EPCC-SS-2001-10

DPD: A Java implementation

Jo Hoffmann

Abstract

The following report will document the development of a Java program to simulate Dissipative Particle Dynamics (DPD). In the first section the physics behind DPD will be explained. This is followed by a short analysis of the structure of an existing Fortran implementation on which the Java code is partially based. The remainder of this document is devoted to explain the internals of the Java implementation.

Contents

List of Figures

1 Introduction

The aim of the project was to implement a new Dissipative Particle Dynamics simulation with Lees Edwards boundary conditions. Previous efforts on DPD simulations here at Edinburgh already produced a Fortran MPI code. But because it was lacking flexibility in use and ease of maintainability it was decided to produce a new code. The main requirement for the new code was to be parallelised with OpenMP as it would mainly be used on shared memory systems. Other advantages of using OpenMP are a slightly better maintainability of the sequential code but on the downside the loss of flexibility in later use of the code (shared memory systems only).

C, C++ and Fortran have traditionally been the languages of choice for programming tasks of this kind. We chose Java due to reasons of portability its OO features and in order to gain more experience on using it for scientific applications. On the downside one can argue that Java codes are slower than their C or Fortran counterparts but recent benchmarks show that the performance difference only lies around a factor of 2 to 4 [2] and with most recent advances in JIT compilers pushing this factor even further down. The overhead that JIT compilers introduce is not of paramount importance for most high performance computing applications either as the total execution time is usually much larger than the compilation time.

1.1 The basics of DPD

Dissipative Particle Dynamics (DPD) is similar to Molecular Dynamics (MD) in that a system is fully defined through the momenta and positions of all the particles in the fluid. The fact that MD deals with individual molecules makes it computationally so intense that it can only be used for small time scales and simulation spaces. As a result complex fluid phenomena and properties such as fluid flow and hydrodynamic behaviour are impossible to simulate on large scales. DPD on the other hand solves these problems by simulating larger lumps of fluid instead of individual molecules. It has first been presented by Hoogerbrugge and Koelman in 1985 [1].

The algorithm has 2 main steps.

The impulse step:

$$\vec{f_i}(t) = \sum_{j \neq i} (\vec{F^c}(\vec{r_{ij}}) + \vec{F^d}(\vec{r_{ij}}, \vec{v_{ij}}) + \vec{F^r}(\vec{r_{ij}})) \tag{1}$$

with $\vec{r_{ij}} = \vec{r_i} - \vec{r_j}$, where $\vec{r_i}$ is the position of particle i

and $\vec{v_{ij}} = \vec{v_i} - \vec{v_j}$, where $\vec{v_i}$ is the velocity of particle i

$\vec{F^c}$ is a conservative force

$\vec{F^d}$ is a dissipative force

$\vec{F^r}$ is a random force

And the propagation step:

$$\vec{r_i^t} = \vec{r_i^{t-1}} + \vec{f_i^t}\frac{\Delta t^2}{m_i} \tag{2}$$

where $\vec{r_i^t}$ is the position of particle i at time t.

From the equation we can see that we need to sum over all particle pairs ij with $j \neq i$ in order to get the total force acting on particle i. In order to conserve momentum we need to have

$\vec{f_{ij}} = -\vec{f_{ji}}$. The force $\vec{f_{ij}}$ will be zero for particles i and j separated by a distance $|\vec{r_{ij}}| > r_c$ where r_c is called the cut-off-distance.

1.2 The basics of Lees Edwards boundaries

Lees Edwards boundary conditions simulate a fluid under extreme shear conditions. Which means that in one direction, say y for instance, a special shear force is acting. In 1 a box A is shown with its images B and C in the yz planes. The particles in A have a random starting velocity with their root mean square equal to the mean thermal velocity. Furthermore all particles have an additional velocity in the y direction

$$\Delta v_y = v_d \left(\frac{x}{L} - \frac{1}{2} \right) \tag{3}$$

where x is the x-component of the position of the particle and L is the length of the simulation space (in the x direction in case xyz dimensions differ).

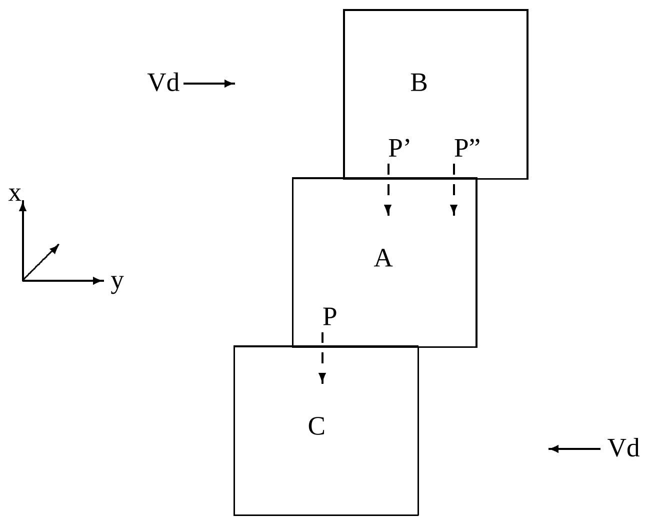


Figure 1: A model for Lees-Edwards boundary conditions

If a particle leaves the box at some point P at time $t = n_t \dot{\Delta} t$ it will be reintroduced at P'' instead of P' with its velocity $\vec{v'} = (v'_x, v'_y, v'_z)$ set to:

$$v'_x = v_x v'_y = v_y + v_d v'_z = v_z \tag{4}$$

The displacement in the y direction of the point P'' compared to P' is $n_t \dot{v}_d \dot{\Delta} t - \lfloor \frac{n_t \dot{v}_d \dot{\Delta} t}{L} \rfloor L$

