Molecular Dynamics of Water

Stochastic Modelling of Biological Processes

Candidate Number: 1060612

1 Introduction

In this paper, we discuss how to simulate N-body systems with distance constraints between the particles. First, we discuss how to find potentials and thus the force on each particle. Then we derive the numerical scheme Verlet velocity integration to solve kinematic problems. Additionally, we incorporate Quadtrees to make large particle systems more efficient to simulate. Also, we discuss different methods to implement fixed distance constraints and implement the RATTLE algorithm. Finally, we apply this to simulate a system of TIP3P water molecules.

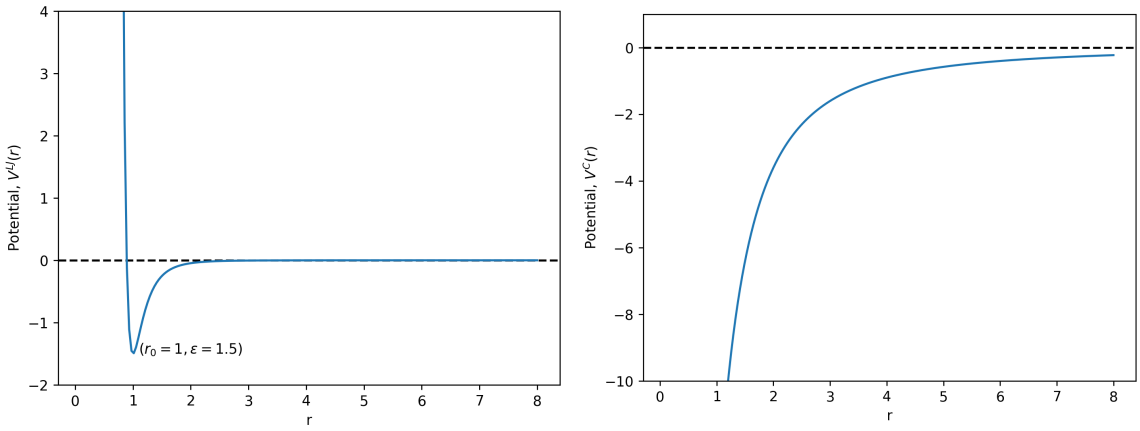
The simulation of the N-body system was done in C++ but the creation of graphs and plots was done in Python.

2 Potentials

For the numerical methods used in our simulations we need to calculate the force on each particle in the system. The force on the particle is defined as the negative of the derivative of the potential. In this paper, we will use the Lennard-Jones potential and the Coulomb potential. To calculate the potential on a particle the sum of the pairwise potential with all other particles is calculated. This leads to the equation of the force on particle i being

$$\boldsymbol{F}_i = -\frac{\partial}{\partial \boldsymbol{r}_i} \sum_{j \in S_i} V(||\boldsymbol{r}_{ij}||) = -\sum_{j \in S_i} \dot{V}(||\boldsymbol{r}_{ij}||) \frac{\boldsymbol{r}_{ij}}{||\boldsymbol{r}_{ij}||}. \tag{1}$$

With $S_i = \{1, 2, ..., \} \setminus \{i\}$ and $\boldsymbol{r}_{ij} = \boldsymbol{r}_i - \boldsymbol{r}_j$.



(a) $V^{(LJ)}$, with $r_0 = 1, \epsilon = 1.5$.

(b) $V^{(C)}$, with $k_e = 14.4, Q = 1, q = -1$.

Figure 1: Plots of Potentials.

2.1 Lennard-Jones Potential

We will denote the Lennard-Jones potential as $V^{(LJ)}$. Thus we have

$$V^{(LJ)}(r) = \epsilon \left[\left(\frac{r_0}{r} \right)^{12} - 2 \left(\frac{r_0}{r} \right)^{6} \right], \tag{2}$$

$$\dot{V}^{(LJ)}(r) = -\frac{12\epsilon}{r^2} \left[\left(\frac{r_0}{r} \right)^{11} - \left(\frac{r_0}{r} \right)^{5} \right]. \tag{3}$$

In Figure 1a we have the plot for $V^{(LJ)}$. We have the minimum point at (r_0, ϵ), at short distances when two particles have a distance less than r_0 they are repelled away from each other and when their distance is more than r_0 they are attracted to each other. For long distances, particles do not interact.

For sufficiently small ϵ the potential models the hard shell potential. Thus this can be used to approximate collisions. However, decreasing ϵ means our integration time step needs to be smaller to stop particles from shooting off.

2.2 Coulomb Potential

We will denote the Coulomb potential as $V^{(C)}$. Thus we have

$$V^{(C)}(r) = \frac{k_e Q q}{r}, \tag{4}$$

$$\dot{V}^{(C)}(r) = -\frac{k_e Q q}{r^2}. \tag{5}$$

From Figure 1b we can see long-distance particles have limited interaction.

2.3 Spring Potential

We have the spring potential as

$$V^{(S)}(x) = \frac{1}{2} k (x - \ell)^2, \tag{6}$$

$$\dot{V}^{(S)}(x) = k(x - \ell). \tag{7}$$

Where k is the spring constant and ℓ is the natural length of the spring. To make this a damped spring we can add a force directly proportional to the speed squared in the direction opposite to the movement of the spring.

3 Verlet Integration

If we know the force acting on the particles in a system we can simulate the system by using Verlet Integration. We use a derivation based on ideas from [1]. We want

to find $\boldsymbol{r}(t+h)$, this can be done by using Taylor expansion.

$$\boldsymbol{r}(t \pm h) = \boldsymbol{r}(t) \pm h\dot{\boldsymbol{r}}(t) + \frac{h^2}{2}\ddot{\boldsymbol{r}}(t) \pm \frac{h^3}{6}\dddot{\boldsymbol{r}}(t) + \mathcal{O}(h^4). \tag{8}$$

From $\boldsymbol{r}(t+h) + \boldsymbol{r}(t-h)$ after some algebraic manipulation we get

$$\boldsymbol{r}(t+h) = 2\boldsymbol{r}(t) - \boldsymbol{r}(t-h) + h^2\ddot{\boldsymbol{r}}(t) + \mathcal{O}(h^4). \tag{9}$$

From $\boldsymbol{r}(t+h) - \boldsymbol{r}(t-h)$ we can calculate velocity ($\dot{\boldsymbol{r}}$) which is

$$\dot{\boldsymbol{r}}(t) = \frac{\boldsymbol{r}(t+h) - \boldsymbol{r}(t-h)}{2h} + \mathcal{O}(h^2). \tag{10}$$

This algorithm has an error of $\mathcal{O}(h^4)$ when only the position is needed. However, when doing calculations with velocity the error is increased to $\mathcal{O}(h^2)$. An example calculation would be the kinetic energy which is $\frac{1}{2}m||\dot{\boldsymbol{r}}||^2$.

Therefore, we will use Verlet Velocity integration. And we define acceleration as $\ddot{\boldsymbol{r}}(t) = \boldsymbol{f}(t)$ thus we get

$$\boldsymbol{r}(t+h) = \boldsymbol{r}(t) + h\dot{\boldsymbol{r}}(t) + \frac{h^2}{2}\boldsymbol{f}(t) + \mathcal{O}(h^3), \tag{11}$$

$$\dot{\boldsymbol{r}}(t+h) = \dot{\boldsymbol{r}}(t) + h\boldsymbol{f}(t) + \frac{h^2}{2}\dddot{\boldsymbol{r}}(t) + \mathcal{O}(h^3). \tag{12}$$

But we need calculate the jerk term ($\dddot{\boldsymbol{r}}(t)$). This is retrieved from the Taylor expansion of $\boldsymbol{f}(t+h)$, thus for jerk we get

$$\dddot{\boldsymbol{r}}(t) = \frac{\boldsymbol{f}(t+h) - \boldsymbol{f}(t)}{h} + \mathcal{O}(h). \tag{13}$$

Now we substitute (13) into (12) to get the Verlet Velocity integration scheme.

$$\boldsymbol{r}(t+h) = \boldsymbol{r}(t) + h\dot{\boldsymbol{r}}(t) + \frac{h^2}{2}\boldsymbol{f}(t) + \mathcal{O}(h^3), \tag{14}$$

$$\dot{\boldsymbol{r}}(t+h) = \dot{\boldsymbol{r}}(t) + \frac{h}{2}(\boldsymbol{f}(t) + \boldsymbol{f}(t+h)) + \mathcal{O}(h^3). \tag{15}$$

And for concise notation (and to help us implement constraints) we introduce new variables $\boldsymbol{q}$ and $\boldsymbol{p}$. This gives us the Verlet integration scheme

$$\boldsymbol{q} = \dot{\boldsymbol{r}} + \frac{h}{2}\boldsymbol{f}, \tag{16}$$

$$\boldsymbol{r}^+ = \boldsymbol{r} + h\boldsymbol{q}, \tag{17}$$

$$\boldsymbol{p} = \boldsymbol{q} + \frac{h}{2}\boldsymbol{f}^+, \tag{18}$$

$$\dot{\boldsymbol{r}}^+ = \boldsymbol{p}. \tag{19}$$

Where $^+$ denotes the next step.

4 Boundary Conditions

4.1 Periodic Boundary Conditions

To simulate periodic boundary conditions we copy the positions of the particles in the square or cube and we paste those squares and cubes around the original square or cube at every time step. We only have to calculate the potentials for the particles in the centre square or cube. In 2D there are now 9 times the original particles and in 3D there are 27 times the original particles.

4.2 Walls

Here we discuss how to use the Lennard-Jones potential stated in (2) to approximate collisions against the boundary of a box. First, We calculate the distance of the particle from the closest wall (edge of the box). This is calculated by using the signed distance function of a cube.

```
double sdBox(vec3 p) {//Cude centered at origin with side length 1.
    vec3 q = abs(p) - vec3(0.5, 0.5, 0.5);
    return length(max(q, 0.0)) + min(max(q.x, max(q.y, q.z)), 0.0);}
double sdBox30(vec3 p){
    //Cube over [0,30]
    p = p-vec3(15., 15., 15.);
    p = p/30.;
    return sdBox(p)*30.;}
```

Other Signed distance functions can be found at [2]. To get the input parameter r for the potential function $V^{(LJ)}$ we do

$$r = -\text{sdBox30}(\boldsymbol{p}), \tag{20}$$

where $\boldsymbol{p}$ represents the position of the particle. Now we need to calculate the direction of the force. A solution is to take the gradient of the signed distance function to get the outward normal of the cube. This can be done with the following code

```
vec3 GetNormalBox30(vec3 p) //Gets normal of the surface
{
    vec3 GradF = vec3(0.0,0.0,0.0);
    double F = sdBox30(p);
    GradF.x = sdBox30(p+epsilon*vec3(1.,0.,0.));
    GradF.y = sdBox30(p+epsilon*vec3(0.,1.,0.));
    GradF.z = sdBox30(p+epsilon*vec3(0.,0.,1.));
    GradF = GradF-F;
```

```
    return normalize(GradF);
}
```

Thus we now have the inward normal. We calculate the force from the wall with this formula

```
Force = -dotV_LJ(-sdBox30(r), eps, r_0)*(-GetNormalBox30(r));
```

This method can also be used in 2D by changing the signed distance function to a signed distance function of a square.

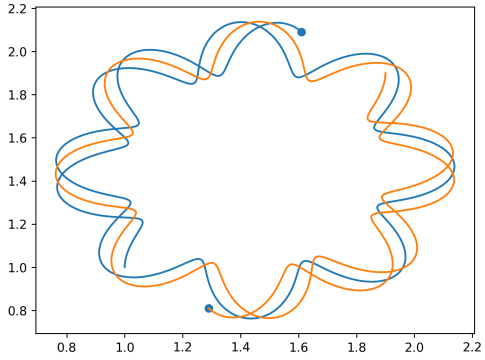
5 Small N-Body Simulations in 2D

Now we simulate some small systems where the particle interactions can be calculated by brute force and no spatial partitioning is needed. In the simulations shown in Figure 2, we use periodic boundary conditions and only consider the Lennard-Jones potential.

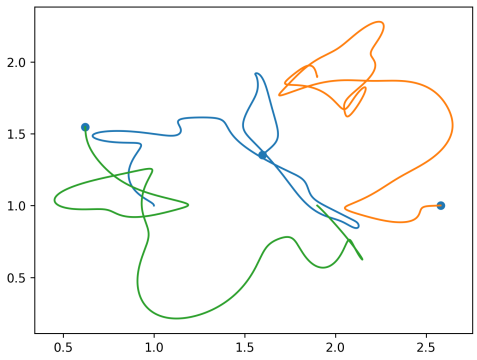
In Figure 2a we have a simulation of 2 particles. It has initial conditions $\boldsymbol{r}_0(0) = (1, 1), \boldsymbol{r}_1(0) = (1.9, 1.9), \dot{\boldsymbol{r}}_0(0) = (0, 0.5)$, and $\dot{\boldsymbol{r}}_1(0) = (0, -0.5)$ and parameters $\epsilon = 1$ and $r_0 = 1$. This has a periodic solution, the box size is 10 thus the particles are not impacted by the periodic boundary conditions. This simulation is a strong indication our code works.

