University of Utrecht

Homework Report 1

Student name: *Siyuan Wang 0485101, Linglan Li 4508459*

Course: *Modeling and Simulation*
Due date: *February 27th, 2024*

1. Introduction

In the *Homework Report 1* I will do the following things:

1. Write a C program to generate the data file of a cubic lattice.

2. Calculate the maximum volume fraction for cubic lattice.

3. Generate the data file of a face-centered cubic (FCC) lattice.

4. Calculate the maximum volume fraction of a FCC lattice and compare it with a cubic lattice.

5. Complete the `read_data()` function.

6. Complete the `move_particle()` function and implement displacement without considering overlapping.

7. Improve the `move_particle()` function by considering overlapping.

8. Visualize the model.

9. Do more research on `packing_fraction`.

2. Implementation

2.1. Build a cubic lattice. The implementation of the cubic lattice was written in `cubic_lattice.c`. Some code snippets are shown in Sec. 3.1. The generated data file is `cubic.dat`. The first 6 lines are:

```
3375
0.000000 15.000000
0.000000 15.000000
0.000000 15.000000
0.000000  0.000000  0.000000  1.000000
0.000000  0.000000  1.000000  1.000000
```

cubic.dat

Here $N = 3375$ because I let the length of the cubic be 15. That makes $15^3 = 3375$. The visualized cubic is shown in Fig. 1

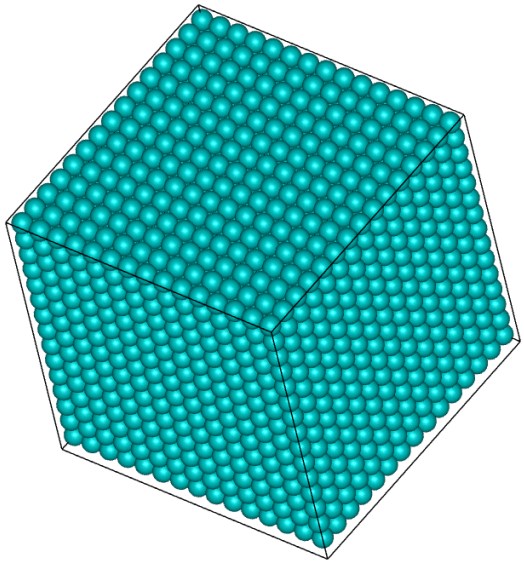


Figure 1: A Cubic Lattice.

2.2. Calculate the maximum volume fraction of a cubic lattice. To calculate the maximum volume fraction of a cubic lattice, we can refer to Fig. 2. As we can see, the sphere has occupied the maximum volume of the cubic, with the radius r. Meanwhile, the cubic has its side length $2r$.

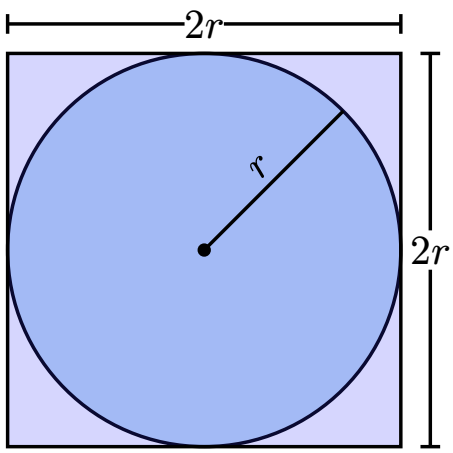


Figure 2: Fraction Volume of a Cubic Lattice.

Then we can get **the volume of the sphere**:

$$V_{\text{sphere}} = \frac{4}{3}\pi r^3, \tag{2.1}$$

the volume of the cubic:

$$V_{\text{cubic}} = (2r)^3 = 8r^3, \tag{2.2}$$

the maximum volume fraction:

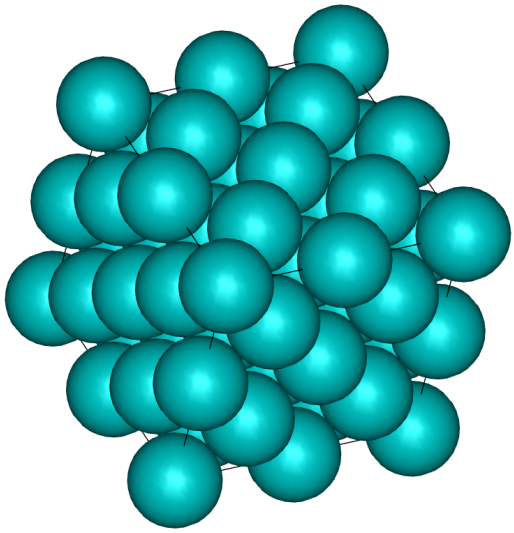


Figure 3: A FCC Lattice.