2 The Java implementation

Even though Java is an Object Oriented language it is not necessary to stick to good OO practice to write a Java program. But as I am a principled person there was no other choice as to stick to the best principles I knew of. Initially, this meant drawing a model of the class hierarchy.

2.1 The Class model

While some of the detail in the classes changed over time the main features remained the same throughout the project.

2.1.1 The main classes

The two main components in the simulation are the particles and the space in which they flow around. Because I thought that they would be the more complex parts and would be most likely to change or be enhanced I decided to create a full hierarchy with interface, abstract and final classes.

The result for the simulation space is shown in figure 2. It shows an interface with methods calcForces() and calcVelPos() for doing the force calculation and the velocity and position update respectively. The other three methods are not useful in the way the space classes are used. Having a working simulation I would rather not have an interface for the space classes but just an abstract class. When I started off with the model I didn't quite know yet what would change between different boundary conditions and where it would be reflected in the code. Therefore my first approach was to have the calcForces() and calcVelPos() methods only implemented in the final classes, while the set up of a space would be coded into the abstract class. As it turns out the force calculation is exactly the same whereas the set up of the space differs from cyclic to Lees-Edwards boundary conditions. In the end all the functionality is implemented in the abstract class and the final classes only override some of the methods. For the CyclicSpace this is the velocity and position calculation and for the LE_Space this is also the method for creating the cut-off-boxes. Each final Space class has a main method that creates an instance of itself and starts the simulation.

The particle hierarchy is also setup with interface, abstract and final class. Unlike the space this setup seems more useful to me at the moment of writing this document. This is probably due to the fact that particles are extensively handled by other classes in the simulation and therefore need a proper interface. Figure 3 shows the Particle classes. The Particle interface defines a method to calculate the force between 2 particles (this and p), a method for the velocity and position update plus other methods to retrieve the force, velocity, position, type id and a type object. In order to stay flexible I introduced a class Type that contains all physical parameters of the different particles but more on that later. DPDParticle is then the abstract class that implements all methods in the interface except for the force velocity and position calculations. On top of those it also defines methods to add or set the force, velocity and position. The set methods are provided to allow greater flexibility in setting up a simulation while the add methods are mainly needed for the LE_Space. Monomer is then a complete implementation of a Particle. It implements the methods for the force, velocity and position calculations.

But beside Monomer there are also Dimers and Colloids. They are not particles in the sense used in this code. They are rather structures of Monomers where the force calculation is done on the individual Monomers, but the position and velocity update needs to look at the structure as a whole. They are therefore united under the CompoundParticle interface (see figure **??**). This interface only defines two methods: getParticles() to retrieve the individual particles (in this case Monomers) that make up the CompoundParticle and calcVelPos() for the velocity and position update. At the moment Dimer is the only implementation of the interface. Beside the

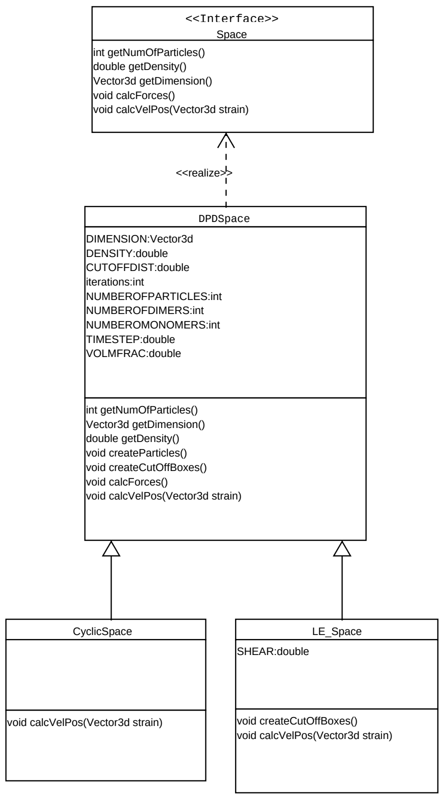


Figure 2: The simulation space hierarchy

methods from the interface it also implements methods addVelocity() and addPosition() that are again used in LE_Space.

A class used within Monomer and mentioned already earlier is Type (figure 3. It contains information about the nature of the particle. At the moment it has properties like mass and a numerical id to identify the particle types. It also contains a two dimensional static array to hold the specific forces that act between different particle types. All methods in this class are just concerned with retrieving the different parameters.