In Figure 2b we have a simulation of 3 particles. It has the same conditions as the simulation shown in Figure 2a and $\boldsymbol{r}_2(0) = (1.9, 1.)$ $\dot{\boldsymbol{r}}_3(0) = (0, 0)$. From visual inspection it can be seen that adding an extra particle leads to chaotic motion.

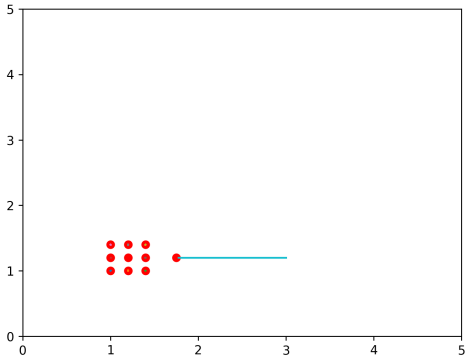
In Figures 2c, 2d, 2e and 2f we have a simulation of 10 particles. At $T = 0$ nine particles are at rest in the shape of a square and particle ten has initial conditions $\boldsymbol{r}_9(0) = (3, 1.2), \boldsymbol{r}_9(0) = (-5, 0)$. Other conditions we have are $\epsilon = 0.2, r_0 = 0.2$ and a box size of 5. This demonstrates a particle colliding into a group of particles and shows us our code can simulate periodic boundary conditions.



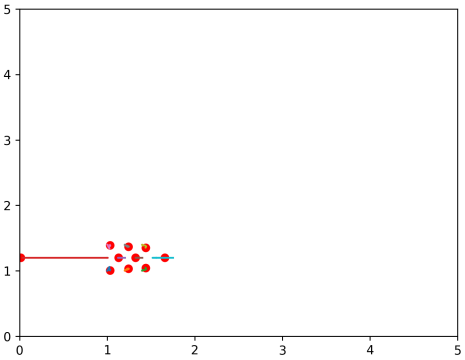
(a) $N = 2$.



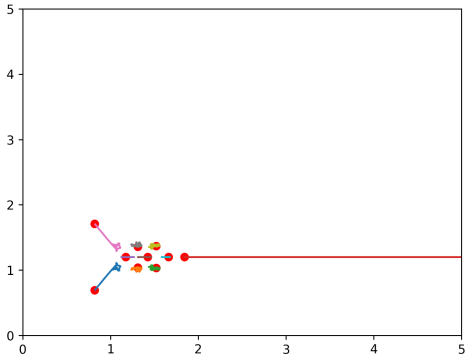
(b) $N = 3$.



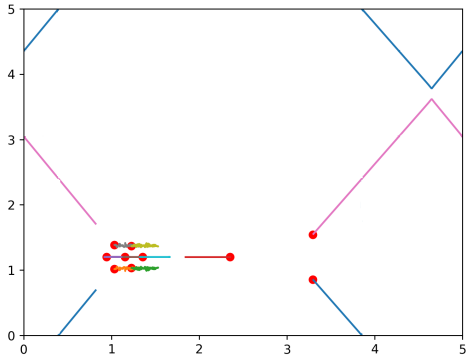
(c) $N = 10$. At $T = [0, 0.25]$.



(d) $N = 10$. At $T = [0.25, 0.53]$



(e) $N = 10$. At $T = [0.535, 1.5]$.



(f) $N = 10$. At $T = [1.5, 3.5]$

Figure 2: Small N-body simulations.

6 Space Partitioning With Quadtrees

We want to simulate a system of N particles. For the potential functions discussed in this paper, we need to calculate the pairwise potential of particles. There are $N(N-1)/2$ pairs in a N particle system. Therefore, calculating all pairwise potentials is of time complexity $\mathcal{O}(N^2)$. So for small systems, this works but for large systems, a better method is needed. This is also known as the N-body problem.

For our simulations, we will use the approach discussed in this article [3]. It explains how to calculate the gravitational force on a particle in an N-body system with time complexity $\mathcal{O}(N\log(N))$. The idea is to use a Quadtree in 2D and an Octtree in 3D to represent the position of particles. To find particles inside a circle we can query the tree to return all data points in a rectangle of side length equivalent to the diameter of the circle and centred at the circle's centre. Then we can easily check if the particle lies in the circle by taking its distance from the circle's centre.

Now we discuss the algorithms to insert data into a Quadtree and query the Quadtree for points inside a circle. Both these algorithms use recursion to complete their task. First, we show the variables that the Quadtree has and the global variables it needs.

```
//Global Vars
const int N = 600; //Number of particles
const int NPer = 9*N; //To include periodic boundary conditions
vec2 rPer[NPer]; //Stores position of particles
int rId[NPer]; //Stores Id of points inside given circle
int rIdSize; //States the number of points inside given circle
//Stores size of rId, gives fixed size. So don't have to add entries
//dynamically

struct Quad
{
    int Id; //Stores Id of point from rPer
    int PointSize; //Capacity is one
    bool hasSub;
    Rectangle d0; //boundary d \Omega
    Quad* NW; //Pointers to trees children
    Quad* NE;
    Quad* SE;
    Quad* SW;
//Functions for Quad ...
};
```

Now we show the algorithm for inserting a data point into the tree.

```
//Function Belongs to Quad
bool insert(int Id)
{
    if (!dO.containsPoint(rPer[Id]))
        return false; //Point not in this Quad
    //Add to tree
    if (PointSize == 0 && !hasSub) {
        this->Id = Id;
        PointSize += 1;
        return true;
    }
    //Check if not subdivided
    if (!hasSub) {
        sub_divide();
        //Remove current point in this node
        //And insert it into its children
        if (PointSize == 1) {
            PointSize -= 1;
            this->insert(this->Id);
        }
    }
    //Insert into children
    if (NW->insert(Id))
        return true;
    if (NE->insert(Id))
        return true;
    if (SE->insert(Id))
        return true;
    if (SW->insert(Id))
        return true;
    return false;
}
```

We have made our Quadtree to store a maximum of one point per square. Now we show the algorithm to query data from a Quadtree.

```
//Function Belongs to Quad
void FindInsideRec(Circle C)
{
    //Check if this quad intersects with rectangle made from C
    if (!RectOverlap(dO, C.GetRec()))
        return;
    //Check point in circle
```

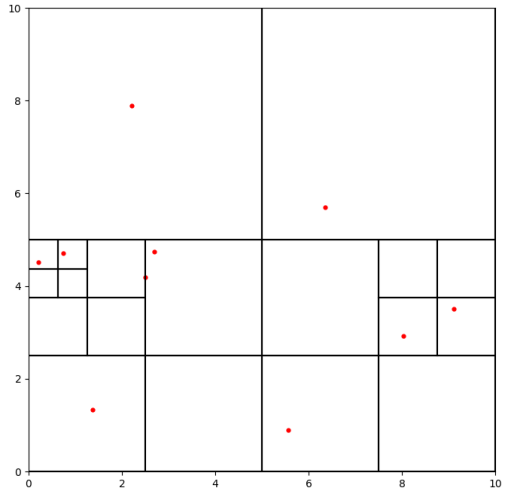
```
    if ((PointSize == 1) && (C.containsPoint(rPer[Id])))
    {
        rId[rIdSize] = Id;
        rIdSize += 1;
    }
    if (hasSub)
    {
        //Then apply to children
        NW->FindInsideRec(C);
        NE->FindInsideRec(C);
        SE->FindInsideRec(C);
        SW->FindInsideRec(C);
    } else { return; } //At end of tree no more points left
    //to check
}
void FindInside(Circle C)
{
    rIdSize = 0; //Clear the array to store points inside circle
    this->FindInsideRec(C);
}
```

In Figure 3 we show the Quadtree in action on a system of N uniformly randomly distributed particles. And the particles within the circle of radius 1.5 centred at $(3, 4)$ are highlighted in green.

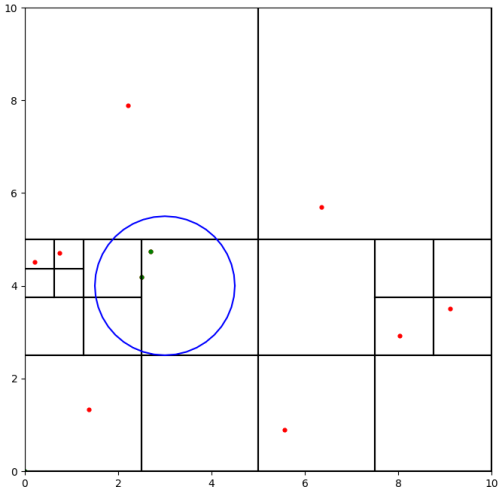
Another solution to the N-body problem is using hash tables. The idea with this is to split the space into squares or cubes and create a hash function to find which square a particle belongs in. Then when finding nearby points, the program looks up neighbouring squares and compares them with the particles inside.

These methods are not complete solutions to the N-body problem because they approximate the force on the particle. However, for short-range potentials like the Lennard-Jones potential, this approximation is good. In this paper, we use the coulomb potential to simulate water molecules and we do not simulate any systems with a particle that has a large absolute charge relative to the water molecules thus we only need to find pairwise potentials for nearby particles. To deal with long-range potentials we use the Barnes-Hut Algorithm stated in [3]. This algorithm works by grouping distant particles together and treating them as a single particle.

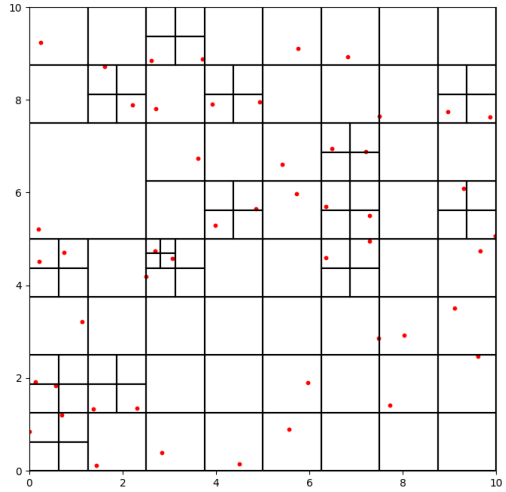
It is worth noting that these methods are only worth doing for sufficiently large systems as the Quadtree or hash function needs to be updated every time step.



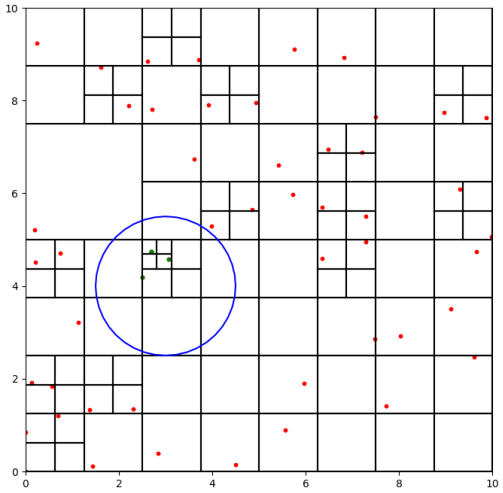
(a) $N = 10$.



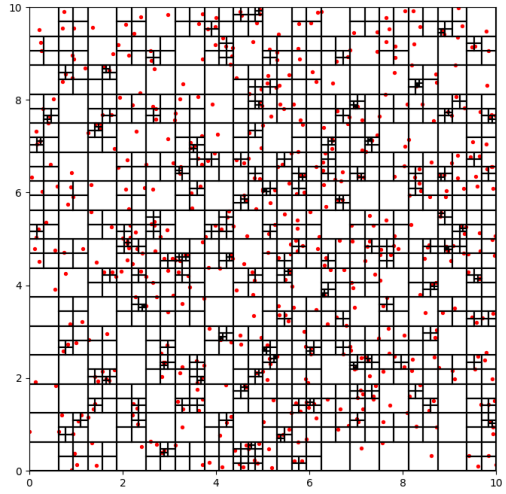
(b) $N = 10$. With Circle query.



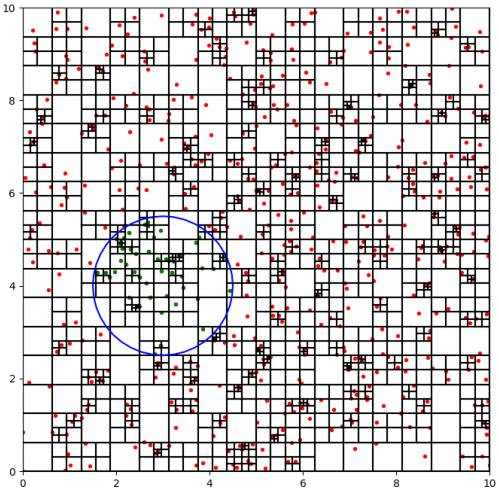
(c) $N = 50$.



(d) $N = 50$. With Circle query.



(e) $N = 600$.