$$fraction = \frac{V_{\text{sphere}}}{V_{\text{cubic}}} = \frac{\pi}{6}. \tag{2.3}$$

2.3. Build a FCC lattice. The implementation of the FCC lattice was written in `fcc_lattice.c`. Some code snippets are shown in Sec. 3.2. The generated data file is `fcc.dat`. The visualized FCC lattice is shown in Fig. 3.

2.4. Calculate the maximum volume fraction of an FCC lattice. To calculate the maximum volume fraction of a FCC lattice, we can refer to Fig. 4. In the 3 dimensional space, the maximum volume means there are 2 complete balls in the cubic.

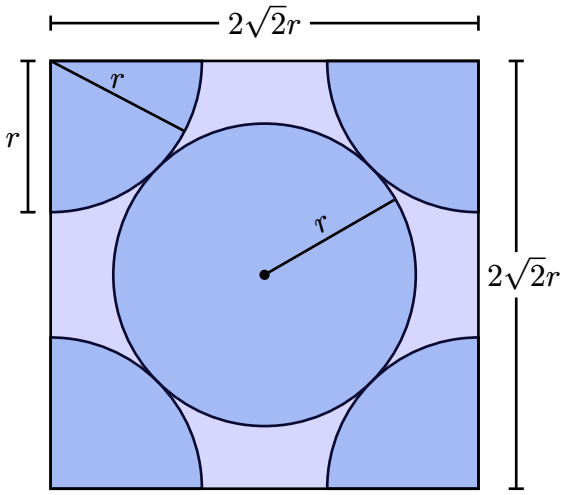


Figure 4: Fraction Volume of a Cubic Lattice.

Then we can get the volume of the spheres:

$$V_{\text{sphere}} = 2 \times \frac{4}{3}\pi r^3, \tag{2.4}$$

the volume of the cubic:

$$V_{\text{cubic}} = \left(2\sqrt{2}r\right)^3 = 16\sqrt{2}r^3. \tag{2.5}$$

Thus the maximum volume fraction is:

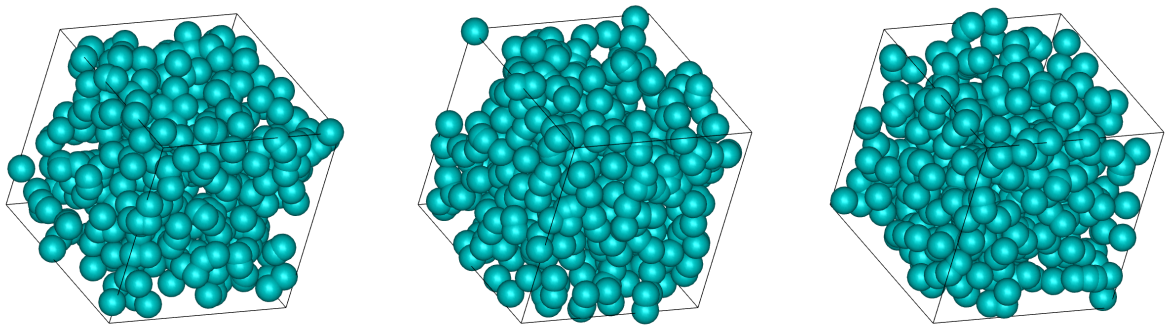
$$fraction = \frac{V_{\text{sphere}}}{V_{\text{cubic}}} = \frac{\pi}{6\sqrt{2}}. \tag{2.6}$$

Compared to the maximum volume fraction of a cubic lattice, we can see that the value is a little smaller, which means FCC lattices have lower space utilization.

2.5. Complete the `read_data()` function. The code of `read_data()` imitates the structure of `write_data(int step)`, allowing it to correctly read the data in `coords_step0000000.dat`. The code snippet is shown in Sec. 3.3.