A class not so apparent at first is the CutOffBox. In order to reduce the complexity of the force calculation the space is split up in smaller boxes, the cut-off-boxes. Instead of then considering all particles for calculating the force on a particle one only needs to consider the particles in that cut-off-box and in its neighbouring boxes. The CutOffBox is also responsible for moving particles around. That means after the new position of the particle has been calculated the CutOffBox has to find out whether the particle moved out or not. When the particle moved out, the box has to identify the neighbour which it moved to and pass it on. Therefore a CutOffBox also needs to hold a list of its neighbours.

2.1.2 Helper classes

Beside the classes mentioned until now there are others that either provide additional facilities or that try to make life easier. One of the classes that makes life easier is Vector3d. As the name might suggest it deals with 3 dimensional Vectors. Methods include add, sub, dotProduct,

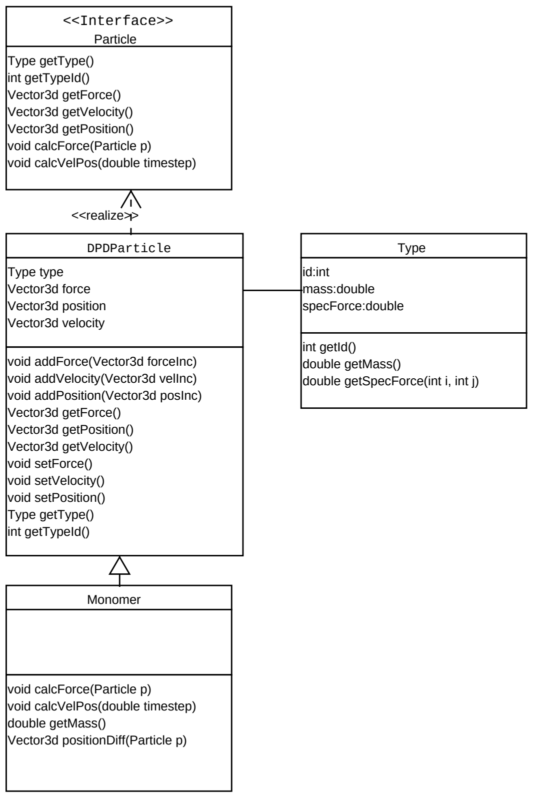
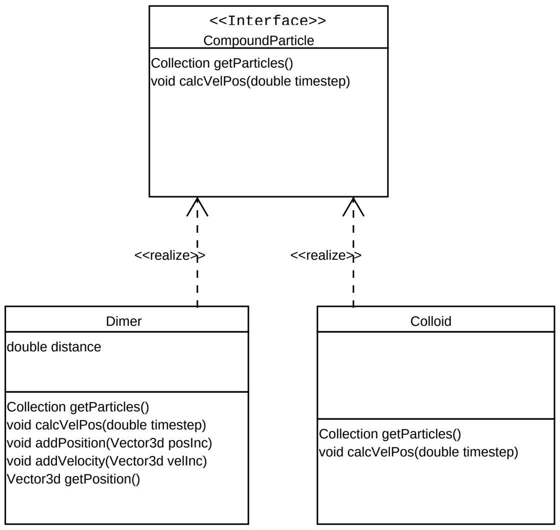


Figure 3: The particle hierarchy

scalarMul and others. All methods that would be natural candidates for returning a Vector3d object actually don't but instead modify *this* object. This is meant to increase performance by reducing the number of object creations. But this has obviously some negative effects due to more obscured programming and the fact that in certain circumstances the original vector is not meant to change. For instance calculating the difference of two velocities without changing the original vectors. In that case the original vector needs to be cloned prior to performing the operation. This then leads to more overhead as the clone method defined in java.lang.Object returns an Object and hence the object has to be downcast to a Vector3d.

```
Vector3d velocityDif = (Vector3d)velocity\_1.clone();\\
velocityDif.sub(velocity\_2);\\
```

Classes providing additional facilities are DPDInput and FluidStats. DPDInput is able to create and edit input files that contain the parameters to set up a simulation. Parameters include the number of particles, the density of the fluid, the number of iterations and so on. The methods deal mainly with the creation of an input file by asking for user input while explaining how the input needs to be formated and offering default values for most fields. A main method enables to start the process from the command line even though the provided methods can be used on their own. The file that DPDInput creates is a serialized instance of itself. This has the advantage of being easy to read in, as no parsing has to be done, or write to disk, as no file structure needs to be created. But this represents a big disadvantage as well. Any change in the instance variables (even changing a name) will render old input files unusable as the file descriptor changes.



<<Interface>>
CompoundParticle
Collection getParticles()
void calcVelPos(double timestep)
<<realize>>
<<realize>>
Dimer
double distance
Collection getParticles()
void calcVelPos(double timestep)
void addPosition(Vector3d posInc)
void addVelocity(Vector3d velInc)
Vector3d getPosition()
Colloid
Collection getParticles()
void calcVelPos(double timestep)

Figure 4: The CompoundParticle hierarchy

FluidStats is meant to provide the methods to do analysis on the overall state of the fluid. But at the moment it only provides 2 methods to test whether the simulation is stable and behaves properly. They are calcMonomerPhysics which calculates the average kinetic Energy of all Monomers in the fluid. And dimerTest which checks whether the dimers start aligning.