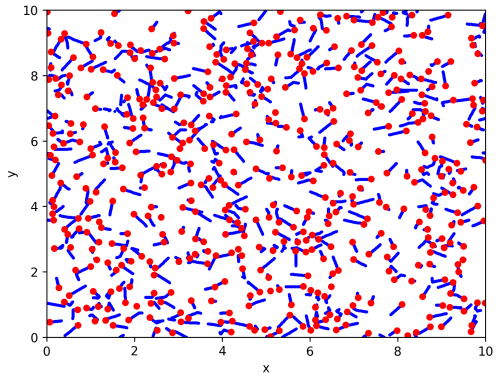


(f) $N = 600$. With Circle query.

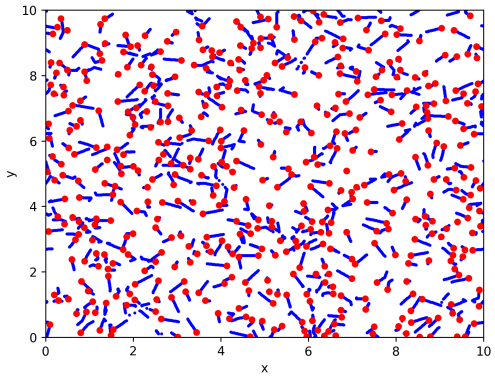
Figure 3: Visualisation of Quadtree for N uniformly randomly distributed particles.

7 Large N-Body Simulation

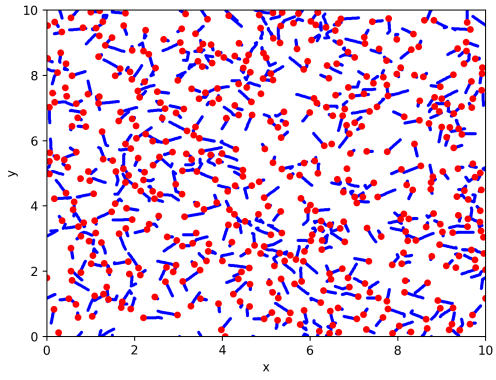
By using a Quadtree data structure at every time step we can simulate large N-body systems. In Figure 4 we simulate a system with $N = 500$ particles and periodic boundary conditions with Lennard-Jones potential. For initial conditions, we have the particles are uniformly randomly distributed and have random velocities. However, due to chaotic motion initial conditions can be considered irrelevant. Visual inspection strongly suggests our code is able to simulate 2D systems with many particles.



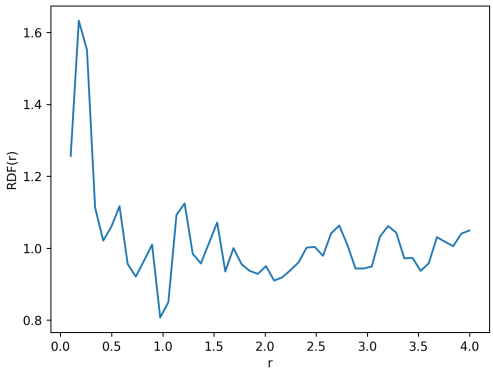
(a) $T = [0.65, 0.75]$.



(b) $T = [0.75, 0.85]$.



(c) $T = [0.85, 0.95]$.



(d) RDF at $T = 5$.

Figure 4: Simulation of an N-body system with periodic boundary conditions. $N = 500, \epsilon = 0.2$ and $r_0 = 0.2$.

In Figure 4d we have the radial distribution function. We notice the peak is located at a distance of $r \approx 0.2$. This suggests the particles like to be located at a distance of 0.2 away from each other. This makes sense as we are using the Lennard-Jones potential with $r_0 = 0.2$ thus it has a global minimum at $r = 0.2$. And particles

like to minimize their potential energy.

8 RATTLE Algorithm

The RATTLE algorithm is used to enforce constraints. This is done by using Lagrange multipliers and in this paper we will only discuss fixed distance constraints between particle i and particle j of the form

$$\sigma_{ij} = ||\boldsymbol{r}_i - \boldsymbol{r}_j||^2 - d_{ij}^2 = 0, \tag{21}$$

$$\dot{\sigma}_{ij} = 2(\dot{\boldsymbol{r}}_i - \dot{\boldsymbol{r}}_j) \cdot (\boldsymbol{r}_i - \boldsymbol{r}_j) = 0. \tag{22}$$

We start with the Verlet Velocity algorithm (14) and (15) then we implement new notation to get

$$\boldsymbol{r}_i^+ = \boldsymbol{r}_i + h\dot{\boldsymbol{r}}_i + \frac{h^2}{2}\boldsymbol{f}_i(\boldsymbol{r}_i), \tag{23}$$

$$\dot{\boldsymbol{r}}_i^+ = \dot{\boldsymbol{r}}_i + \frac{h}{2}(\boldsymbol{f}_i(\boldsymbol{r}_i) + \boldsymbol{f}_i(\boldsymbol{r}_i^+)), \tag{24}$$

where $\boldsymbol{r}_i = \boldsymbol{r}_i(t)$ and $\boldsymbol{r}_i^+ = \boldsymbol{r}_i(t+h)$. In other words the $^+$ means the next step. Now we implement constraints this can be done by defining the following variables

$$m_i\boldsymbol{f}_i = \boldsymbol{F}_i + \boldsymbol{G}_i, \tag{25}$$

$$\boldsymbol{F}_i = -\frac{\partial}{\partial \boldsymbol{r}_i}V(\boldsymbol{r}), \tag{26}$$

$$\boldsymbol{G}_i = -\frac{1}{2}\sum_{\alpha=1}^{n}\lambda_{i\alpha}\frac{\partial}{\partial \boldsymbol{r}_i}\sigma_{i\alpha} = -\sum_{\alpha=1}^{n}\lambda_{i\alpha}\boldsymbol{r}_{i\alpha}, \tag{27}$$

with $\boldsymbol{r}_{i\alpha} = \boldsymbol{r}_i - \boldsymbol{r}_\alpha$. The $\boldsymbol{F}_i$ is the potential when constraints are neglected. Additionally, we use

$$g_{ij} = h\lambda_{ij}, \tag{28}$$

$$k_{ij} = h\lambda_{ij}^+. \tag{29}$$

When we substitute (25) to (29) into (23) and (24) we get

$$\boldsymbol{r}_i^+ = \boldsymbol{r}_i + h\left[\dot{\boldsymbol{r}}_i + \frac{h}{2m_i}\boldsymbol{F}_i - \frac{1}{m_i}\sum_{\alpha=1}^{n}g_{i\alpha}\boldsymbol{r}_{i\alpha}\right], \tag{30}$$

$$\dot{\boldsymbol{r}}_i^+ = \left[\dot{\boldsymbol{r}}_i + \frac{h}{2m_i}\boldsymbol{F}_i - \frac{1}{m_i}\sum_{\alpha=1}^{n}g_{i\alpha}\boldsymbol{r}_{i\alpha}\right] + \frac{h}{2m_i}\boldsymbol{F}_i^+ - \frac{1}{m_i}\sum_{\alpha=1}^{n}k_{i\alpha}\boldsymbol{r}_{ij}^+. \tag{31}$$

Now defining $\boldsymbol{q}_i$ and $\boldsymbol{p}_i$ to be

$$\boldsymbol{q}_i = \dot{\boldsymbol{r}}_i + \frac{h}{2m_i}\boldsymbol{F}_i - \frac{1}{m_i}\sum_{\alpha=1}^{n} g_{i\alpha}\boldsymbol{r}_{i\alpha}, \tag{32}$$

$$\boldsymbol{p}_i = \boldsymbol{q}_i + \frac{h}{2m_i}\boldsymbol{F}_i^{+} - \frac{1}{m_i}\sum_{\alpha=1}^{n} k_{i\alpha}\boldsymbol{r}_{ij}^{+}. \tag{33}$$

Thus substituting (30) and (31) into (32) and (33) we get

$$\boldsymbol{r}_i^{+} = \boldsymbol{r}_i + h\boldsymbol{q}_i, \tag{34}$$

$$\dot{\boldsymbol{r}}_i^{+} = \boldsymbol{p}_i. \tag{35}$$

Now we must calculate the Lagrange multipliers g_{ij} and k_{ij}. We start solving for g and then for k, this is solved by using an iterative method and considering each constraint separately.

Solving for g. We start the iterative method with

$$\boldsymbol{q}_i = \dot{\boldsymbol{r}}_i + \frac{h}{2m_i}\boldsymbol{F}_i. \tag{36}$$

Now consider the distance constraint related to particle i and particle j. We have

$$\boldsymbol{s} = \boldsymbol{r}_i + h\boldsymbol{q}_i - \boldsymbol{r}_j - h\boldsymbol{q}_j. \tag{37}$$

Thus we check if $|\boldsymbol{s}\cdot\boldsymbol{s} - d_{ij}^2| < \epsilon$ where $\epsilon \ll 1$. If this is satisfied for all constraints we can move onto the velocity constraint section. If this is not satisfied we update our $\boldsymbol{q}_i$ and $\boldsymbol{q}_j$ with

$$\boldsymbol{q}_i^{+} = \boldsymbol{q}_i - \frac{g}{m_i}\boldsymbol{r}_{ij}, \tag{38}$$

$$\boldsymbol{q}_j^{+} = \boldsymbol{q}_j + \frac{g}{m_j}\boldsymbol{r}_{ij}, \tag{39}$$

$$g = \frac{\boldsymbol{s}\cdot\boldsymbol{s} - d_{ij}^2}{2h\boldsymbol{s}\cdot\boldsymbol{r}_{ij}(\frac{1}{m_i} + \frac{1}{m_j})}, \tag{40}$$

where the formulation of g is shown in section 8.1. Then we pick another constraint. We repeat this process until all constraints are satisfied. Now we discuss the velocity constraint. We start the iterative method with

$$\boldsymbol{p}_i = \boldsymbol{q}_i + \frac{h}{2m_i}\boldsymbol{F}_i^{+}. \tag{41}$$

Now we consider the velocity constraint between particle i and particle j. We have

$$\boldsymbol{p}_i^+ = \boldsymbol{p}_i - \frac{k}{m_i}\boldsymbol{r}_{ij}^+, \tag{42}$$

$$\boldsymbol{p}_j^+ = \boldsymbol{p}_j + \frac{k}{m_j}\boldsymbol{r}_{ij}^+, \tag{43}$$

$$k = \frac{(\boldsymbol{p}_i - \boldsymbol{p}_j)\cdot \boldsymbol{r}_{ij}^+}{d_{ij}^2(m_i^{-1} + m_j^{-1})}, \tag{44}$$

where the formulation of k is shown in section 8.2. Just like the previous section we repeat this for each constraint until the velocity constant

$$|(\dot{\boldsymbol{r}}_i^+ - \dot{\boldsymbol{r}}_j^+)\cdot(\boldsymbol{r}_i^+ - \boldsymbol{r}_j^+)| = \dot{\boldsymbol{r}}_{ij}^+ \cdot \boldsymbol{r}_{ij}^+ = 0, \tag{45}$$

is satisfied. This paper [4] goes into more detail about the RATTLE algorithm including its global error. Below we have a code extract from the code file "main.cpp" shown in appendix D. It shows the main stages of using Verlet velocity integration with RATTLE to simulate a constrained system.

```
int main () {
    //Make uniform random Box
    Initialise_Atoms();
    Save_r_ToFile();
    //Time Simulation
    StopWatch My_Watch;
    My_Watch.Start();
    std::cerr << r[0] << r[1] << r[2] << "Start \n";
    for(int step = 0; step < steps; ++step) {
        //Calc q
        Calc_Position();
        //Position Correction
        // For every Molecule
        Position_Correction();
        //Update r
        for(int index = 0; index<3*N; ++index)
        {
            r[index] = r[index] + h*q[index];
        }
        //Calc p and store in dr
        Calc_Velocity();
        //Velocity Correction
        Velocity_Correction();
        //Saved to dr
        //Save position to file
```

```
        Save_r_ToFile();
        std::cerr << "\rSteps remaining: " << steps - step << ' '
                  << std::flush;
    }
    //Completed time step
    My_Watch.Stop();
    std::cerr << "\n End, Time: " << My_Watch.Time() << "\n";
    return 0;
}
```

8.1 Derivation of g

We want to pick g such that

$$|\boldsymbol{s}^+ \cdot \boldsymbol{s}^+ - d_{ij}^2| \approx 0. \tag{46}$$

Therefore,

$$d_{ij}^2 = \boldsymbol{s}^+ \cdot \boldsymbol{s}^+ = ||\boldsymbol{r}_{ij}^+||^2, \tag{47}$$
$$d_{ij}^2 = ||\boldsymbol{r}_{ij} + h(\boldsymbol{q}_i^+ - \boldsymbol{q}_j^+)||^2, \tag{48}$$
$$d_{ij}^2 = ||\boldsymbol{r}_{ij} + h(\boldsymbol{q}_i - \boldsymbol{q}_j) - hg\boldsymbol{r}_{ij}(m_i^{-1} + m_j^{-1})||^2, \tag{49}$$
$$d_{ij}^2 = ||\boldsymbol{s} - hg\boldsymbol{r}_{ij}(m_i^{-1} + m_j^{-1})||^2, \tag{50}$$
$$d_{ij}^2 = \boldsymbol{s} \cdot \boldsymbol{s} - 2hg\boldsymbol{s} \cdot \boldsymbol{r}_{ij}(m_i^{-1} + m_j^{-1}). \tag{51}$$

Where we have used the fact that $|g| < 1$ thus g^2 can be neglected. Thus after rearranging we get the equation (40).

