minimum and maximum value of the uniform velocity distribution are assigned to:

```
double uniformly_randomly_velocity_distribution_lower[3];

double uniformly_randomly_velocity_distribution_upper[3];
```

An example of `cell0.input` is as follows:

```
[IN] rcut
1.122462048309373

[IN] sigma
1

[IN] mass
1

[IN] particle distribution
20 20 20

% velocity distribution type: 0 for orderly, 1 for uniformly random.
[IN] velocity distribution type
0

[IN] orderly velocity distribution
0.3 0.3 0.3

[IN] uniformly randomly velocity distribution lower
-1 -1 -1

[IN] uniformly randomly velocity distribution upper
1 1 1

[IN] output directory name
Tutorials/tutorialSimpleMD/output/Cell_outputs/cell0/
```

Common "CellSpace" Functions

"CellSpace" provides a large number of functions for users to create simulations. The commonly used functions are the following:

- "moveParticles()" is used for moving particles across processors. Particles that are not within the simulation box of the current processor are moved to the neighbor processors.

```
void moveParticles();
```

- "move_particles_to_the_correct_cells()" is used to move particles to the correct cell in the cell list according to their positions within the same processor. This is recommend to use after integration and before moving particles across processors with "moveParticles()"

```
void move_particles_to_the_correct_cells();
```

- "copyParticles()" is used for copying particles that are on the borders of a processor to the shell layer of adjacent processors as pseudo particles for pair computations. This is required to perform pair-potential calculations.

```
void copyParticles();
```

- "deleteBorders()" is used to delete pseudo particles in shell layers. It is recommended to be used before performing "move_particles_to_the_correct_cells()" to avoid the deletions of particles that are moved to the shell layer in preparation for "moveParticles()"

```
void deleteBorders();
```

- "cfgwriter->print()" is used to print .cfg files for visualization. The files are printed to the provided output directory name. Please refer to the online API for complete instructions on how to visualize these files.

```
void cfgwriter->print();
```

1.4 Using Iteration Classes

The program provides a very organized way for users to implement potentials and functions as subclasses to one of the "iteration" base classes. Two examples are `getPartList` and `getCellList`. The users create function objects out of these classes. The input arguments for these objects are the pointers of the "`CellSpace`" of the particles that the user iterates. More "iteration" classes are available in IMPETUS.

The "`getPartList`" class is commonly used to program single action to particles. The "`getCellList`" is commonly used to program pair actions between particles and their neighbors in the cell list. The following is an example on how to use "`getPartList`" to set the force values of all particles to zero in preparation of new calculations. The actions are programmed in the function "`action(Particle *i)`":

```
class ZeroForce1 : public getPartList {
      public:
      ZeroForce1(vamde::CellSpace *_s) : getPartList(_s) {}
            void action(Particle *i)  {
                  for (int d=0; d<DIM; d++) {
                        i->F[d] =0;
                  }
            }
};
```

Examples on how to initiate and use the functions defined above:

```
ZeroForce1 zeroforce(s0);

zeroforce.apply();
```

Next, we show an example on how to use "`getCellList`" to create a simple spring potential between pairs of neighboring particles. The actions are programmed in "`action(Particle *i, Particle *j)`". Note that users only need to modify i and not j, and the loop will provide both the `i<->j` interaction and the `j<->i` interaction:

```
class Spring1 : public getCellList {
      public:
      Spring1(vamde::CellSpace *_s) : getCellList(_s) {}

      void action(Particle *i, Particle *j) {
            real sigma = 1.0;
            real epsilon = 5.0;
            real r = 0.0;
            for (int d=0; d<DIM; d++)
                  r += sqr(j->x[d] - i->x[d]);
                  real dx = sqrt(r);
                  if (dx <= 1.1) {
                        real f =  epsilon * (dx - sigma) ;
                        for (int d=0; d<DIM; d++){
                              i->F[d] += f * (j->x[d] - i->x[d]) ;
                        }
                  }
            }
};
```

Examples on how to initiate and use the functions defined above:

```
Spring1 spring(s0);

spring.apply();
```