2.2 Class relationships

The UML diagram shown in figure 5 shows the complete class structure with their relationships. For instance DPDSpace contains 1 or more CutOffBoxes, 0 or more Particles and 0 or more CompoundParticles. This basically means that there needs to be a Collection for Monomers, Dimers, Colloids and CutOffBoxes.

A CutOffBox on the other hand contains 0 or more Particles but no CompoundParticles. In case of the Colloid the problem is trivial. A Colloid is so big that it never fits in a single CutOffBox. The dimers are a bit more tricky as they easily fit into a CutOffBox. But what happens if one of the Monomers is in one box but the other in a second one. Well at first I tried to calculate the position of the centre of mass and depending on it have the Dimer in one or the other CutOffBox. While that might work I still found that it would be easier to have the dimers only within the Space and have just their individual Monomers in the CutOffBoxes. Like this they can both be in different boxes. So in the end the CutOffBox needs basically only a Collection to hold the individual Monomers. But that means that the Monomers are once referenced to in the Space class and once in the CutOffBox.

Finally a Dimer has exactly 2 Monomers and a Colloid has 3 or more Monomers.

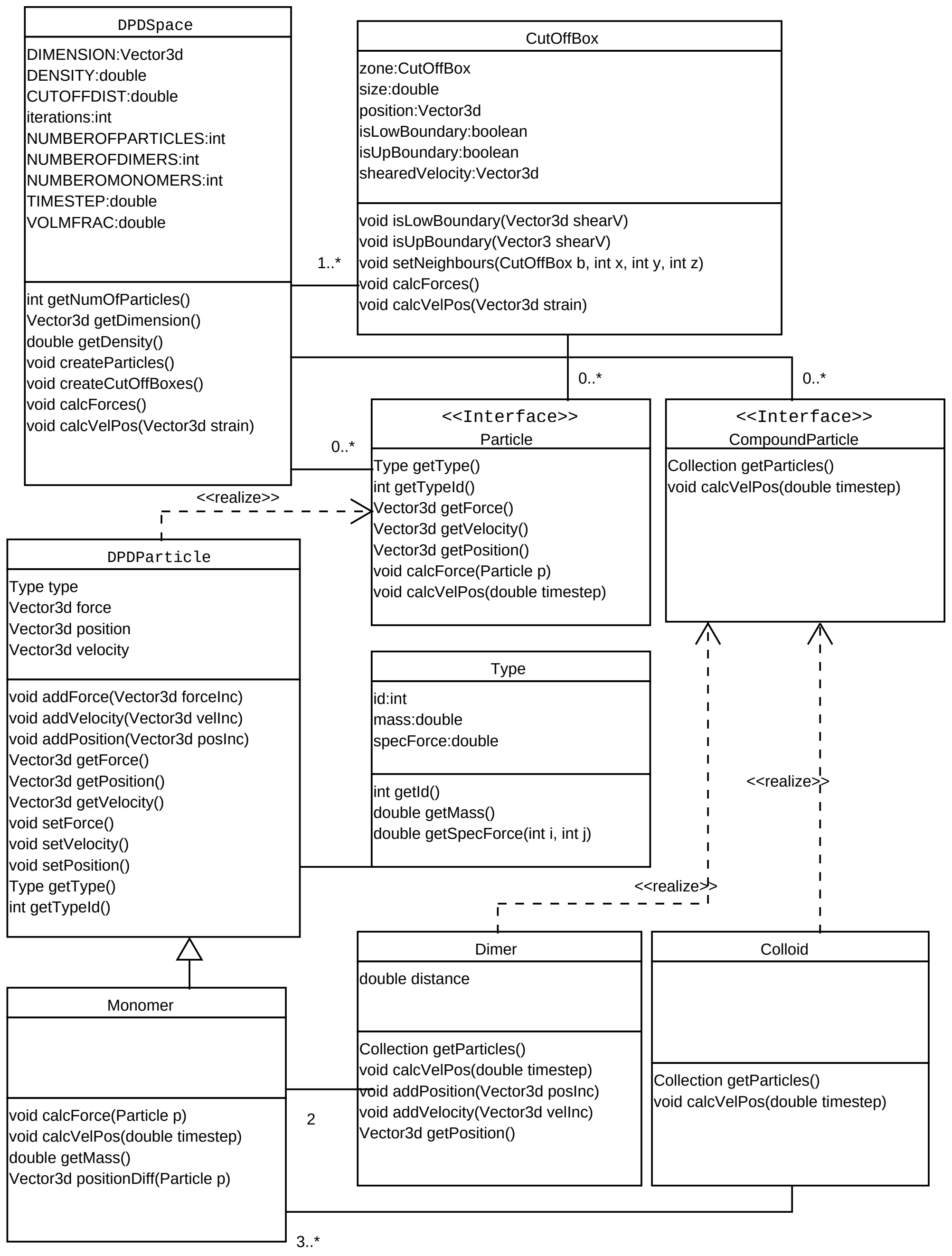


Figure 5: The particle hierarchy and their relationships

2.3 How it works together

The Space classes together with the input file control how the simulation is setup and takes place. The main method of a Space class creates an instance of itself by calling its constructor. The constructor reads in a DPDInput object and initialises its variables. Then it starts creating the CutOffBoxes and finally the particles. The particles are created according to the definitions in the DPDInput object. The DPDInput class has 2 instance methods createMonomer and createDimer to create the respective particles. Both methods need to be changed in order to create more complex setups. Each particle is then inserted into the particle collection of the Space class and in addition into the proper CutOffBox.