8.2 Derivation of k

We want to pick k such that the constraint $\dot{\boldsymbol{r}}_{ij}^+ \cdot \boldsymbol{r}_{ij}^+ = 0$ is satisfied. And from (35) we have $\dot{\boldsymbol{r}}_i^+ = \boldsymbol{p}_i$. Thus

$$\dot{\boldsymbol{r}}_{ij}^+ \cdot \boldsymbol{r}_{ij}^+ = \boldsymbol{p}_{ij}^+ \cdot \boldsymbol{r}_{ij}^+ = (\boldsymbol{p}_{ij} - k\boldsymbol{r}_{ij}^+(m_i^{-1} + m_j^{-1})) \cdot \boldsymbol{r}_{ij}^+. \tag{52}$$

And when we substitute $d_{ij}^2 = \boldsymbol{r}_{ij}^+ \cdot \boldsymbol{r}_{ij}^+$ and rearange for k we get (44).

8.3 Fixed Distance with Springs

As a side note, we can replace all the fixed distances with damped springs using the potential in (6). Then the RATTLE algorithm is not required as the damped spring

potential will enforce the fixed distance. The larger the spring constant k the more the spring acts as a fixed distance constraint. But when k gets larger the system gets more difficult to simulate.

9 Water Models

In this paper, we simulate water molecules using the modified TIP3P model. This involves each water molecule having 3 sites (2 Hydrogen Atoms and 1 Oxygen atom) in the configuration "H-O-H". From this paper [5] we get the assumed properties of water molecules using the modified TIP3P model.

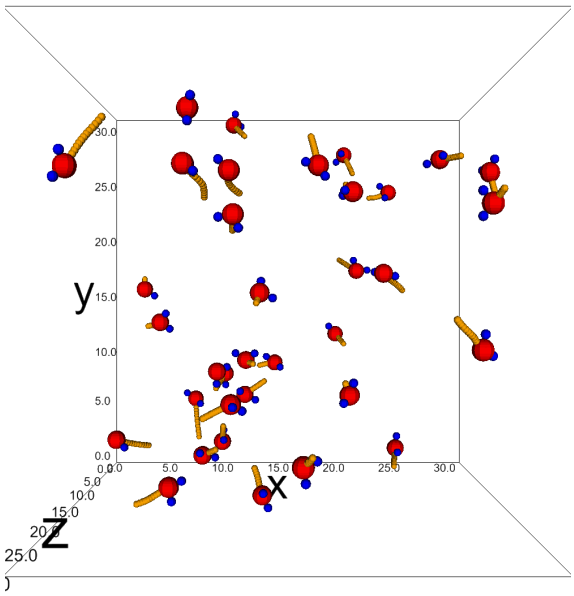
```
//Constants
const double k_e = 14.3996;
const double cut_off = 12.;
const double tol = 0.0005;
const double Mass_Oxygen = 16.; const double m_O = Mass_Oxygen;
const double Mass_Hydrogen =1.0; const double m_H = Mass_Hydrogen;
const double q_O = -0.834;
const double q_H = 0.417;
const double d_OH = 0.9572;
const double d_HH = 1.5139;
const double r_0_OO = 3.5365;
const double r_0_OH = 1.993;
const double r_0_HH = 0.449;
const double eps_OO = 0.1521;
const double eps_OH = 0.084;
const double eps_HH = 0.046;
```

9.1 Simulation Of Water Model

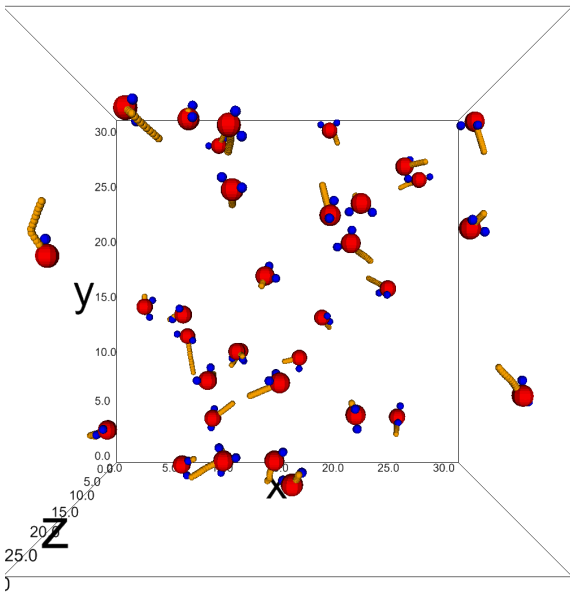
To simulate this we use Verlet velocity integration to solve the kinematic equations for the particles. Then we apply the RATTLE algorithm to enforce the fixed distance between the atoms in the water molecule. In Figure 5 we have a simulation of $N = 35$ water molecules with time step $h = 0.02$. Due to time constraints, a hash or Octtree was not implemented into the water molecule simulation. Thus we are restricted to time complexity $\mathcal{O}(N^2)$. Additionally, periodic boundary conditions were not used as this would increase the number of molecules. Therefore, we implemented the wall method described in section 4.2 and from the left side of Figure 5b we can see a water molecule bouncing off a wall.

In Figure 5d we have the radial distribution function between oxygen atoms. The RDF is calculated at the last time step of the simulation. From inspection the graph is not as smooth as Figure 4 in paper [5], this is because there are not enough water molecules in our simulation. However, visual inspection of Figure 5d proves the findings in [5] because both graphs have similar shapes. For example, the peak is located at $(3, 3)$ in Figure 5d and the peak is located at approximately $(3, 2.8)$ in the paper.

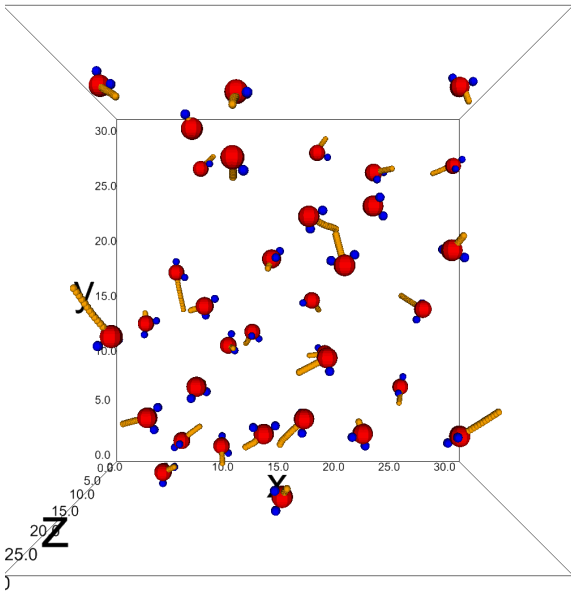
In Figure 5 the orange spheres represent the path the Oxygen molecules have taken.



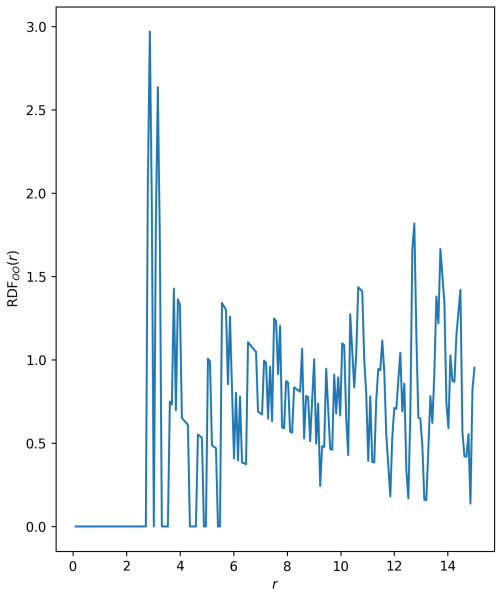
(a) $T = [4.56, 5.12]$.



(b) $T = [5.5, 6.06]$.



(c) $T = [6.44, 7]$.



(d) RDF$_{OO}$ at last time step $T = 10$.

Figure 5: Simulation of $N = 35$ water molecules inside a cube $[0, 30]$. Where boundary conditions cause molecules to bounce of walls.

10 Conclusion

In this paper, we derived numerical methods for simulating N-body systems with distance constraints. Then we implemented the Verlet velocity integration with Quadtree and periodic boundary conditions to solve N-body problems in 2D. Also, we incorporated the RATTLE and Verlet velocity integration algorithm in 3D to simulate basic water models.

From these results, we have proved the findings from paper [5] for the modified TIP3P model. Also, we have found efficient ways to calculate short-range potentials in time complexity $\mathcal{O}(N \log(N))$ by using Quadtrees. Additionally, all the C++ and Python code that is shown in this paper was written by myself. However, the code "MathObjects.hpp", "MathFunctions.cpp" and the signed distance function for a box from section 4.2 were written for my C++ special topic.

Further research and development would be focused on using space partitioning for modelling water molecules in 3D as this will improve the time complexity and thus more molecules can be simulated. And finding a way to incorporate a speed limit for the water molecules, as at high speeds time steps need to be smaller to stop particles from overlapping thus causing the derivative of Lennard-Jones potential to be large and causing a particle to shoot to infinity.

Also, analysis of other water molecules like the SPC/E model and the SPC model. Additionally, the current simulation only works for short-range potentials as we only compare to nearby particles, incorporating the Barnes Hut algorithm [3] would allow for water simulations that have long-range potentials.

References

[1] James Schloss. *Verlet Integration*. Last accessed 06.07.2022. URL: `https://www.algorithm-archive.org/contents/verlet_integration/verlet_integration.html`.

[2] Inigo Quilez. *3D SDF*. Last accessed 06.07.2022. URL: `https://iquilezles.org/articles/distfunctions/`.

[3] TOM VENTIMIGLIA and KEVIN WAYNE. *The Barnes-Hut Algorithm*. Last accessed 06.07.2022. URL: `http://arborjs.org/docs/barnes-hut`.

[4] Hans C Andersen. “Rattle: A “velocity” version of the shake algorithm for molecular dynamics calculations”. In: *Journal of Computational Physics* 52.1 (1983), pp. 24–34. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(83)90014-1. URL: https://www.sciencedirect.com/science/article/pii/0021999183900141.

[5] Pekka Mark and Lennart Nilsson. “Structure and Dynamics of the TIP3P, SPC, and SPC/E Water Models at 298 K”. In: *The Journal of Physical Chemistry A* 105.43 (2001), pp. 9954–9960. DOI: 10.1021/jp003020w. eprint: https://doi.org/10.1021/jp003020w. URL: https://doi.org/10.1021/jp003020w.

A Code File: “README.txt”

```
To compile the 2D N-body simulation run
gcc main2DsimNoBonds.cpp MathFunctions.cpp -lstdc++ -lm -o main2D
./main2D >> Results2D.txt

But for faster execution time N must be decreased.
Then to see the visulization of run the python code Vis2D.py

To compile the 3D N-body simulation of water molecules run
gcc main.cpp MathFunctions.cpp -lstdc++ -lm -o main3D
./main3D >> Results3D.txt

Then to see the visulization of run the python code Vis3D.py.
However, ipyvolume is needed.


To display a picture of a Quadtree run
gcc ShowQuadTree.cpp MathFunctions.cpp -lstdc++ -lm -I/usr/include/
   python3.8 -lpython3.8 -o tree
./tree

The output graph will be saved as "MyGraph.png"
But if this does not work the output picture is shown in the
Quadtree section.
```