2.6. Complete the `move_particle()` function without considering overlapping. The code of `move_particle()` is written in Sec. 3.4. For this function, overlapping is not detected, and it will always return 1. Periodic boundary condition is always applied.

Here are some outputs when the packing fraction equals 0.6 and Δx equals 0.1. From Fig. 5 we can see overlapping takes place in every figure. Thus we must consider some ways to fix this problem.



(a) $step = 100$. (b) $step = 1000$. (c) $step = 9000$.

Figure 5: Outputs when packing fraction = 0.6 and $\Delta x = 0.1$.

2.7. Complete the `movo_particle()` function by considering overlapping. Now we consider the overlapping. By calculating the distance between a newly moved particle and other particles, we can know whether the movement should be accepted. The improved code is placed in Sec. 3.5.

According to the lecture note, a very small or very large value of Δx makes the simulation very inefficient. We should choose a Δx to make the acceptance to be around 50%. The lecture note suggests that in all cases, Δx should not be larger than half the box size.

Here are several examples of packing fraction and delta in Table 1.

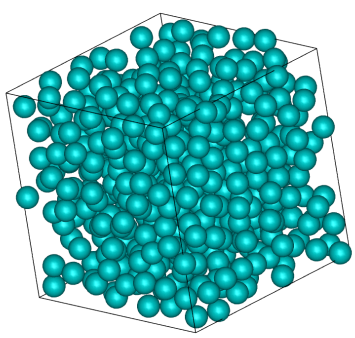
From Table 1 we can see that the value of chosen Δx increases monotonically as the packing fraction decreases, which is consistent with the principle of crystal packing.

Table 1: List of resonable Δx.

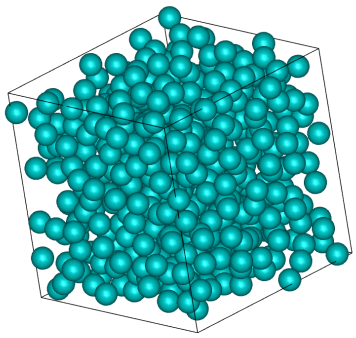
Packing Fraction	$\boldsymbol{\Delta x}$	**Average Acceptance**
0.7	0.006	0.524040
0.6	0.03	0.517964
0.5	0.05	0.536721
0.4	0.08	0.510905
0.3	0.15	0.528917
0.2	0.3	0.549655
0.1	1.0	0.512143

When the packing fraction drops, crystal density decreases, causing atoms to be farther apart. With more space between atoms, the chance of moving any atom without overlap increases.

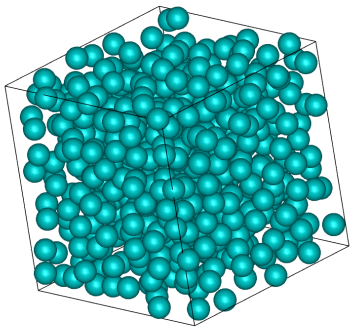
Hence, the Δx values also increase. This simple connection shows how changes in packing fraction impact crystal density and atomic movements.



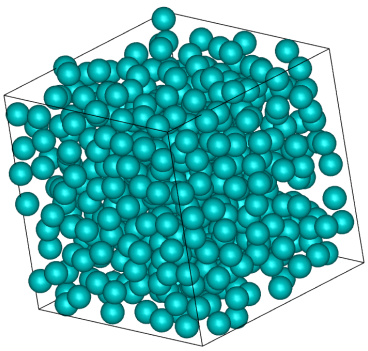
(a) $step = 100$.



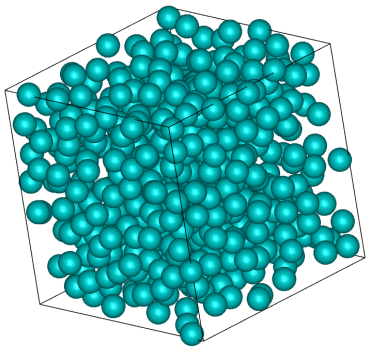
(b) $step = 500$.



(c) $step = 1000$.



(d) $step = 5000$.