Then the main method goes on and executes a loop as many times as requested in the input file. Within the body of the loop the calcForces and calcVelPos methods are called on the Space object. These methods iterate over the CutOffBoxes and call their calcForces and calcVelPos methods. The CutOffBox methods simply iterate over the particles they contain and call their respective methods.

2.4 Limitations

A major limitation at the moment is the serialization of the DPDInput class. This is only a good solution if the probability of changing the data in the class is very low. But even the slightest change (adding a transient field for instance) will change the serialVersionUID and hence make all older input files useless. At this stage the Class is still very unstable. It is very likely that other parameters will be added or changed in order to make the setup of a simulation as flexible as possible without the need to change methods and recompile the code.

In addition because DPDInput is relying on Type any change to Type will have the same effect. Furthermore I had to introduce 2 other classes DimerProto and ColloidProto to store the basic information to create Dimers and Colloids. DimerProto only stores the Type of each of the Monomers it contains and the distance by which they are apart. The ColloidProto class is not yet implemented and hence implementing it will change the serialized object.

A cure to the problem could be to change the output file to either a text file or a binary file with a well defined format. The text file would have the advantage of editing it with any text editor.

Another problem is the use of static variables in DPDSpace and CutOffBox. As I took most of the algorithms from a Fortran code it seemed necessary to have global variables all over the place. So at first I made all variables in DPDSpace static. Now I cope with just CUTOFFDIST and DIMENSION to be static. They are used in classes like FluidStats, Monomer and DPDParticle. In CutOffBox some of the variables like size and sizeSquare are static as well. This is mainly for performance reasons as those values are extensively used during the force calculation.

2.5 To Be Done

Despite a lot of work their is still a lot of work left. Further tests are required to ensure correct behaviour. Initial results are included in section 6.

After that the first concern will probably be to parallelise the code with either OpenMP or Java Threads. As my experience with parallel applications is very limited I am not able to judge on how easy it will be to change the sequential code.

Another task is the development of a GUI. The first stage will be requirements gathering to see what kind of parameters the users want to control and what information they want to get out of it. This will also mean to introduce more useful methods into the FluidStats class to do the necessary calculations

And finally I think that most people would like to know how the code compares to the original Fortran code. But that would first of all mean to make some basic changes in the Fortran code.

3 Java, OO and the science

Writing Java programs is by most people considered to be good fun (and it certainly is by me). But I have to admit that at first I had a bit of a restraint towards Java and wasn't at all convinced by its usefulness before I actually wrote some code myself. The ease with which you can create and modularise complicated tasks is amazing. But the fact that so much of the complexity is hidden away brings some new difficulties mainly in the ability to judge performance.

3.1 Performance issues

One of the very nice features of Java is certainly the garbage collection. No need to worry about freeing memory it is all done for you. But to check for unused objects takes valuable time that could be spend on your computation. It becomes apparent that minimising the creation of objects and maximising their reuse will be highly beneficial to the performance of a program. It will be beneficial in 2 ways. First computation time for garbage collection is reduced and second less memory is unnecessarily wasted.

Iterators are a common way of accessing a Collection of data in OO programming. An iteration over a Vector might look like this:

```
Vector v;
.
.
Iterator iter = v.iterator();
while(iter.hasNext()){
 Object o = iter.next();
.
.
}
```

While that is perfectly alright the use of the hasNext method will slow the loop down. Using a for loop with the bounds of the iteration predefined will always be faster.

```
Vector v;
.
.
int vectorEnd = c.size();
```

```
for(int i=0; i<vectorEnd;i++){
 Object o = v.get(i);
.
.
}
```

An alternative, which gives slightly poorer performance than a for loop, but providing the ability to use Iterator features like remove(), is the following.

```
Vector v;
.
.
Iterator iter = v.iterator();
try{
 while(true){
  Object o = iter.next();
  .
  .
 }
} catch(NoSuchElementException e){}
```

Polymorphism will have an effect on performance as well. A Monomer can appear as a DPDParticle Particle and an Object. For instance inserting a Monomer into a Vector will upcast it to an Object. So requesting a Monomer from a vector will return an Object that can potentially be a Monomer but before it is it needs to be explicitly downcasted. This takes up some computational resources as the cast process needs to make sure that the Object is indeed a Monomer and if it isn't throw a ClassCastException.

Good OO style requires instance variables to be accessed through methods and never directly, as the methods can ensure that the data is not corrupted. A method call will however be more expensive than direct access as some stack operations take place.

All the potential problems I highlighted suggest that doing it the OO way is bad for performance. In HPC performance is important, hence a balance must exist between good OO style code readability and performance. Having listed all the potential drawbacks the advantages of OO designs will now be considered.

3.2 Benefits of using OO features

Imagine a scenario where the 3d vector representation needs to be changed (eg to improve performance). If instance variables had been addressed directly the whole code may require modification. However by using methods to access them only minimal modifications are required.