B Code File: “Quadtree.hpp”

```
int rId[NPer];
int rIdSize;
```

```
3
4  struct Rectangle
5  {
6      vec2 BL;
7      vec2 TR;
8
9      Rectangle(vec2 BL, vec2 TR)
10     {
11         this->BL = BL;
12         this->TR = TR;
13     }
14
15     Rectangle()
16     {
17         BL = vec2(0.);
18         TR = vec2(1.);
19     }
20
21
22     bool containsPoint(vec2 p)
23     {
24         return ((BL.x<=p.x) && (p.x <= TR.x))
25                 && ((BL.y<=p.y) && (p.y<=TR.y));
26     }
27
28 };
29
30 struct Circle
31 {
32     vec2 mid;
33     double r;
34
35     Circle(vec2 mid, double r)
36     {
37         this->mid = mid;
38         this->r = r;
39     }
40
41     Circle()
42     {
43         mid = vec2(0.);
44         r = 1.;
45     }
```

```
46
47      Rectangle GetRec()
48      {
49          return Rectangle(mid-r,mid+r);
50      }
51
52      bool containsPoint(vec2 p)
53      {
54          return (length(p-mid)<r);
55      }
56  };
57
58  Rectangle dO_NW(Rectangle dO)
59  {
60      vec2 BL = vec2(dO.BL.x, (dO.BL.y+dO.TR.y)/2.);
61      vec2 TR = vec2((dO.BL.x+dO.TR.x)/2., dO.TR.y);
62      return Rectangle(BL, TR);
63  }
64
65  Rectangle dO_NE(Rectangle dO)
66  {
67      vec2 BL = (dO.BL+dO.TR)/2.;
68      vec2 TR = dO.TR;
69      return Rectangle(BL, TR);
70  }
71
72  Rectangle dO_SE(Rectangle dO)
73  {
74      vec2 BL = vec2((dO.BL.x+dO.TR.x)/2.,dO.BL.y);
75      vec2 TR = vec2(dO.TR.x,(dO.BL.y+dO.TR.y)/2.);
76      return Rectangle(BL, TR);
77  }
78
79  Rectangle dO_SW(Rectangle dO)
80  {
81      vec2 BL = dO.BL;
82      vec2 TR = (dO.BL+dO.TR)/2.;
83      return Rectangle(BL, TR);
84  }
85
86  bool RectOverlap(Rectangle R1, Rectangle R2)
87  {
88      return !(R1.BL.x>R2.TR.x ||
```

```
89              R2.BL.x>R1.TR.x ||
90              R1.BL.y>R2.TR.y ||
91              R2.BL.y>R1.TR.y);
92 }
93
94 struct Quad
95 {
96     int Id; //Stores Id of point from rPer
97     int PointSize; //Capacity is one
98     bool hasSub;
99     Rectangle d0; //boundary d \Omega
100    Quad* NW; //Pointers to trees children
101    Quad* NE;
102    Quad* SE;
103    Quad* SW;
104
105    Quad(Rectangle boundary = Rectangle())
106    {
107        d0 = boundary;
108        PointSize = 0;
109        hasSub = false;
110    }
111
112    Quad* getNewQuad(Rectangle boundary)
113    {
114        Quad* newQuad = new Quad(boundary); //Need to delete at some
115        // point??
116        return newQuad;
117    }
118
119    void sub_divide()
120    {
121        NW = getNewQuad(d0_NW(d0));
122        NE = getNewQuad(d0_NE(d0));
123        SE = getNewQuad(d0_SE(d0));
124        SW = getNewQuad(d0_SW(d0));
125        hasSub = true;
126    }
127
128    void FindInsideRec(Circle C)
129    {
130        //Check if this quad intersects with rectangle made from C
131        if (!RectOverlap(d0, C.GetRec()))
```

```
132             return;
133         //Check point in circle
134 
135         if ((PointSize == 1) && (C.containsPoint(rPer[Id])))
136         {
137             rId[rIdSize] = Id;
138             rIdSize += 1;
139         }
140         if (hasSub)
141         {
142             //Then apply to children
143             NW->FindInsideRec(C);
144             NE->FindInsideRec(C);
145             SE->FindInsideRec(C);
146             SW->FindInsideRec(C);
147         } else { return; } //At end of tree no more points left
148         //to check
149     }
150     void FindInside(Circle C)
151     {
152         rIdSize = 0; //Clear the array to store points inside circle
153         this->FindInsideRec(C);
154     }
155 
156     void QuadRemove()
157     {
158         //Deletes all subtrees
159         if (!hasSub)
160         {
161             return;
162         } else {
163         //Remove all children
164         NW->QuadRemove();
165         NE->QuadRemove();
166         SE->QuadRemove();
167         SW->QuadRemove();
168 
169         //Delete all children
170         delete NW;
171         delete NE;
172         delete SE;
173         delete SW;
174 
```

```
175
176
177         hasSub = false;}
178
179     }
180
181     bool insert(int Id)
182     {
183         if (!d0.containsPoint(rPer[Id]))
184             return false; //Point not in this Quad
185         //Add to tree
186         if (PointSize == 0 && !hasSub) {
187             this->Id = Id;
188             PointSize += 1;
189             return true;
190         }
191         //Check if not subdivided
192         if (!hasSub) {
193             sub_divide();
194             //Remove current point in this node
195             //And insert it into its children
196             if (PointSize == 1) {
197                 PointSize -= 1;
198                 this->insert(this->Id);
199             }
200         }
201         //Insert into children
202         if (NW->insert(Id))
203             return true;
204         if (NE->insert(Id))
205             return true;
206         if (SE->insert(Id))
207             return true;
208         if (SW->insert(Id))
209             return true;
210         return false;
211     }
212 };
```

C Code File: "main2DsimNoBonds.cpp"

```
1 #include <iostream>
2 #include <string>
```

```
3  #include <cassert>
4  #include <random>
5  #include "MathObjects.hpp"
6
7
8  //Constants
9  const double k_e = 14.3996;
10 const double cut_off = 5.;
11 const double tol = 0.0005;
12 const int seed = 1;
13
14 const double h = 0.005;
15 const double steps = 500;
16
17 //#define vec3 vec2
18
19
20 //Number of molecules with one particle
21 const int N = 200;
22 const int NPer = N*9;
23 const double BoxSize = 10;
24 vec2 r[N];
25 vec2 dr[N];
26 vec2 q[N];
27
28 //Periodic vars
29 vec2 rPer[NPer];
30 vec2 V2Add[9] = {vec2(0.,0.), vec2(-1., 0.), vec2(-1.,1.),
31                  vec2(0.,1.),  vec2(1., 1.), vec2(1.,0.),
32                  vec2(1.,-1.), vec2(0.,-1.), vec2(-1.,-1.)};
33
34
35
36 using namespace std;
37 #include "QuadTree.hpp"
38
39 double dotV_LJ(double r, double eps, double r_0)
40 {
41     return -12.*eps*(pow(r_0/r,11)-pow(r_0/r,5))/(r*r);
42 }
43
44 double dotV_C(double r, double Q, double q)
45 {
```

```
46      return -k_e*Q*q/(r*r);
47  }
48
49  vec2 F(int index, Quad &myQuad)
50  {
51      //Calculate potential with every other particle
52      vec2 total = vec2(0.,0.);
53      myQuad.FindInside(Circle(r[index], cut_off));
54      //Particles need to calculate are stores in rId
55      for(int i = 0; i < rIdSize; ++i)
56      {
57          int atomPerId = rId[i];
58          double dist = length(r[index]-rPer[atomPerId]);
59          if (dist>tol)
60          {
61              total = total - dotV_LJ(dist,
62                      0.2, 0.2)*(r[index]-rPer[atomPerId])/dist;
63          }
64      }
65      /*
66      for(int atomPerId = 0; atomPerId < index; ++atomPerId)
67      {
68          if (length(r[index]-rPer[atomPerId]) < tol) {
69          std::cerr << atomPerId << " : Error \n";
70          std::cerr << "Pos: " << rPer[atomPerId] << "\n";}
71          total = total - dotV_LJ(length(r[index]-rPer[atomPerId]),
72                          0.2, 0.2)*normalize(r[index]-rPer[atomPerId
    ]);
73      }
74      for(int atomPerId = index+1; atomPerId < NPer; ++atomPerId)
75      {
76          if (length(r[index]-rPer[atomPerId]) < tol) {
77          std::cerr << atomPerId << " : Error \n";
78          std::cerr << "Pos: " << rPer[atomPerId] << "\n";}
79          total = total - dotV_LJ(length(r[index]-rPer[atomPerId]),
80                          0.2, 0.2)*normalize(r[index]-rPer[atomPerId
    ]);
81      }   */
82
83      return total;
84  }
85
86  void Update_rPer()
```

```
87 {
88     //Update rPer
89     for(int boxId = 0; boxId < 9; ++boxId)
90     {
91     for(int atomId = 0; atomId < N; ++atomId)
92     {
93         int atomPerId = boxId*N + atomId;
94         rPer[atomPerId] = r[atomId] + BoxSize*V2Add[boxId];
95     }
96     }
97 }
98
99 void Update_Quad(Quad &myQuad)
100 {
101     //Refresh Quod
102     //myQuad.QuadRemove();
103     for(int i = 0; i<NPer; ++i)
104     {
105         myQuad.insert(i);
106     }
107 }
108
109 void Initialise_Atoms()
110 {
111     //Make uniform random Box
112     uniform_real_distribution <double>distribution(0.0 ,1.);
113     mt19937_64 engine(seed);
114     for(int index = 0; index < N; ++index)
115     {
116         r[index] = BoxSize*vec2(distribution(engine), distribution(
    engine));
117         dr[index] = 5.*(vec2(distribution(engine), distribution(
    engine))-0.5);
118     }
119
120     /*
121     for (int xi = 0; xi < 3; ++xi)
122     {
123     for (int yi = 0; yi < 3; ++yi)
124     {
125         r[3*yi + xi] = vec2(1.+0.2*((double)(xi)), 1.+0.2*((double)(
    yi)));
126         dr[3*yi + xi] = vec2(0.);
```

```
127     }
128     }
129 
130     r[9] = vec2(3.0,1.2);
131     dr[9] = vec2(-5.0,0.); */
132     /*
133     r[0] = vec2(1.);
134     r[1] = vec2(1.2);
135     r[2] = vec2(1.2,1.);
136     r[3] = vec2(1.,1.2);
137     dr[0] = vec2(0.);
138     dr[1] = vec2(0.);
139     dr[2] = vec2(0.);
140     dr[3] = vec2(0.);*/
141 }
142 
143 int main () {
144 
145     Initialise_Atoms();
146     std::cerr << r[0] << r[1] << r[2] << "Start \n";
147 
148     for (int index = 0; index < N; ++index)
149     {
150         std::cout<<r[index]<<std::endl;
151     }
152 
153 
154     //START LOOP
155     for(int step = 0; step < steps; ++step)
156     {
157 
158     Update_rPer(); //For periodic conditions
159     Quad myQuad1 = Quad(Rectangle(-vec2(BoxSize),vec2(2.*BoxSize)));
160     Update_Quad(myQuad1);
161     //Calc q
162     for (int index = 0; index < N; ++index)
163     {
164         vec2 F_i = F(index, myQuad1);
165         double m_i = 1.;
166         q[index] = dr[index] + h*F_i/(2.*m_i);
167     }
168 
169     //Update r
```

```
170     for(int index = 0; index<N; ++index)
171     {
172         r[index] = r[index] + h*q[index];
173     }
174 
175 
176     Update_rPer();
177     Quad myQuad2 = Quad(Rectangle(-vec2(BoxSize),vec2(2.*BoxSize)));
178     Update_Quad(myQuad2);
179 
180     //Calc p and store in dr
181     for (int index = 0; index < N; ++index)
182     {
183         vec2 F_i = F(index, myQuad2);
184         double m_i = 1.;
185         dr[index] = q[index] + h*F_i/(2.*m_i);
186         //Ensure not to fast otherwise simulation breaks
187         dr[index] = min(1.,5./length(dr[index]))*dr[index];
188 
189     }
190 
191     //Modulate r to be inside box
192     for (int index = 0; index < N; ++index)
193     {
194         r[index] = mod(r[index], BoxSize);
195     }
196 
197     //Saved to dr
198     //Save position to file
199     for (int index = 0; index < N; ++index)
200     {
201         std::cout<<r[index]<<std::endl;
202     }
203     std::cerr << "\rSteps remaining: " << steps - step << ' ' << std
    ::flush;
204     }
205     //Completed time step
206     std::cerr << "\n" << r[0] << r[1] << r[2] << "End \n";
207     std::cerr << dr[0] << dr[1] << dr[2] << "End \n";
208     return 0;
209 }
```