The following example illustrates how to employ "getPartList" in order to implement a slightly more complex function. Here, we use the leapfrog algorithm. The "LeapFrog1" class can have its own variable and functions (e.g `int PART` , `void step(int _step)`). Note that the "CellSpace" was constructed using "Parameters", therefore users can call the "Parameters" object from "CellSpace".

```
class LeapFrog1: public getPartList {
      public:
      LeapFrog1(vamde::CellSpace *_s) : getPartList(_s) {
                  PART = 0;
                  dt =_s->param-> delta_t;
            }
            void step(int _step){
                  PART = _step;
                  apply();
                  PART = 0;
            }
      private:
            int PART;
            double dt;
            void action(Particle *i)  {
                  for (int d=0; d<DIM; d++) {
                        if (PART == 1) {
                              i->v[d] = i->v[d] + (0.5* dt * i->F[d]/i->m);
                              i->x[d] = i->x[d] + ( dt * i->v[d] );
                        } else {
                              i->v[d] = i->v[d] + (0.5* dt * i->F[d]/i->m);
                        }
                  }
            }
};
```

Examples on initiating and using these functions:

```
LeapFrog1 leapfrog(s0);

leapfrog.step(1);

leapfrog.step(2);
```

1.5 Creating a Simple Molecular Dynamics Simulation

By combining what we discussed up to this point in this tutorial, we can build simple molecular dynamics simulations using the provided library:

```
int main(int argc, char *argv[]) {

	MPI_Init(&argc, &argv);

	/// Creating Parameters object
	vamde::Parameters * param = new vamde::Parameters(MPI_COMM_WORLD);
	param->readinput("workspace/noinput_20160503/input/param.input" );

	/// Creating CellSpace
	vamde::CellSpace * s0;
	s0 = new
vamde::CellSpace("workspace/noinput_20160503/input/cell0.input",param);

	Spring1 spring(s0);
	LenardJones1 lenardjones(s0);
	ZeroForce1 zeroforce(s0);
	LeapFrog1 leapfrog(s0);

	/// print Cfgs
	s0->cfgwriter->print();
	while (param->step() < param->end_step) {

		/// Advance the clock by one step
		param->clock->advance();

		/// LeapFrog step 1
		leapfrog.step(1);

		/// delete shell particles
		s0->deleteBorders();
		/// move particles to new cells after integration before moving
interprocessor
		s0->move_particles_to_the_correct_cells();
		/// move particles across processors.
		s0->moveParticles();
		/// copy new ghost particles
		s0->copyParticles();

		/// Set all forces to zero
		zeroforce.apply();

		/// Apply spring pair potential
		spring.apply();
		//~ lenardjones.apply();

		/// LeapFrog step 2
		leapfrog.step(2);

		/// print Cfgs
		s0->cfgwriter->print();

	}
}
```

1.5.1 Visualization with Atomeye

To view particle configurations, we use the visualizer Atomeye. We provide functions to print out the particle coordinate files. In addition, we also provide methods to convert these files to configuration files for the Atomeye. For more information, please visit http://li.mit.edu/Archive/Graphics/A/ .

- The coordinates of the particles are saved in the "`output_directory_name`" variable that is assigned to the "`CellSpace`" object. For example,

```
[IN] output directory name

Tutorials/tutorialSimpleMD/output/Cell_outputs/cell0/
```

- To generate the configuration file, go to the directory input above and run the provided `.m` executable script to convert the output coordinate files to Atomeye format. Users have the option to run the script in Octave, FreeMat, or Matlab. The name of the script is:

```
mergeCellCfgOctave2016.m
```

- The converted file is saved in the folder named `Cfg/`

Note that users can use the provided class template for coordinate printing to create their own functions to print out files for other visualizers. User can also write their own scripts to convert the coordinate files to the appropriate format of their visualizer.

1.5.2 Building a Lennard-Jones (LJ) Liquid Model

This tutorial demonstrates how to build the LJ liquid simulation.

- Include the header for the simple molecular dynamics simulation by putting this line in `main.cpp`

```
#include "../Tutorials/tutorialSimpleMD/liquid.h"
```

- Note that the main difference between the liquid simulation and the simple molecular dynamics simulation discussed above is the use of the LJ potential instead of the spring potential. LJ potential (expression) is declared as follows:

```
LennardJones1 lennardjones(s0);
```

- The LJ function provided in this tutorial allows users to apply, by default, the entire LJ potential or only the repulsive part. When the repulsive LJ potential is implemented, make sure that the `rcut` in the input file is at least 1.12246 and then use the following function:

```
"lennardjones.setRepulsive()"
```