This is the benefit of encapsulation. The internal workings of the classes are hidden away. The user of a class does not need to know how the data is internally represented but instead a well defined interface provides access to the data. So later changes for whatever reasons won't affect the users of a class as long as the methods keep producing the results they are meant to.

Another plus point is inheritance. It might be that a new type of Monomer using different algorithms for force and velocity calculations needs to be introduced. In that case the new class will simply extend DPDParticle and implement the relevant methods but inherit all others including instance and class variables from DPDParticle. Testing the new class will be limited to the newly implemented methods as the inherited ones had been tested previously.

OO programming also promotes modularity. An application brakes down very naturally with OO techniques. Of course thought is necessary to setup the class model but once that is done programming becomes much easier. With a thoughtful design class interdependencies can be eliminated (or minimised) and make coding much easier as you don't necessarily need to have the details of other parts of the code in your head.

3.3 Conclusion

Modularity, code structure and reuse are the major advantages of OO programming. This will result in better maintainability and hence the possibility of extending an application easily with additional functionality. From my experience it seems easier to write inefficient OO codes than procedural programs. But by knowing the potential pitfalls it will be easy to work around them.

Earlier on I mentioned that even programming in Java doesn't mean that one has to stick to good OO programming style. I chose to stick to the best OO practice I know of and tried to minimise things like static variables and straight access to instance variables. This wasn't too easy as I took most of the algorithms from the Fortran code that was seeded with global variables. I also designed a class hierarchy for the simulation with interfaces, abstract and final classes. This can be overkill in some circumstances and might have an effect on the performance too. For instance the calcForce(Particle p) method is in the Particle interface defined to take a Particle as parameter. And so it the calcForce method for Monomer needs to accept a Particle too. But before starting this particle needs to be downcast to Monomer. Making the design less abstract could improve the performance (but I don't know how significant it is).

On the other hand it is also possible to write the whole code by squeezing it into a single source file. Disadvantages of this approach are difficult maintainability and difficulties to work on the code in a team.

My final verdict is for a code that should last many years, be extensible and maintainable a decent OO design will be necessary. For small one of codes that are not likely to be widely used a more relaxed approach can probably be taken. But who knows where a program ends up?

4 Appendix A: Complexity of the force calculation

In order to calculate the force acting on one particle all other particles need to be considered. This is the most basic conclusion from the algorithm. That means having n particles $n \cdot (n-1)$ iterations need to be performed to calculate all the forces. Luckily there are a number of improvements to this as the simulation preserves momentum:

$$f_{ij} = -f_{ji}$$

where f_{ij} is the force between particles i and j.
The amount of work simplifies to $\frac{n \cdot (n-1)}{2}$ the sum of the first $n-1$ numbers.

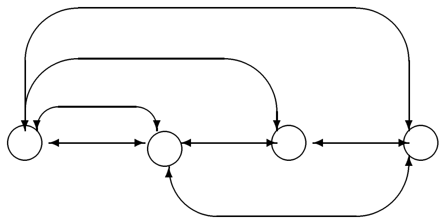


Figure 6: The forces that need to be calculated with 4 particles.

Another simplification is the use of cut-off-boxes so that only the particles that are potentially in reach need to be considered.

The volume of the simulation space is $V_s = \frac{N}{d}$ where N is the number of particles and d is the density.
The volume of a cut-off-box is $Vc = c^3$ where c is the cut-off-distance.
The number of cut-off-boxes is $N_c = \frac{V_s}{V_c}$
The number of particles per cut-off-box is on average:

$$N_{pc} = \frac{N}{N_c} = \frac{N \cdot V_c}{V_s} = d \cdot V_c$$

which is as expected equal to the density times the volume of the cut-off-box. In reality this is not very likely and I am sure that some random distribution will model this more accurately.

First the forces within the cut-off-box need to be calculated $\frac{d \cdot V_c \cdot (d \cdot V_c - 1)}{2}$
Then the forces in the surrounding boxes will be calculated. This is proportional to the square of the number of particles as the previous simplifications don't work in this case. $(d \cdot V_c)^2$. In total there are 26 neighbouring boxes but only 13 of them need to be considered because of the preservation of momentum. Hence the total work for one cut-off-box is:
$\frac{d \cdot V_c \cdot (d \cdot V_c - 1)}{2} + 13 \cdot (d \cdot V_c)^2$
That means for the whole space

$$\frac{V_s}{V_c}\left(\frac{d \cdot V_c \cdot (d \cdot V_c - 1)}{2} + 13 \cdot (d \cdot V_c)^2\right)$$

$$\frac{N}{d \cdot V_c}\left(\frac{d \cdot V_c \cdot (d \cdot V_c - 1)}{2} + 13 \cdot (d \cdot V_c)^2\right)$$

$$N \frac{27 \cdot d \cdot V_c - 1}{2}$$

5 Appendix B: Input parameters and algorithms

Input parameters are:

- cut-off-distance c_{dist}: particles separated by cut-off-distance won't interact with each other
- density d: the density of the fluid
- GAMMA $gamma$: Dissipative coefficient used in force calculation between Monomers
- SIGMA $sigma$: noise coefficient used in force calculation between Monomers
- iterations i: number of times the calculations should take place
- expected energy E_e: the energy that the fluid is expected to settle at, used for setup
- number of particles N: in the simulation space.
- shear $shear$: shear rate acting in Lees Edwards boundary conditions
- specForce F_{spec}: specific constant quantifying the interaction between different particle types
- timestep Δt: real amount of time between iterations (within the scope of the simulation)
- volumeFrac v_F: the volume fraction of Monomers being involved in dimers
- particle mass m: the mass of a single Monomer
- dimer distance d_D: the distance Monomers are separated in a Dimer.