D Code File: "main.cpp"

```cpp
#include <iostream>
#include <string>
#include <cassert>
#include <random>
#include "MathObjects.hpp"


//Constants
const double k_e = 14.3996;
const double cut_off = 12.;
const double tol = 0.0005;
const double Mass_Oxygen = 16.;
const double Mass_Hydrogen =1.0;
const double m_O = Mass_Oxygen;
const double m_H = Mass_Hydrogen;
const double q_O = -0.834;
const double q_H = 0.417;
const double d_OH = 0.9572;//1.;//
const double d_HH = 1.5139;//pow(2.,0.5);//
const double r_0_OO = 3.5365;
const double r_0_OH = 1.993;
const double r_0_HH = 0.449;
const double eps_OO = 0.1521;
const double eps_OH = 0.084;
const double eps_HH = 0.046;

const int seed = 3;
const double h = 0.02;
const double steps = 500;

//#define vec3 vec2


//Number of water molecules
const int N = 35;
//Takes form |O-H-H|O-H-H|O-H-H|...
vec3 r[3*N];
vec3 dr[3*N];
vec3 q[3*N];

using namespace std;
```

```
42
43
44 double sdBox(vec3 p) {//Cude centered at origin with side length 1.
45     vec3 q = abs(p) - vec3(0.5, 0.5, 0.5);
46     return length(max(q, 0.0)) + min(max(q.x, max(q.y, q.z)), 0.0);}
47 double sdBox30(vec3 p){
48     //Cube over [0,30]
49     p = p-vec3(15., 15., 15.);
50     p = p/30.;
51     return sdBox(p)*30.;}
52
53 vec3 GetNormalBox30(vec3 p) //Gets normal of the surface
54 {
55     vec3 GradF = vec3(0.0,0.0,0.0);
56     double F = sdBox30(p);
57     GradF.x = sdBox30(p+epsilon*vec3(1.,0.,0.));
58     GradF.y = sdBox30(p+epsilon*vec3(0.,1.,0.));
59     GradF.z = sdBox30(p+epsilon*vec3(0.,0.,1.));
60     GradF = GradF-F;
61     return normalize(GradF);
62 }
63
64 double dotV_LJ(double r, double eps, double r_0)
65 {
66     return -12.*eps*(pow(r_0/r,11)-pow(r_0/r,5))/(r*r);
67 }
68
69 double dotV_C(double r, double Q, double q)
70 {
71     return -k_e*Q*q/(r*r);
72 }
73
74
75 void Inner_Loop_O(int index, vec3 r_c, vec3 &total) {
76         //O to O
77         double dist = length(r_c-r[index]);
78         assert(dist>tol);
79         if (dist < cut_off)
80         {
81         total = total - (dotV_LJ(dist, eps_OO, r_0_OO) + dotV_C(dist
    , q_O, q_O))*normalize(r_c-r[index]);
82         }
83         //O to H
```

```
84          dist = length(r_c-r[index+1]);
85          assert(dist>tol);
86          if (dist < cut_off)
87          {
88          total = total - (dotV_LJ(dist, eps_OH, r_0_OH) + dotV_C(dist
    , q_O, q_H))*normalize(r_c-r[index+1]);
89          }
90          dist = length(r_c-r[index+2]);
91          assert(dist>tol);
92          if (dist < cut_off)
93          {
94          total = total - (dotV_LJ(dist, eps_OH, r_0_OH) + dotV_C(dist
    , q_O, q_H))*normalize(r_c-r[index+2]);
95          }
96  }
97
98  vec3 F_O(int i)
99  {
100     vec3 r_c = r[i];
101     vec3 total = 0.;
102
103     //Run through every Molecule Apart from itself
104     for (int index = 0; index < i; index += 3)
105     {
106         Inner_Loop_O(index, r_c, total);
107     }
108     for (int index = i+3; index < 3*N; index += 3)
109     {
110         Inner_Loop_O(index, r_c, total);
111     }
112
113     //Its atoms in molecule
114     //total -= dotV_LJ(length(r_c-r[i+1]), eps_OH, r_0_OH) + dotV_C(
    length(r_c-r[i+1]), q_O, q_H);
115     //total -= dotV_LJ(length(r_c-r[i+2]), eps_OH, r_0_OH) + dotV_C(
    length(r_c-r[i+2]), q_O, q_H);
116     total = total - (dotV_C(length(r_c-r[i+1]), q_O, q_H)+dotV_LJ(
    length(r_c-r[i+1]), eps_OH, r_0_OH))*normalize(r_c-r[i+1]);
117     total = total - (dotV_C(length(r_c-r[i+2]), q_O, q_H)+dotV_LJ(
    length(r_c-r[i+2]), eps_OH, r_0_OH))*normalize(r_c-r[i+2]);
118
119     //Against Wall
120     total = total - dotV_LJ(-sdBox30(r_c), eps_OH, 0.5*r_0_OH)*(-
```

```
    GetNormalBox30(r_c));
121     //std::cerr << "Pos: " << r_c << ", Force: " << -dotV_LJ(-
    sdBox30(r_c), eps_OH, 2.*r_0_OH)*(-GetNormalBox30(r_c)) << "\n";
122     return total;
123 }
124 
125 
126 void Inner_Loop_H(int index, vec3 r_c, vec3 &total) {
127         //H to O
128         double dist = length(r_c-r[index]);
129         assert(abs(dist)>tol);
130         if (dist < cut_off)
131         {
132         total = total - (dotV_LJ(dist, eps_OH, r_0_OH) + dotV_C(dist
    , q_H, q_O))*normalize(r_c-r[index]);
133         }
134         //H to H
135         dist = length(r_c-r[index+1]);
136         assert(abs(dist)>tol);
137         if (dist < cut_off)
138         {
139         total = total - (dotV_LJ(dist, eps_HH, r_0_HH) + dotV_C(dist
    , q_H, q_H))*normalize(r_c-r[index+1]);
140         }
141         dist = length(r_c-r[index+2]);
142         assert(abs(dist)>tol);
143         if (dist < cut_off)
144         {
145         total = total - (dotV_LJ(dist, eps_OH, r_0_OH) + dotV_C(dist
    , q_O, q_H))*normalize(r_c-r[index+2]);
146         }
147 }
148 
149 vec3 F_H(int i)
150 {
151     vec3 r_c = r[i];
152     vec3 total = vec3(0.);
153 
154     //Run through every Molecule Apart from itself
155     for (int mol_index = 0; mol_index < i/3; mol_index += 1)
156     {
157         Inner_Loop_H(3*mol_index, r_c, total);
158     }
```

```
159     for (int mol_index = i/3 + 1; mol_index < N; mol_index += 1)
160     {
161         Inner_Loop_H(3*mol_index, r_c, total);
162     }
163 
164     //Its atoms in molecule
165     //H-O
166     //std::cout<<total<<"\n";
167     int id = 3*(i/3);
168     //total -= dotV_LJ(length(r_c-r[id]), eps_OH, r_0_OH) + dotV_C(
    length(r_c-r[id]), q_O, q_H);
169     total = total - (dotV_C(length(r_c-r[id]), q_O, q_H)+dotV_LJ(
    length(r_c-r[id]), eps_OH, r_0_OH))*normalize(r_c-r[id]);
170     //std::cout<<total<<"\n";
171     //H-H
172     double dist = length(2.*r_c-r[id+1]-r[id+2]);
173     //total -= dotV_LJ(dist, eps_HH, r_0_HH) + dotV_C(dist, q_H, q_H
    );
174     total = total - (dotV_C(dist, q_H, q_H)+dotV_LJ(dist, eps_HH,
    r_0_HH))*normalize(2.*r_c-r[id+1]-r[id+2]);
175 
176 
177 
178     return total;
179 }
180 
181 bool Apply_Position_Constraint(int i, int j, double m_i, double m_j,
     double d_ij)
182 {
183     vec3 s = r[i]+h*q[i]-r[j]-h*q[j];
184     if (abs(dot(s,s)-d_ij*d_ij) > tol)
185     {
186         double g = (dot(s,s)-d_ij*d_ij)/(2.*h*dot(s,r[i]-r[j])*(1/
    m_i + 1/m_j));
187         q[i] = q[i] - g*(r[i]-r[j])/m_i;
188         q[j] = q[j] + g*(r[i]-r[j])/m_j;
189         return false;
190     } else {return true;}
191 }
192 
193 bool Apply_Velocity_Constraint(int i, int j, double m_i, double m_j,
     double d_ij)
194 {
```

```
195     if (abs(dot(dr[i]-dr[j], r[i]-r[j])) > tol)
196     {
197         double k = dot(dr[i]-dr[j],r[i]-r[j])/(d_ij*d_ij*(1./m_i +
    1./m_j));
198         dr[i] = dr[i] - k*(r[i]-r[j])/m_i;
199         dr[j] = dr[j] + k*(r[i]-r[j])/m_j;
200         return false;
201     } else {return true;}
202 }
203
204 void Save_r_ToFile()
205 {
206     //Save position to file
207     for (int index = 0; index < 3*N; ++index)
208     {
209         std::cout<<r[index]<<std::endl;
210     }
211 }
212
213 void Initialise_Atoms()
214 {
215     //Make uniform random Box
216     uniform_real_distribution <double>distribution(2. ,28.0); // U
    (0 ,30)
217     mt19937_64 engine(seed);
218     for(int index = 0; index < 3*N; index += 3)
219     {
220         r[index] = vec3(distribution(engine), distribution(engine),
    distribution(engine));
221         dr[index] = (vec3(distribution(engine), distribution(engine)
    ,distribution(engine))-15.)/3.;
222         r[index+1] = r[index] + vec3(0.,1.,0.);
223         r[index+2] = r[index] - vec3(0.,0.,1.);
224     }
225 }
226
227 void Velocity_Correction()
228 {
229     // For every Molecule
230     int counter = 0;
231     for(int index = 0; index < 3*N; index += 3)
232     {
233     //Velocity Correction
```

```
234     bool Acceptable = false;
235     counter = 0;
236     while (Acceptable == false)
237     {
238         //O is index
239         // 1st H is index+1
240         // 2nd H is index+2
241         Acceptable = true;
242         //O to 1st H
243         Acceptable = Apply_Velocity_Constraint(index, index+1, m_O,
    m_H, d_OH);
244 
245         //O to 2nd H
246         Acceptable = Apply_Velocity_Constraint(index, index+2, m_O,
    m_H, d_OH);
247 
248         //1st H to 2nd H
249         Acceptable = Apply_Velocity_Constraint(index+1, index+2, m_H
    , m_H, d_HH);
250         counter += 1;
251         if (counter > 200)
252             break;
253     }
254     }
255 }
256 
257 void Calc_Velocity()
258 {
259     //Calc p for O
260     for (int index = 0; index < 3*N; index += 3)
261     {
262         vec3 F_i = F_O(index);
263         double m_i = Mass_Oxygen;
264         dr[index] = q[index] + h*F_i/(2.*m_i);
265     }
266 
267     //Calc p for H
268     for (int index = 1; index < 3*N; index += 3)
269     {
270         double m_i = Mass_Hydrogen;
271 
272         vec3 F_i = F_H(index);
273         dr[index] = q[index] + h*F_i/(2.*m_i);
```

```
274
275         F_i = F_H(index+1);
276         dr[index+1] = q[index+1] + h*F_i/(2.*m_i);
277     }
278 }
279
280 void Position_Correction()
281 {
282     //For every molecule
283     int counter = 0;
284     for(int index = 0; index < 3*N; index += 3)
285     {
286     //Correction
287     bool Acceptable = false;
288     counter = 0;
289     while (Acceptable == false)
290     {
291         //O is index
292         // 1st H is index+1
293         // 2nd H is index+2
294         Acceptable = true;
295         //O to 1st H
296         Acceptable = Apply_Position_Constraint(index, index+1, m_O,
    m_H, d_OH);
297
298         //O to 2nd H
299         Acceptable = Apply_Position_Constraint(index, index+2, m_O,
    m_H, d_OH);
300
301         //1st H to 2nd H
302         Acceptable = Apply_Position_Constraint(index+1, index+2, m_H
    , m_H, d_HH);
303         counter += 1;
304         if (counter > 200)
305             break;
306     }
307     }
308 }
309
310 void Calc_Position()
311 {
312     //Calc q for O
313     for (int index = 0; index < 3*N; index += 3)
```

```
314     {
315         vec3 F_i = F_O(index);
316         double m_i = Mass_Oxygen;
317         q[index] = dr[index] + h*F_i/(2.*m_i);
318     }
319     //Calc q for H-H
320     for (int index = 1; index < 3*N; index += 3)
321     {
322         double m_i = Mass_Hydrogen;
323 
324         vec3 F_i = F_H(index);
325         q[index] = dr[index] + h*F_i/(2.*m_i);
326 
327         F_i = F_H(index+1);
328         q[index+1] = dr[index+1] + h*F_i/(2.*m_i);
329     }
330 }
331 
332 int main () {
333 
334     //Make uniform random Box
335     Initialise_Atoms();
336     Save_r_ToFile();
337 
338     //Time Simulation
339     StopWatch My_Watch;
340     My_Watch.Start();
341 
342     std::cerr << r[0] << r[1] << r[2] << "Start \n";
343     for(int step = 0; step < steps; ++step) {
344         //Calc q
345         Calc_Position();
346 
347         //Position Correction
348         // For every Molecule
349         Position_Correction();
350 
351         //Update r
352         for(int index = 0; index<3*N; ++index)
353         {
354             r[index] = r[index] + h*q[index];
355         }
356 
```

```
        //Calc p and store in dr
        Calc_Velocity();

        //Velocity Correction
        Velocity_Correction();

        //Saved to dr
        //Save position to file
        Save_r_ToFile();
        std::cerr << "\rSteps remaining: " << steps - step << ' '
                  << std::flush;
    }
    //Completed time step
    My_Watch.Stop();
    std::cerr << "\n End, Time: " << My_Watch.Time() << "\n";
    return 0;
}


    /*
    //Modulate Position //For inside loop
    //And Fix max speed
    for (int index = 0; index < 3*N; index += 3)
    {


        if (sdBox30(r[index])>1.5*d_OH)
        {//Outside box
            std::cerr<< r[index] << " :Point, " << sdBox30(r[index])
    << " :Dist \n";
            r[index] = mod(r[index], 30);
            r[index+1] = mod(r[index+1], 30);
            r[index+2] = mod(r[index+2], 30);
        }
        //Apply Maximum Velocity
        //dr[index] = min(1.,8./length(dr[index]))*dr[index];

    }*/
```