- To apply the pair potential to all close-range particles in `s0`, set the `rcut` to at least 2.5 and apply the following function:

```
"lennardjones.setAttractive()";
```

- To use LJ, insert the following:

```
"lennardjones.apply()";
```

In addition, here we discuss two postprocessing functions: the measurement of (i) the radial distribution function (RDF) and (ii) the mean square displacement (MSD) distribution **(Supplementary Note 1)**. For RDF, build the "`CellSpace`" with an `rcut` of 4.0. The codes for MSD and RDF are not discussed in the **Supplementary Information** document but are provided in the online tutorial file. To construct and use of the functions insert:

```
MSD1 msd(s0);

msd.print();


RDF1 rdf(s0);

rdf.print();
```

For this tutorial, the files are saved in:

`Tutorials/tutorialSimpleMD/output/MSD/`

`Tutorials/tutorialSimpleMD/output/RDF/`

The user can view the plots using the provided .m Octave executable function in `Tutorials/tutorialSimpleMD/output/plotMSDandRDF.m`

Results for a very similar simulation but in NVT ensemble is shown in **(Supplementary Note 1, Supplementary Figure 1)**.

Tutorial 2 : Solving the Diffusion equation using Interactive Field

2.1 Introduction

This tutorial will guide users to create a simple simulation to solve the diffusion equation $\partial C(\mathbf{r},t)/\partial t = D\nabla^2 C(\mathbf{r},t)$ using the "`InteractiveField`" MID. Where $C(\mathbf{r},t)$ is the concentration and D is the constant diffusion coefficient, by using the continuum field component of the program.

- The tutorial is located in:

```
Tutorials/tutorialHeat/
```

- For a diffusion simulation, insert the line below in the main.cpp:

```
#include "../Tutorials/tutorialHeat/heat.h"
```

2.2. The "InteractiveField" Object

The "`InteractiveField`" object is the discretized physical space defined by a grid of nodes. Field equations can be solved numerically at the nodes. Here, we follow the explicit finite difference method. The "`Parameters`" object is required to create the "`InteractiveField`" object. The following example shows how to create the object using an input file:

```
vamde::InteractiveField * c0;

vamde::InteractiveField::Cinit continit;

continit.readinput("Tutorials/tutorialHeat/input/cont0.input");

c0 = new vamde::InteractiveField(continit,param);
```

The required parameters for constructing the "`InteractiveField`" are

- Minimum number of total nodes on each axis: `nx, ny, nz`. The simulation engine will divide the continuum to the processors and rounding up so that the number of nodes is the same in every processor. Therefore the resulting number of `nx, ny` and `nz` may be slightly higher than the assigned value.

  ```
  int nx, ny, nz;
  ```

- Ghost layer is the size of the shell layer used in processor communication

  ```
  int ghost_layer;
  ```

- A printing path is required for the generated .cfg configuration files but it is not required to construct the field:

```
char *output_directory_name;
```

An example for an input file `cell0.input` is shown below:

```
[IN] nx ny nz
120 120 120

[IN] ghost_layer
1

[IN] output directory name
Tutorials/tutorialHeat/output/Cont_outputs/cont0/
```

Common InteractiveField Functions:

- The diffusivity can be adjusted:

```
c0->diffusivity = 0.05;
```

- Several boundary conditions can be used. Examples on how to use sink/source $C(\mathbf{r},t)$ $C(r,t)=k$ and insulators $\frac{dC}{dt}(r,t)=0$ as boundary conditions are as follows:

```
c0->setAllGlobalBoundarySink();

c0->setAllGlobalBoundaryInsulated()
```

- Users can set a concentration value "`concentration_val`" to a certain location `x0, y0, z0` as follow:

```
c0->setConcentration( x , y , z , concentration_val);
```

- To print configuration .cfg files to a provided output directory name, we use the function:

```
c0->cfgwriter->print();
```

- To copy the nodes of the border of one processor to the shell layer of its adjacent processor as pseudo nodes, we use the function:

```
c0->copyParticles();
```

- This function will specifically integrate the parabolic equation.

```
c0->IterateParabolicPDE();
```