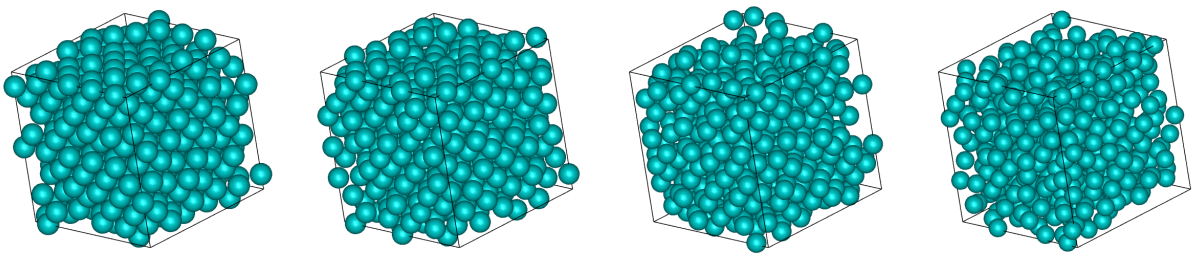
(e) $step = 9900$.

Figure 6: Outputs when packing fraction = 0.6 and $\Delta x = 0.1$.

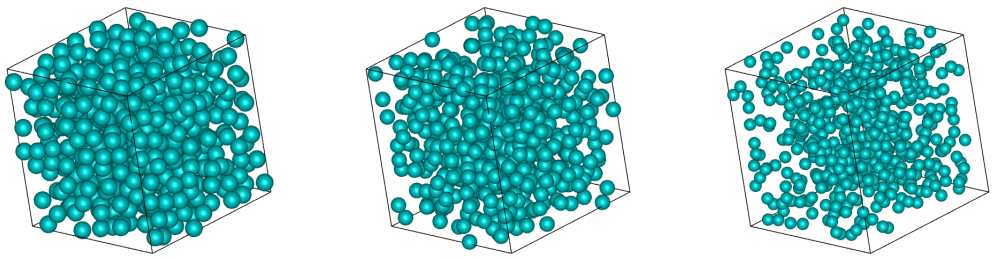
2.8. Visualize the model. We choose the value of packing fraction equals 0.3 and Δx equals 0.15 to run the simulation and check the visual result using the Webgl code. The results are shown in Fig. 6.

2.9. Estimate of packing fraction when the FCC crystal melts. To estimate the packing fraction, we will run the simulation using various packing fractions.

FCC crystals have approximately 70% packing, this is the highest possible packing for spheres of the same size. For the melting situation, we choose to adjust the value of the packing fraction decreasing from 0.7 to 0.1, and we also adjust the Δx to make sure acceptance is around 0.5 in the meantime, as shown in Table 1.



(a) $P.F. = 0.7$. (b) $P.F. = 0.6$. (c) $P.F. = 0.5$. (d) $P.F. = 0.4$.



(e) $P.F. = 0.3$. (f) $P.F. = 0.2$. (g) $P.F. = 0.1$.

Figure 7: Outputs when $step = 100$.

From Fig. 7, we can see that the FCC crystal structure becomes disordered as the packing fraction continues to decrease. When the packing fraction is equal to 0.4, the crystal completely loses its ordered structure, which proves that the crystal melts from solid to liquid at this time. This point is the melting point.

3. Code

Code snippets are shown here.

3.1. Code snippets for Sec. 2.1. Code snippet for generating data points.

cubic_lattice.c

```
#define NDIM 3
#define LENGTH 15
const float amplifier = 1.0;
const double diameter = 1.0;
int N = (int)pow(LENGTH, NDIM);
// Here I use a pointer so that I can decide
// how much space to malloc for different dimensional lattices.
double (*r)[NDIM];
r = malloc(N * sizeof(*r));
```

```c
double box[NDIM];
void ConstructCubicLattice()
{
    int n = 0;
    if (NDIM == 2)
    {
        for (int i = 0; i < LENGTH; i++)
        {
            for (int j = 0; j < LENGTH; j++)
            {
                r[n][0] = i * amplifier;
                r[n][1] = j * amplifier;
                n++;
            }
        }
    }
    else if (NDIM == 3)
    {
        for (int i = 0; i < LENGTH; i++)
        {
            for (int j = 0; j < LENGTH; j++)
            {
                for (int k = 0; k < LENGTH; k++)
                {
                    r[n][0] = i * amplifier;
                    r[n][1] = j * amplifier;
                    r[n][2] = k * amplifier;
                    n++;
                }
            }
        }
    }
}
```

Code snippet for saving the data points to file `cubic.dat`.