E Code File: "Vis2D.py"

```
import numpy as np
import matplotlib.pyplot as plt
```

```
#Read from file
file = open("Results2D.txt")
Dim = 2
a = file.readlines()
b = np.array([[float(a_[1:-2].split(', ')[i]) for i in range(Dim)]
    for a_ in a])
file.close()

#Load in data
N = 200
steps = 500#4057
end = steps*N#15*9
x = np.array([[b[i][0] for i in range(start,N+start,1)] for start in
     range(0, end, N)])
y = np.array([[b[i][1] for i in range(start,N+start,1)] for start in
     range(0, end, N)])
z = np.array([[0.1 for i in range(start,N+start,1)] for start in
    range(0, end, N)])

#Plot Graph
end = 190
plt.xlim(0,10)
plt.ylim(0,10)
plt.xlabel("x")
plt.ylabel("y")

for i in range(1,20):
    plt.scatter(x[end-i],y[end-i], s = 2, color = "blue")

plt.scatter(x[end],y[end], s=20, color = "red")
```

F Code File: "Vis3D.py"

```
#ipyVolume only works in Jupyter Notebook
import ipyvolume as ipv
import ipywidgets as widgets
import numpy as np
import matplotlib.pyplot as plt


#Read Data
file = open("Results3D.txt")
Dim = 3
```

```
a = file.readlines()
b = np.array([[float(a_[1:-2].split(', ')[i]) for i in range(Dim)]
    for a_ in a])
file.close()

#Format Data
N = 35
steps = 500
end = 3*steps*N#15*9
x_O = np.array([[b[i][0] for i in range(start,3*N+start,3)] for
    start in range(0, end, 3*N)])
x_H1 = np.array([[b[i+1][0] for i in range(start,3*N+start,3)] for
    start in range(0, end, 3*N)])
x_H2 = np.array([[b[i+2][0] for i in range(start,3*N+start,3)] for
    start in range(0, end, 3*N)])
y_O = np.array([[b[i][1] for i in range(start,3*N+start,3)] for
    start in range(0, end, 3*N)])
y_H1 = np.array([[b[i+1][1] for i in range(start,3*N+start,3)] for
    start in range(0, end, 3*N)])
y_H2 = np.array([[b[i+2][1] for i in range(start,3*N+start,3)] for
    start in range(0, end, 3*N)])
if Dim == 3:
     z_O = np.array([[b[i][2] for i in range(start,3*N+start,3)] for
    start in range(0, end, 3*N)])
     z_H1 = np.array([[b[i+1][2] for i in range(start,3*N+start,3)]
    for start in range(0, end, 3*N)])
     z_H2 = np.array([[b[i+2][2] for i in range(start,3*N+start,3)]
    for start in range(0, end, 3*N)])


#Needed for trace of path
def rA(arr, d):
    if d < 0:
        d=len(arr)+d

    temp = []
    n = len(arr)
    arr = arr.tolist()
    i = 0
    while (i < d):
        temp.append(arr[i])
        i = i + 1
    i = 0
```

```
    while (d < n):
        arr[i] = arr[d]
        i = i + 1
        d = d + 1
    arr[:] = arr[: i] + temp
    return np.array(arr)

#Display Data
ipv.figure()
s = ipv.scatter(x_O, y_O, z_O, marker='sphere', size=4.5)
h1 = ipv.scatter(x_H1, y_H1, z_H1, marker='sphere', size=2., color =
    "blue")
h2 = ipv.scatter(x_H2, y_H2, z_H2, marker='sphere', size=2., color =
    "blue")
ipv.animation_control(s)#, add = True, interval=200)
mylink1 = widgets.link((s, 'sequence_index'), (h1, 'sequence_index')
   )
mylink2 = widgets.link((s, 'sequence_index'), (h2, 'sequence_index')
   )
links = []
hs = []
for i in range(1,15):
    h = ipv.scatter(rA(x_O,-2*i), rA(y_O,-2*i), rA(z_O,-2*i), marker
   ='sphere', size=1.5, color = "orange")
    link = widgets.link((s, 'sequence_index'), (h, 'sequence_index')
   )

    links.append(link)
    hs.append(h)

ipv.show()
```

G Code File: "ShowQuadTree.cpp"

```
#include <iostream>
#include <string>
#include <random>

#include "MathObjects.hpp"

#define WITHOUT_NUMPY
#include "matplotlibcpp.h"
//I did not make the matplotlib interface
```

```
10
11
12
13 using namespace std;
14 namespace plt = matplotlibcpp;
15 const int seed = 1;
16 const int N = 400;
17 const int NPer = N;
18
19 const double PartSize = 14.;
20
21
22 vec2 rPer[N];
23
24
25 #include "QuadTree.hpp"
26
27
28 double sdBox(vec3 p) {
29     //Cude centered at origin with side length 1.
30     vec3 q = abs(p) - vec3(0.5, 0.5, 0.5);
31     return length(max(q, 0.0)) + min(max(q.x, max(q.y, q.z)), 0.0);}
32
33 double sdBox30(vec3 p){
34     //Cube over [0,30]
35     p = p-vec3(15., 15., 15.);
36     p = p/30.;
37     return sdBox(p)*30.;
38 }
39
40
41 void PlotRectangle(Rectangle& myRec, std::string color = "black")
42 {
43     std::vector<double> x(5), y(5);
44     x[0] = myRec.BL.x; y[0] = myRec.BL.y;
45     x[1] = myRec.BL.x; y[1] = myRec.TR.y;
46     x[2] = myRec.TR.x; y[2] = myRec.TR.y;
47     x[3] = myRec.TR.x; y[3] = myRec.BL.y;
48     x[4] = x[0]; y[4] = y[0]; //Loop around
49
50     plt::plot(x,y, {{"color", color}});
51 }
52
```

```
53 void PlotQuad(Quad& myQuad)
54 {
55     if (myQuad.hasSub)
56     {
57         PlotQuad(*myQuad.NW);
58         PlotQuad(*myQuad.NE);
59         PlotQuad(*myQuad.SE);
60         PlotQuad(*myQuad.SW);
61     } else if (myQuad.PointSize == 1) {
62         //Store output since last \n
63         std::vector<double> x(1), y(1);
64         x[0] = rPer[myQuad.Id].x;
65         y[0] = rPer[myQuad.Id].y;
66         plt::scatter(x,y, PartSize, {{"color", "red"}});
67         //Plot rectangle with points inside
68         PlotRectangle(myQuad.d0);
69 
70     } else {
71     //Plot rectangle with no points
72     PlotRectangle(myQuad.d0, "black");}
73 
74 }
75 
76 void PlotCircle(Circle C, std::string color)
77 {
78     int ths = 40;
79     std::vector<double> x(ths+1), y(ths+1);
80     for(int i=0; i<=ths; ++i)
81     {
82         double theta = 2.*PI*((double)(i))/((double)(ths));
83         x[i] = C.mid.x+C.r*cos(theta);
84         y[i] = C.mid.y+C.r*sin(theta);
85     }
86     plt::plot(x, y, {{"color", color}});
87 
88 }
89 
90 void PlotrIdData(Circle C)
91 {
92     PlotCircle(C, "blue");
93     std::cerr << rIdSize << " ID Size \n";
94     std::vector<double> x(rIdSize+1), y(rIdSize+1);
95     for(int i=0; i<rIdSize; ++i)
```

```
    {
        x[i] = rPer[rId[i]].x;
        y[i] = rPer[rId[i]].y;
    }
    plt::scatter(x,y, PartSize, {{"color", "green"}});
}


int main()
{
   uniform_real_distribution <double>distribution(0.0 ,10.); // U (0
    ,30)
   mt19937_64 engine(seed);
   std::cout << "Hello World \n";
   StopWatch My_Watch;
   My_Watch.Start();
   Quad myQuad = Quad(Rectangle(vec2(0.),vec2(10.)));

   //Insert loop
   for (int i = 0; i < N; ++i)
   {
       rPer[i] = vec2(distribution(engine), distribution(engine));
       myQuad.insert(i);
   }

   myQuad.FindInside(Circle(rPer[0],4.));
   std::cerr << rIdSize << " :rIdSize \n";
   My_Watch.Stop();
   /*
   myQuad.insert(vec2(2.5));
   myQuad.insert(vec2(3.5));
   myQuad.insert(vec2(4.5));*/
   std::cout << "Time: " << My_Watch.Time() << "\n";
   //std::cout << myQuad << "\n";

   //Plot Figure
   plt::figure_size(800,800);
   PlotQuad(myQuad);
   PlotrIdData(Circle(rPer[0],4.));
   plt::xlim(0,10);
   plt::ylim(0,10);
   plt::save("MyGraph.png");

```

```
    return 0;
}
```

H Code File: "MathObjects.hpp"

```
#include <iostream>
#include <math.h>


//StopWatch
#include <ctime>
#include <ratio>
#include <chrono>

#define float double
//Set __CUDA_CALL__ to correct vaule depends if c++ or cuda
//__CUDA_CALL__ means it can be called by GPU
#ifdef __CUDACC__
#define __CUDA_CALL__ __host__ __device__
#else
#define __CUDA_CALL__
#endif

using namespace std;


// ************************
// ************************
// **                    **
// ** RayMarch Constants **
// **                    **
// ************************
// ************************

#define PI 3.1415926538
#define MAX_STEPS 70
#define MAX_DISTANCE 100.
#define MIN_STEP_SIZE 0.001
#define epsilon 0.00001
#define inf 100000000.


// ***************************
```

```
39 // ***************************
40 // **                       **
41 // ** Standard Math Objects **
42 // **                       **
43 // ***************************
44 // ***************************
45
46 template<typename T, int SIZE>
47 struct VectorOps
48 {
49
50     virtual __CUDA_CALL__ T* getVec() = 0;
51
52     __CUDA_CALL__
53     T operator+(const T& b)
54     {
55         T* a = this->getVec();
56         T c;
57         for (int i=0; i<SIZE; i++)
58         {
59             c.data[i] = a->data[i] + b.data[i];
60         }
61         return c;
62     }
63
64     __CUDA_CALL__
65     T operator+(const double& b)
66     {
67         T a = *(this->getVec());
68         return a+T(b);
69     }
70
71     __CUDA_CALL__
72     T operator-(T b)
73     {
74         T a = *(this->getVec());
75         return a+b*(-1.);
76     }
77
78     __CUDA_CALL__
79     T operator-(const double& b)
80     {
81         T a = *(this->getVec());
```

```
82          return a+T(-b);
83      }
84
85      __CUDA_CALL__
86      T operator-()
87      {
88          T a = *(this->getVec());
89          return a*(-1.);
90      }
91
92      __CUDA_CALL__
93      T operator*(const T& b)
94      {
95          T* a = this->getVec();
96          T c;
97          for (int i=0; i<SIZE; i++)
98          {
99              c.data[i] = a->data[i] * b.data[i];
100         }
101         return c;
102     }
103
104     __CUDA_CALL__
105     T operator*(const double& b)
106     {
107         T a = *(this->getVec());
108         return a*T(b);
109     }
110
111     __CUDA_CALL__
112     T operator/(const T& b)
113     {
114         T* a = this->getVec();
115         T c;
116         for (int i=0; i<SIZE; i++)
117         {
118             c.data[i] = a->data[i] / b.data[i];
119         }
120         return c;
121     }
122
123     __CUDA_CALL__
124     double& operator[](int i) //For some reason works for assignment
```

```
    as well :)
    {
        T* a = this->getVec();
        return a->data[i];
    }
};

template<typename T>
__CUDA_CALL__
T operator* (double const& lhs, T rhs) {
  return rhs*lhs;
}

template<typename T>
__CUDA_CALL__
T operator+ (double const& lhs, T rhs) {
  return rhs+lhs;
}


template<typename T>
__CUDA_CALL__
T operator- (double const& lhs, T rhs) {
  return -(rhs-lhs);
}



template<int POS>
struct scalar_swizzle
{
    double v[POS+1];
    double& operator=(const double x)
    {
        v[POS] = x;
        return v[POS];
    }
    operator double() const
    {
        return v[POS];
    }
};