5.1 Algorithms

- Volume of the Space $V_s = \frac{N}{d}$
- Dimension of the space:
  The space has the same length in each direction with the value being
  $dim_x = dim_y = dim_z = \sqrt[3]{V_s}$
- Number of cut-off-boxes $N_c = \lceil \frac{\sqrt[3]{V_s}}{c} \rceil$
- Number of Dimers $N_D = \lceil \frac{N \cdot v_F}{2} \rceil$
- Number of single Monomers $N_M = N - N_D$
- Shear velocity increment in y direction $\Delta v_y = shear \cdot dim_x$
- Shear position increment $\Delta r_y = \Delta v_y \cdot \Delta t$

5.1.1 Force calculation

The algorithm for creating the Gaussian Normal distribution:

```
public static double gaussianRandom(double d_sigma, double timeStep){
  final double A1 = 3.949846138;
  final double A3 = 0.252408784;
  final double A5 = 0.076542912;
  final double A7 = 0.008355968;
  final double A9 = 0.029899776;
  double sum =0.0;
  double result=0.0;

  for(int i=0;i<12;i++){
      sum += Math.random();
  }
  result = (sum-6.0)/4.0;
  double resultPow2 = result * result;
  result = ((((A9*resultPow2 + A7)*resultPow2 + A5)*resultPow2 +
      A3)*resultPow2 + A1)*result;
  result = d_sigma * result / Math.sqrt(timeStep);
  return result;
}
```

where d_sigma is $sigma$

Position of particle i $\overline{p_i}$
Position of particle j $\overline{p_j}$

Velocity of particle i $\overline{v_i}$
Velocity of particle j $\overline{v_j}$

Position difference between 2 particles $\overline{p_{ij}} = \overline{p_j} - \overline{p_i}$
Velocity difference between 2 particles $\overline{v_{ij}} = \overline{v_j} - \overline{v_i}$

Distance between 2 particles $r_{ij} = \sqrt{\overline{p_{ij}} \cdot \overline{p_{ij}}}$
$vdotr = \overline{p_{ij}} \cdot \overline{v_{ij}}$

$$F_d = -\frac{gamma \cdot (1/r_{ij} - 1/c_{dist})}{{r_{ij}}^2} \cdot vdotr$$

$$F_c = F_{spec}(ij) \cdot \frac{1}{r_{ij}} - \frac{1}{c_{dist}}$$

$$F_r = \Omega \cdot \frac{1}{r_{ij}} - \frac{1}{c_{dist}}$$

where F_d is the dissipative force, F_c is the conservative force and F_r the random force. Ω is a random variable with a Gaussian normal distribution ($N(0,1)$). The force increment on the particle is then $(F_d + F_c + F_r) \cdot \overline{p_{ij}}$.

5.1.2 Velocity and Position

The velocity change on particle i is $\Delta\overline{v_i} = \frac{\overline{f_i} \cdot \Delta t}{m}$
The position change on particle i is $\Delta\overline{r_i} = \overline{v_i} \cdot \Delta t$

6 Appendix C: Test results

6.1 Settings for graph energy_1

```
Density= 6.0
cut-off-distance= 1.0
Shear= 0.01
Number of particles= 1000
Number of iterations= 10000
GAMMA= 5.625
SIGMA= 0.0
Volume fraction of dimers= 0.7
Density= 6.0
Expected Energy= 25.0

Specific forces:
0       1       2       3       4
1       25.0    25.0    0.0     50.0
2       25.0    0.0     0.0     50.0
3       0.0     0.0     25.0    30.0
4       50.0    50.0    30.0    25.0

The basic types are:
Particle type 1 has mass 1.0
Particle type 2 has mass 1.0
Particle type 3 has mass 1.0
Particle type 4 has mass 1.0

The types reserved for Monomers are:
Particle type 1 has mass 1.0
Particle type 2 has mass 1.0

The dimer prototypes are
Dimer prototype definition:
Particle type 3 has mass 1.0
Particle type 4 has mass 1.0
The distance between the particles is: 0.5
```

6.2 Settings for graph energy_2

```
Density= 6.0
cut-off-distance= 1.0
Shear= 0.01
Number of particles= 1000
Number of iterations= 10000
GAMMA= 5.625
SIGMA= 1.0
```

```
Volume fraction of dimers= 0.7
Density= 6.0
Expected Energy= 25.0

Specific forces:
0       1       2       3       4
1       25.0    25.0    0.0     50.0
2       25.0    0.0     0.0     50.0
3       0.0     0.0     25.0    30.0
4       50.0    50.0    30.0    25.0

The basic types are:
Particle type 1 has mass 1.0
Particle type 2 has mass 1.0
Particle type 3 has mass 1.0
Particle type 4 has mass 1.0

The types reserved for Monomers are:
Particle type 1 has mass 1.0
Particle type 2 has mass 1.0

The dimer prototypes are
Dimer prototype definition:
Particle type 3 has mass 1.0
Particle type 4 has mass 1.0
The distance between the particles is: 0.5
```

The only difference between the 2 graphs is the SIGMA coefficient which defines the strength of the random component in the force calculation.