Creating a Simulation

The following example shows how to use the functions defined above along with what we discussed in **(Supplementary Software 2)** to build a simple diffusion simulation.

```
void runSimulation(vamde::Parameters * param)  {

	vamde::TimeKeeper * tk;
	tk = new vamde::TimeKeeper(param);

	/// parameters:
	double delta_t = param-> delta_t;
	double end_step = param-> end_step;
	double & t = param->clock->t;
	int & step = param->clock->step;
	/// Initiate clock
	param->clock->set_step(0);

	vamde::InteractiveField * c0;
	vamde::InteractiveField::Cinit continit;
	continit.readinput("Tutorials/tutorialHeat/input/cont0.input");
	c0 = new vamde::InteractiveField(continit,param);
	c0->diffusivity = 0.05;

	double x_mid = (param->gridinfo->world.lo[0] + param->gridinfo-
>world.hi[0]) /2;
	double y_mid = (param->gridinfo->world.lo[1] + param->gridinfo-
>world.hi[1]) /2;
	double z_mid = (param->gridinfo->world.lo[2] + param->gridinfo-
>world.hi[2]) /2;

	c0->setConcentration(x_mid,y_mid,z_mid,10 );
	c0->setAllGlobalBoundarySink();
	c0->cfgwriter->print();

	while (step < end_step) {

		/// Print Runtime progress
		tk->print_progress(10);
		/// Advace the clock by one step
		param->clock->advance();

		c0->copyParticles();
		c0->IterateParabolicPDE();
		c0->cfgwriter->print();

	}
}
```

Results are shown in **(Supplementary Note 5, Supplementary Figure 5)**.

Tutorial 3: Simple Chemotaxis Model

3.1 Introduction

This section of the tutorial will guide the users to couple the "CellSpace" MID with the "InteractiveField" MID into one simulation. We will demonstrate a numerical model that simulates chemotaxis. Particles from "CellSpace" will detect the concentration field generated by the "InteractiveField" and migrate along the maximum concentration gradient.

This section of the tutorial is located in:

```
Tutorials/tutorialChemotaxis/
```

Include the header for the simple molecular dynamics simulation by inserting the following line in main.cpp

```
#include "../Tutorials/tutorialChemotaxis/chemotaxis.h"
```

3.2 More InteractiveField Functions

- To get the concentration $C(\mathbf{r},t)$ at a certain coordinate `x0, y0, z0`,the user should use:

  ```
  double concentration =  c -> getConcentration(x0, y0, z0);
  ```

- To get the gradient of the concentration $\nabla C(\mathbf{r},t)$ at a certain coordinate `x0, y0, z0`, the user should use:

  ```
  vec3 gradient = c -> getGradient(x0, y0, z0);
  ```

3.3 Building a Cross-interactive Function using "getPartList":

The function "getPartList" will use the coordinate of the particles in the "CellSpace" to acquire the gradient vector of the diffusion. The particles will follow this gradient to migrate towards the highest concentration point.

```
class Migration_tutorial : public getPartList {

      public:
            double threshold;
            double f;
        Migration_tutorial(vamde::CellSpace *_s, vamde::InteractiveField *
_c) : getPartList(_s) , c(_c){
                  threshold = 1 ;
                  f =  0.25;
            }
      private:
            vamde::InteractiveField  * c;

            void action(Particle *i){
                  double concentration =  c -> getConcentration(i->x[0],i-
>x[1],i->x[2]);
                  vec3 gradient = c -> getGradient(i->x[0],i->x[1],i-
>x[2]);
                  double dr[3];
                  double rr = 0;
                  for (int d=0; d<DIM; d++) {
                        dr[d] = gradient.r[d];
                        rr += dr[d] * dr[d];
                  }
                  double x =sqrt(rr);
                  double drhat[3];
                  for (int d=0; d<DIM; d++) {
                        drhat[d] = dr[d] / x;
                  }
                  if (x > threshold) {
                        for (int d=0; d<DIM; d++){
                              i->F[d] += f * drhat[d];
                        }
                  }
            }
};
```

- The function is called as:

```
Migration_tutorial mg0(s0, c0);
```

- It is used as:

```
mg0.apply();
```

3.3 Creating Sinks and Sources

In this example, we introduce a sink and a source in the concentration field. The source is simulated by constantly setting the concentration at the point where the source is located at a specific value. Here, we use the value of 10. The sink is simulated by constantly setting the concentration to -10). The two functions are placed in a “while” loop.

```
      double x_mid = (param->gridinfo->world.lo[0] + param->gridinfo-
>world.hi[0]) /2;

      double y_mid = (param->gridinfo->world.lo[1] + param->gridinfo-
>world.hi[1]) /2;

      double z_mid = (param->gridinfo->world.lo[2] + param->gridinfo-
>world.hi[2]) /2;

      c0->setConcentration(x_mid+0.4*x_mid,y_mid,z_mid,10 );

      c0->setConcentration(x_mid-0.4*x_mid,y_mid,z_mid,-10 );
```