```

```
167 //Overload subscript operator. []
168
169 struct vec2 : VectorOps<vec2, 2>
170 {
171     /*
172     union{
173         double data[2];
174         scalar_swizzle<0> x;
175         scalar_swizzle<1> y;
176     };
177     */
178
179     union{
180         struct{ double data[2];};
181         struct{double x,y;};
182         struct{double r,g;};
183     };
184
185     __CUDA_CALL__
186     vec2()
187     {
188         this->x=0.0;
189         this->y=0.0;
190     }
191
192     __CUDA_CALL__
193     vec2(double x, double y)
194     {
195         this->x=x;
196         this->y=y;
197     }
198
199     __CUDA_CALL__
200     vec2(double x)
201     {
202         this->x=x;
203         this->y=x;
204     }
205
206     //VectorOps Inheritence
207     __CUDA_CALL__
208     vec2* getVec()
209     {
```

```
210         return this;
211     }
212     using VectorOps<vec2, 2>::operator+;
213     using VectorOps<vec2, 2>::operator*;
214 
215     friend std::ostream& operator<<(std::ostream& output, const vec2
        & rVec);
216 };
217 
218 
219 
220 struct vec3 : VectorOps<vec3, 3>
221 {
222     /*
223     union{
224         double data[3];
225         scalar_swizzle<0> x;
226         scalar_swizzle<1> y;
227         scalar_swizzle<2> z;
228     };
229     */
230         union{
231         struct{ double data[3];};
232         struct{double x,y,z;};
233         struct{double r,g,b;};
234     };
235 
236     __CUDA_CALL__
237     vec3()
238     {
239         this->x=0.0;
240         this->y=0.0;
241         this->z=0.0;
242     }
243 
244     __CUDA_CALL__
245     vec3(double x, double y, double z)
246     {
247         this->x=x;
248         this->y=y;
249         this->z=z;
250     }
251 
```

```
252     __CUDA_CALL__
253     vec3(vec2 a, double z)
254     {
255         this->x=a.x;
256         this->y=a.y;
257         this->z=z;
258     }
259 
260     __CUDA_CALL__
261     vec3(double x, vec2 b)
262     {
263         this->x=x;
264         this->y=b.x;
265         this->z=b.y;
266     }
267 
268     __CUDA_CALL__
269     vec3(double x)
270     {
271         this->x=x;
272         this->y=x;
273         this->z=x;
274     }
275 
276     //VectorOps Inheritence
277     __CUDA_CALL__
278     vec3* getVec()
279     {
280         return this;
281     }
282     using VectorOps<vec3, 3>::operator+;
283     using VectorOps<vec3, 3>::operator*;
284 
285     friend std::ostream& operator<<(std::ostream& output, const vec3
    & rVec);
286 };
287 
288 
289 // ***************************
290 // ***************************
291 // **                       **
292 // ** Mathematic Operations **
293 // **                       **
```

```
294 // ***************************
295 // ***************************
296
297 //Absolute Value
298 __CUDA_CALL__
299 vec2 abs(vec2 a);
300 __CUDA_CALL__
301 vec3 abs(vec3 a);
302
303
304 //Max
305 __CUDA_CALL__
306 vec2 max(vec2 a, vec2 b);
307 __CUDA_CALL__
308 vec3 max(vec3 a, vec3 b);
309
310 //Min
311 __CUDA_CALL__
312 vec2 min(vec2 a, vec2 b);
313 __CUDA_CALL__
314 vec3 min(vec3 a, vec3 b);
315
316 //Clamp
317 __CUDA_CALL__
318 double clamp(double x, double minVal, double maxVal);
319 __CUDA_CALL__
320 vec2 clamp(vec2 x, double minVal, double maxVal);
321 __CUDA_CALL__
322 vec3 clamp(vec3 x, double minVal, double maxVal);
323
324 //Floor
325 __CUDA_CALL__
326 vec2 floor(vec2 x);
327 __CUDA_CALL__
328 vec3 floor(vec3 x);
329
330 //SmoothStep
331
332
333 //Sign
334 __CUDA_CALL__
335 double sign(double x);
336
```

```
337 //Mod
338 __CUDA_CALL__
339 double mod(double x, double y);
340 __CUDA_CALL__
341 vec2 mod(vec2 x, double y);
342 __CUDA_CALL__
343 vec3 mod(vec3 x, double y);
344
345
346 //ModRange (Not in GLSL)
347 __CUDA_CALL__
348 double mod(double x, double x_low, double x_high);
349
350 //SmoothStep
351 __CUDA_CALL__
352 double smoothstep(double edge0, double edge1, double x);
353 __CUDA_CALL__
354 vec2 smoothstep(double edge0, double edge1, vec2 x);
355 __CUDA_CALL__
356 vec3 smoothstep(double edge0, double edge1, vec3 x);
357
358
359 // **********************
360 // **********************
361 // **                  **
362 // ** Vector Functions **
363 // **                  **
364 // **********************
365 // **********************
366
367 //Cross
368 __CUDA_CALL__
369 vec3 cross(vec3 x, vec3 y);
370
371
372 //Sum (Not in GLSL, here to make code neater)
373 //(Tested by being used by other functions)
374 __CUDA_CALL__
375 double sum(double x);
376 __CUDA_CALL__
377 double sum(vec2 x);
378 __CUDA_CALL__
379 double sum(vec3 x);
```

```
380
381 //Length
382 __CUDA_CALL__
383 double length(double x);
384 __CUDA_CALL__
385 double length(vec2 x);
386 __CUDA_CALL__
387 double length(vec3 x);
388
389 //Dot
390 __CUDA_CALL__
391 double dot(double x, double y);
392 __CUDA_CALL__
393 double dot(vec2 x, vec2 y);
394 __CUDA_CALL__
395 double dot(vec3 x, vec3 y);
396
397 //Normalize
398 __CUDA_CALL__
399 double normalize(double x);
400 __CUDA_CALL__
401 vec2 normalize(vec2 x);
402 __CUDA_CALL__
403 vec3 normalize(vec3 x);
404
405 //Distance
406
407
408 // ***************
409 // ***************
410 // **           **
411 // ** StopWatch **
412 // **           **
413 // ***************
414 // ***************
415 class StopWatch {
416 private:
417  std::chrono::high_resolution_clock::time_point t1;
418  std::chrono::high_resolution_clock::time_point t2;
419
420 public:
421  void Start();
422  void Stop();
```

```
423  float Time();
424 };
```

I Code File: "MathFunctions.cpp"

```
1  #include"MathObjects.hpp"
2  #include<iostream>
3  using namespace std;
4
5
6
7  // ***************************
8  // ***************************
9  // **                       **
10 // ** Standard Math Objects **
11 // ** (Print functions)     **
12 // ***************************
13 // ***************************
14
15 std::ostream& operator<<(std::ostream& output, const vec2& rVec)
16 {
17     output << "(" << rVec.x << ", " << rVec.y << ")";
18     return output;
19 }
20
21 std::ostream& operator<<(std::ostream& output, const vec3& rVec)
22 {
23     output << "(" << rVec.x << ", " << rVec.y << ", " << rVec.z << "
       )";
24     return output;
25 }
26
27 // ***************************
28 // ***************************
29 // **                       **
30 // ** Mathematics Functions **
31 // **                       **
32 // ***************************
33 // ***************************
34
35 //Absolute
36 template<typename T, int SIZE>
37 __CUDA_CALL__
```

```
38 T abs(T a)
39 {
40     for(int i=0; i<SIZE; i++)
41     {
42         a[i] = abs(a[i]);
43     }
44     return a;
45 }
46
47 __CUDA_CALL__
48 vec2 abs(vec2 a) {return abs<vec2,2>(a);}
49 __CUDA_CALL__
50 vec3 abs(vec3 a) {return abs<vec3,3>(a);}
51
52
53 //Max and Min
54 template<typename T, int SIZE>
55 __CUDA_CALL__
56 T Max(T a, T b)
57 {
58     for(int i=0; i<SIZE; i++)
59     {
60         a[i] = max(a[i], b[i]);
61     }
62     return a;
63 }
64
65 __CUDA_CALL__
66 vec2 max(vec2 a, vec2 b) {return Max<vec2,2>(a,b);}
67 __CUDA_CALL__
68 vec3 max(vec3 a, vec3 b) {return Max<vec3,3>(a,b);}
69
70 template<typename T, int SIZE>
71 __CUDA_CALL__
72 T Min(T a, T b)
73 {
74     for(int i=0; i<SIZE; i++)
75     {
76         a[i] = min(a[i], b[i]);
77     }
78     return a;
79 }
80 __CUDA_CALL__
```

```
81 vec2 min(vec2 a, vec2 b) {return Min<vec2,2>(a,b);}
82 __CUDA_CALL__
83 vec3 min(vec3 a, vec3 b) {return Min<vec3,3>(a,b);}
84
85
86 //Clamp
87 template<typename T>
88 __CUDA_CALL__
89 T Clamp(T x, double minVal, double maxVal) {return min(max(x, minVal
      ), maxVal);}
90 __CUDA_CALL__
91 double clamp(double x, double minVal, double maxVal)
92  {return Clamp<double>(x, minVal, maxVal);}
93 __CUDA_CALL__
94 vec2 clamp(vec2 x, double minVal, double maxVal)
95  {return Clamp<vec2>(x, minVal, maxVal);}
96 __CUDA_CALL__
97 vec3 clamp(vec3 x, double minVal, double maxVal)
98  {return Clamp<vec3>(x, minVal, maxVal);}
99
100
101 //Floor (Tested as Needed for Mod)
102 template<typename T, int SIZE>
103 __CUDA_CALL__
104 T Floor(T x)
105 {
106    for(int i=0; i<SIZE; i++)
107     {
108         x[i] = floor(x[i]);
109     }
110     return x;
111 }
112 __CUDA_CALL__
113 vec2 floor(vec2 x) {return Floor<vec2,2>(x);}
114 __CUDA_CALL__
115 vec3 floor(vec3 x) {return Floor<vec3,3>(x);}
116
117 //SmoothStep
118 template<typename T>
119 __CUDA_CALL__
120 T SmoothStep(double edge0, double edge1, T x)
121 {
122     T t;
```

```
123     t = clamp((x - edge0) / (edge1 - edge0), 0.0, 1.0);
124     return t * t * (3.0 - 2.0 * t);
125 }
126 __CUDA_CALL__
127 double smoothstep(double edge0, double edge1, double x){return
       SmoothStep<double>(edge0, edge1, x);}
128 __CUDA_CALL__
129 vec2 smoothstep(double edge0, double edge1, vec2 x){return
       SmoothStep<vec2>(edge0, edge1, x);}
130 __CUDA_CALL__
131 vec3 smoothstep(double edge0, double edge1, vec3 x){return
       SmoothStep<vec3>(edge0, edge1, x);}
132
133 //Sign
134 __CUDA_CALL__
135 double sign(double x)
136 {
137     if (x<0.00001 & x>-0.00001)
138         return 0.;
139     return normalize(x);
140 }
141
142
143
144 //Mod
145 template<typename T>
146 __CUDA_CALL__
147 T Mod(T x, double y) {return x+(-1.)*y*floor(x/y);}
148 __CUDA_CALL__
149 double mod(double x, double y) {return Mod<double>(x,y);}
150 __CUDA_CALL__
151 vec2 mod(vec2 x, double y) {return Mod<vec2>(x,y);}
152 __CUDA_CALL__
153 vec3 mod(vec3 x, double y) {return Mod<vec3>(x,y);}
154
155 //ModRange (Not in GLSL)
156 __CUDA_CALL__
157 double mod(double x, double x_low, double x_high)
158 {return mod(x-x_low,x_high-x_low)+x_low;}
159
160
161 // **********************
162 // **********************
```

```
163 // **                    **
164 // ** Vector Functions **
165 // **                    **
166 // ************************
167 // ************************
168
169 //Cross
170 __CUDA_CALL__
171 vec3 cross(vec3 x, vec3 y)
172 {
173     return vec3(x[1]*y[2]-x[2]*y[1],
174                 x[2]*y[0]-x[0]*y[2],
175                 x[0]*y[1]-x[1]*y[0]);
176 }
177
178 //Sum
179 template<typename T, int SIZE>
180 __CUDA_CALL__
181 double Sum(T x)
182 {
183     double total = 0.;
184     for(int i = 0; i<SIZE; i++)
185     {
186         total += x[i];
187     }
188     return total;
189 }
190 __CUDA_CALL__
191 double sum(double x) {return x;}
192 __CUDA_CALL__
193 double sum(vec2 x) {return Sum<vec2,2>(x);}
194 __CUDA_CALL__
195 double sum(vec3 x) {return Sum<vec3,3>(x);}
196
197
198 //Dot
199 template<typename T>
200 __CUDA_CALL__
201 double Dot(T x, T y) {return sum(x*y);}
202 __CUDA_CALL__
203 double dot(double x, double y) {return Dot<double>(x,y);}
204 __CUDA_CALL__
205 double dot(vec2 x, vec2 y) {return Dot<vec2>(x,y);}
```

```
206 __CUDA_CALL__
207 double dot(vec3 x, vec3 y) {return Dot<vec3>(x,y);}
208
209 //Length
210 template<typename T>
211 __CUDA_CALL__
212 double Length(T x)
213 {
214     return sqrt(dot(x,x));
215 }
216 __CUDA_CALL__
217 double length(double x) {return Length<double>(x);}
218 __CUDA_CALL__
219 double length(vec2 x) {return Length<vec2>(x);}
220 __CUDA_CALL__
221 double length(vec3 x) {return Length<vec3>(x);}
222
223
224 //Normalize
225 template<typename T>
226 __CUDA_CALL__
227 T Normalize(T x) {return x/length(x);}
228 __CUDA_CALL__
229 double normalize(double x) {return Normalize<double>(x);}
230 __CUDA_CALL__
231 vec2 normalize(vec2 x) {return Normalize<vec2>(x);}
232 __CUDA_CALL__
233 vec3 normalize(vec3 x) {return Normalize<vec3>(x);}
234
235 //Distance
236
237
238
239 // ***************
240 // ***************
241 // **            **
242 // ** StopWatch **
243 // **            **
244 // ***************
245 // ***************
246 void StopWatch::Start()
247 {
248     t1 = std::chrono::high_resolution_clock::now();
```

```
249 }
250
251 void StopWatch::Stop()
252 {
253     t2 = std::chrono::high_resolution_clock::now();
254 }
255
256 float StopWatch::Time()
257 {
258     std::chrono::duration<double> time_span = std::chrono::
    duration_cast<std::chrono::duration<double>>(t2 - t1);
259     return time_span.count();
260 }
```