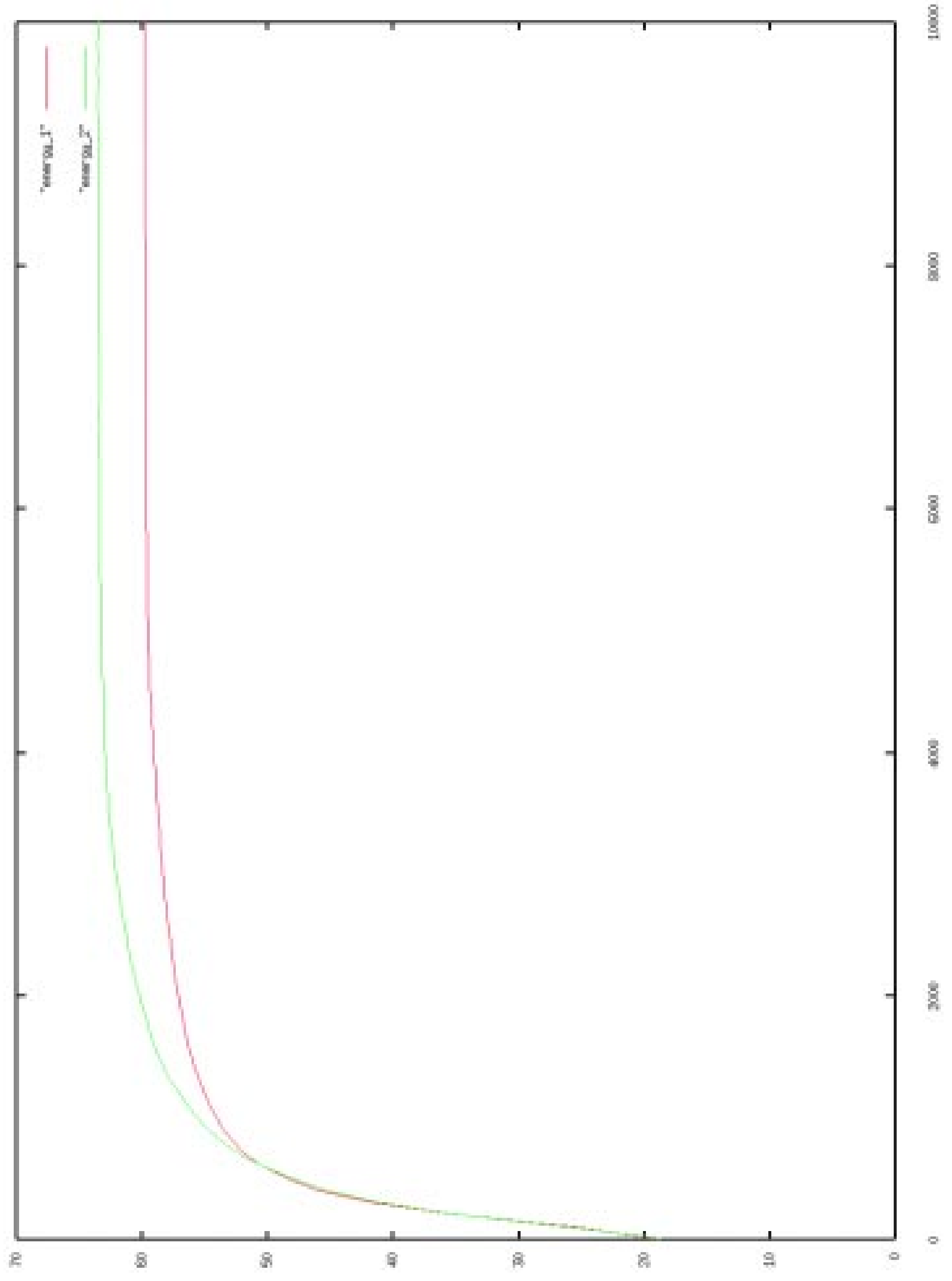


Figure 7: Kinetic energy of 1000 particles over 10000 iterations: energy_1 has SIGMA=0.0, energy_2 has SIGMA=1.0

References

[1] P. J. Hoogerbrugge and J. M. V. A. Koeleman. Simulating Microscopic Hydrodynamic Phenomena with Dissipative Particle Dynamics. Europhysics Letters, June 1992. 19 (3), pp. 155-160.

[2] L. Pottage J.M. Bull, L.A. Smith and R. Freeman. Benchmarking Java against C and Fortran for Scientific Applications. In Proceedings of ACM Java Grande/ISCOPE Confrence, June 2001.

I am currently studying Computer Science at Edinburgh University and will enter my third out of four years this October. My interests range from Computing to Photography including Meteorology and Climate science.

Many thanks to my two supervisors Lorna Smith (EPCC) and Alexander Wagner (Physics Department) who helped me stay focused throughout this project.