cubic_lattice.c

```c
void WriteData()
{
    FILE *fp = fopen(writtenFileNameAbsolute, "w");
    printf("File created.\n");
    fprintf(fp, "%d\n", N);
    for (int d = 0; d < NDIM; d++)
    {
        fprintf(fp, "%lf %lf\n", 0.0, box[d]);
    }
    for (int n = 0; n < N; n++)
    {
        for (int d = 0; d < NDIM; d++)
        {
            fprintf(fp, "%lf\t", r[n][d]);
        }
        fprintf(fp, "%lf\n", diameter);
    }
    fclose(fp);
}
```

3.2. Code snippets for Sec. 2.3. Code snippet for reallocating memory to suit the change of `N`.

fcc_lattice.c

```c
void ReallocForFCC()
{
    if (NDIM == 2)
    {
        perror("Sorry, we only provide 3 dimensional method.");
    }
    int n = 0;
    double x = 0.0, y = 0.0, z = 0.0;
    int yNumber = 1, zNumber = 1;
    while (z < LENGTH + 0.001)
    {
        while (y < LENGTH + 0.001)
        {
            while (x < LENGTH + 0.001)
            {
                x += diameter * sqrt(2);
                n += 1;
            }

            y += diameter * sqrt(2) / 2;
            yNumber += 1;
            if (!(((yNumber % 2 == 0) && !(zNumber % 2 == 0)) || (!(yNumber % 2 == 0) && (zNumber % 2 == 0))))
            {
                x = 0;
            }
            else
            {
                x = diameter * sqrt(2) / 2;
            }
        }

        z += diameter * sqrt(2) / 2;
        zNumber += 1;
        yNumber = 1;
        y = 0;
        if (!(((yNumber % 2 == 0) && !(zNumber % 2 == 0)) || (!(yNumber % 2 == 0) && (zNumber % 2 == 0))))
        {
            x = 0;
        }
        else
        {
            x = diameter * sqrt(2) / 2;
        }
    }

    N = n;

    double(*temp)[NDIM] = realloc(r, N * sizeof(*r));
    if (temp == NULL)
    {
        free(r);
```

```c
        perror("Memory reallocation failed.");
    }
    r = temp;
}
```

Code snippet for generating FCC data points.

fcc_lattice.c

```c
void ConstructFCCLattice()
{
    ReallocForFCC();
    if (NDIM == 2)
    {
        perror("Sorry, we only provide 3 dimensional method.");
    }
    int n = 0;
    double x = 0.0, y = 0.0, z = 0.0;
    int yNumber = 1, zNumber = 1;
    while (z < LENGTH + 0.001)
    {
        while (y < LENGTH + 0.001)
        {
            while (x < LENGTH + 0.001)
            {
                r[n][0] = x * amplifier;
                r[n][1] = y * amplifier;
                r[n][2] = z * amplifier;
                x += diameter * sqrt(2);
                n += 1;
            }

            y += diameter * sqrt(2) / 2;
            yNumber += 1;
            if (!(((yNumber % 2 == 0) && !(zNumber % 2 == 0)) || (!(yNumber % 2 == 0) && (zNumber % 2 == 0))))
            {
                x = 0;
            }
            else
            {
                x = diameter * sqrt(2) / 2;
            }
        }

        z += diameter * sqrt(2) / 2;
        zNumber += 1;
        yNumber = 1;
        y = 0;
        if (!(((yNumber % 2 == 0) && !(zNumber % 2 == 0)) || (!(yNumber % 2 == 0) && (zNumber % 2 == 0))))
        {
            x = 0;
        }
        else
        {
            x = diameter * sqrt(2) / 2;
        }
```

```
    }

    N = n;

    for (int i = 0; i < NDIM; i++)
    {
        box[i] = LENGTH * amplifier;
    }

    WriteData();
}
```