3.4 Creating a Simulation

By combining everything that we discussed so far, we can create the following chemotaxis model:

```
void runSimulation(vamde::Parameters * param)  {

      vamde::TimeKeeper * tk;
      tk = new vamde::TimeKeeper(param);

      /// parameters:
      double delta_t = param-> delta_t;
      double end_step = param-> end_step;
      double & t = param->clock->t;
      int & step = param->clock->step;

      /// Initiate clock
      param->clock->set_step(0);

      vamde::CellSpace ** s;
      s = new vamde::CellSpace  * [param->n_cell];
      s[0] = new
vamde::CellSpace("Tutorials/tutorial3chemotaxis/input/cell0.input",param);
      s[0]->cfgwriter->set_atom_name( "Cs");

      /// Reset the unique ID of all particles
      for (int n=0; n<param->n_cell; n++) {
            s[n]->refreshCell();
      }

      vamde::InteractiveField ** c;
      c = new vamde::InteractiveField  * [param->n_cont];

      vamde::InteractiveField::Cinit continit;
      continit.readinput("Tutorials/tutorial3chemotaxis/input/cont0.input")
;
      c[0] = new vamde::InteractiveField(continit,param);
      c[0]->diffusivity = 0.1;

      double x_mid = (param->gridinfo->world.lo[0] + param->gridinfo-
>world.hi[0]) /2;
      double y_mid = (param->gridinfo->world.lo[1] + param->gridinfo-
>world.hi[1]) /2;
      double z_mid = (param->gridinfo->world.lo[2] + param->gridinfo-
>world.hi[2]) /2;

      c[0]->setConcentration(x_mid+0.4*x_mid,y_mid,z_mid, 10 );
      c[0]->setConcentration(x_mid-0.4*x_mid,y_mid,z_mid,-10 );
      c[0]->setAllGlobalBoundarySink();

      for (int n=0; n<param->n_cont; n++) {
            c[n]->cfgwriter->print();
      }

      LenardJones3 lenardjones(s,param->n_cell );
      ZeroForce3 zeroforce(s,param->n_cell);
      LeapFrog3 leapfrog(s,param->n_cell);
      Viscosity3 viscosity(s,param->n_cell);
```

```
      for (int n=0; n<param->n_cell; n++) {
            s[n]->deleteBorders();
            s[n]->cfgwriter->print();
      }

      while (step < end_step) {
            /// Print Runtime progress
            tk->print_progress(10);
            /// Advace the clock by one step
            param->clock->advance();
            /// LeapFrog step 1
            leapfrog.step(1);
            for (int n=0; n<param->n_cell; n++) {
                  /// delete ghost particles
                  s[n]->deleteBorders();
                  /// move particles to new cells after integration before
moving interprocessor
                  s[n]->move_particles_to_the_correct_cells();
                  /// move particles across processors.
                  s[n]->moveParticles();
                  /// copy new ghost particles
                  s[n]->copyParticles();
            }
            /// Set All forces to zero
            zeroforce.loopLocalParticles();

            /// Apply Lenard Jones Pair potential
            lenardjones.loopNeighbors();

            viscosity.loopLocalParticles();
            /// LeapFrog step 2
            leapfrog.step(2);

            /// print Cfgs
            for (int n=0; n<param->n_cell; n++) {
                  s[n]->cfgwriter->print();
            }
            for (int n=0; n<param->n_cont; n++) {
                  c[n]->copyParticles();
            }
            for (int n=0; n<param->n_cont; n++) {
                  c[n]->Iterate();
            }

            /// continuum
            c[0]->cfgwriter->print();
            c[0]->setConcentration(x_mid+0.4*x_mid,y_mid,z_mid,  10 );
            c[0]->setConcentration(x_mid-0.4*x_mid,y_mid,z_mid, -10 );

            Migration_tutorial mg0(s[0], c[0]);
            mg0.apply();
      }

}
```