3.3. Code snippet for Sec. 2.5. Code snippet for reading data files.

hs-nvt.c

```
void read_data(void)
{
    /*--------- Your code goes here -----------*/
    char *fileName = "coords_step0000000.dat";
    FILE *fp = fopen(fileName, "r");
    if (fp == NULL)
    {
        perror("Error opening file");
    }

    if (fscanf(fp, "%d\n", &n_particles) != 1)
    {
        perror("Error reading number of particles");
        fclose(fp);
    }

    for (int d = 0; d < NDIM; d++)
    {
        double temp;
        if (fscanf(fp, "%lf %lf\n", &temp, &box[d]) != 2)
        {
            perror("Error reading first 3 data line");
            fclose(fp);
        }
    }

    for (int n = 0; n < n_particles; n++)
    {
        double tempDiameter;
        for (int d = 0; d < NDIM; d++)
        {
            if (fscanf(fp, "%lf\t", &r[n][d]) != 1)
            {
                perror("Error reading coordination line");
                fclose(fp);
            }
        }
        if (fscanf(fp, "%lf\n", &tempDiameter) != 1)
        {
            perror("Error reading diameter");
            fclose(fp);
        }
```

```
    }

    fclose(fp);
}
```

3.4. Code snippet for Sec. 2.6.

hs-nvt.c

```
int move_particle(void)
{
    /*--------- Your code goes here -----------*/
    if (NDIM == 2)
    {
        perror("Sorry, this function only support 3-D situation.\n");
    }

    double distance = 0.0, distanceX = 0.0, distanceY = 0.0, distanceZ = 0.0;
    double randNumber = dsfmt_genrand();
    double randX = dsfmt_genrand(), randY = dsfmt_genrand(), randZ = dsfmt_genrand();

    int chosenParticle = (int)(randNumber * n_particles);

    double deltaX = delta * (2 * randX - 1);
    double deltaY = delta * (2 * randY - 1);
    double deltaZ = delta * (2 * randZ - 1);

    double newX = r[chosenParticle][0] + deltaX;
    double newY = r[chosenParticle][1] + deltaY;
    double newZ = r[chosenParticle][2] + deltaZ;

    // To implement periodic boundary condition
    if (newX < 0)
        newX += box[0];
    if (newX > box[0])
        newX -= box[0];
    if (newY < 0)
        newY += box[1];
    if (newY > box[1])
        newY -= box[1];
    if (newZ < 0)
        newZ += box[2];
    if (newZ > box[2])
        newZ -= box[2];

    r[chosenParticle][0] = newX;
    r[chosenParticle][1] = newY;
    r[chosenParticle][2] = newZ;
    return 1;
}
```

3.5. Code snippet for Sec. 2.7.

hs-nvt.c

```
int move_particle(void)
{
```

```
    /*--------- Your code goes here -----------*/
    if (NDIM == 2)
    {
        perror("Sorry, this function only support 3-D situation.\n");
    }

    double distance = 0.0, distanceX = 0.0, distanceY = 0.0, distanceZ = 0.0;
    double randNumber = dsfmt_genrand();
    double randX = dsfmt_genrand(), randY = dsfmt_genrand(), randZ = dsfmt_genrand();

    int chosenParticle = (int)(randNumber * n_particles);

    double deltaX = delta * (2 * randX - 1);
    double deltaY = delta * (2 * randY - 1);
    double deltaZ = delta * (2 * randZ - 1);

    double newX = r[chosenParticle][0] + deltaX;
    double newY = r[chosenParticle][1] + deltaY;
    double newZ = r[chosenParticle][2] + deltaZ;

    // To implement periodic boundary condition
    if (newX < 0)
        newX += box[0];
    if (newX > box[0])
        newX -= box[0];
    if (newY < 0)
        newY += box[1];
    if (newY > box[1])
        newY -= box[1];
    if (newZ < 0)
        newZ += box[2];
    if (newZ > box[2])
        newZ -= box[2];

    // To check overlapping
    for (int i = 0; i < n_particles; i++)
    {
        if (i == chosenParticle)
            continue;

        distanceX = fabs(newX - r[i][0]);
        distanceY = fabs(newY - r[i][1]);
        distanceZ = fabs(newZ - r[i][2]);

        if (distanceX > box[0] / 2)
            distanceX = box[0] - distanceX;
        if (distanceY > box[1] / 2)
            distanceY = box[1] - distanceY;
        if (distanceZ > box[2] / 2)
            distanceZ = box[2] - distanceZ;

        distance = sqrt(pow(distanceX, 2) + pow(distanceY, 2) + pow(distanceZ, 2));

```

```
        if (distance < diameter)
            return 0;
    }

    r[chosenParticle][0] = newX;
    r[chosenParticle][1] = newY;
    r[chosenParticle][2] = newZ;
    return 1;
}
```