Name: Hongbo Zhu Cooperator: Chenxu Yang

UNI: hz2629 UNI: cy2554

Course name: Evolutionary Computation & Design Automation (MECS E4510)

Instructor: Hod Lipson

Date Submitted:2019/12/23/2 PM.

Grace hours used:0

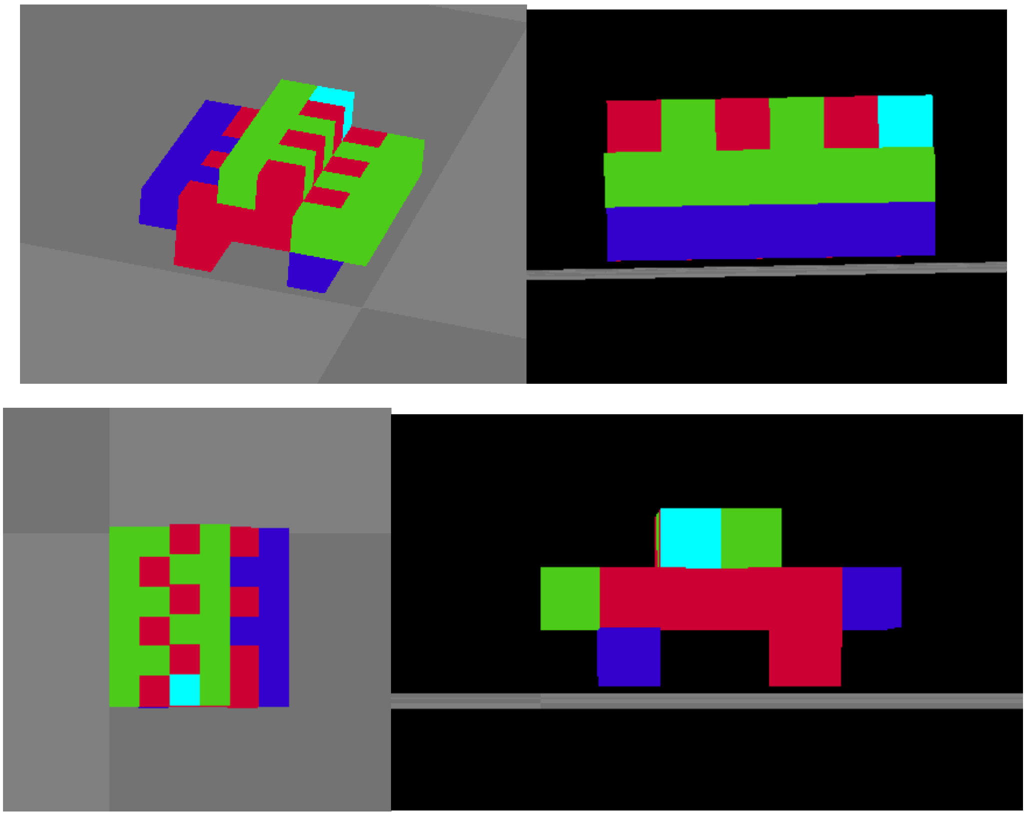
Grace hours remaining: 338.5

Chenxu Yang(197.5)

Hongbo Zhu(141)

Results summary table

Fastest robot



Speed

0.21167m/s

Video link:

Robot Running:

Link: https://drive.google.com/open?id=1VhPUOlsCbBmZrMdnYC7S1gf0JXNBR0Lv

Robot bouncing:

Link: https://drive.google.com/open?id=1mj1D3jIvwiwRgOD1XYIQOSYh3HeNuAL1

Multiple robots running:

Link: https://drive.google.com/open?id=1S8gIG-iwAJiTGTf3o7n0REmUTmNTg80n

method:

Representation :

In this assignment, we evolve a robot with a variable morphology.

We use functions to represent robot morphology, just like HW2, and we are evolving the function.

Firstly, we use a 6X6X6 big cube for evolving, each point(x,y,z) might generate a small cube according to the function, the springs of the cube is also determined by the function, there are 4 kinds of springs with different k, our goal is to evolve a robot combined by many small cubes, we use a 8_high lever tree to represent the function, exactly like hw2.

Random search:

for every time, generate a tree which represents the function, record the longest distance the robot can run in 4 seconds, keep the gene that make the robot run fastest. *Poor performance*

Hill climber:

generate a tree which represent the represents the function, for every time, change one part of the function, the simulate the robot, if the speed is faster, keep that change, else change back.

Better performance than random search, but the speed stuck at the 0.12m/s

GP:

1) generate a population which includes 40 individuals, each individual is a random tree which represent the function, the tree is also called gene.
2) Get each individual's fitness (how far it can run for 4 seconds).
3) Generate a new population:
   (1) Cross: select two individuals, randomly exchange their subtrees.
   (2) Mutation: select an individual, use the method in Hill Climber, make a small change in the tree,
   (3) Select: sort all population according to every individual's fitness, group them into 10 groups keep the top 50% of each group, eliminate the last group and put new random trees into the population
   (4) Always keep the best individual in the previous population.

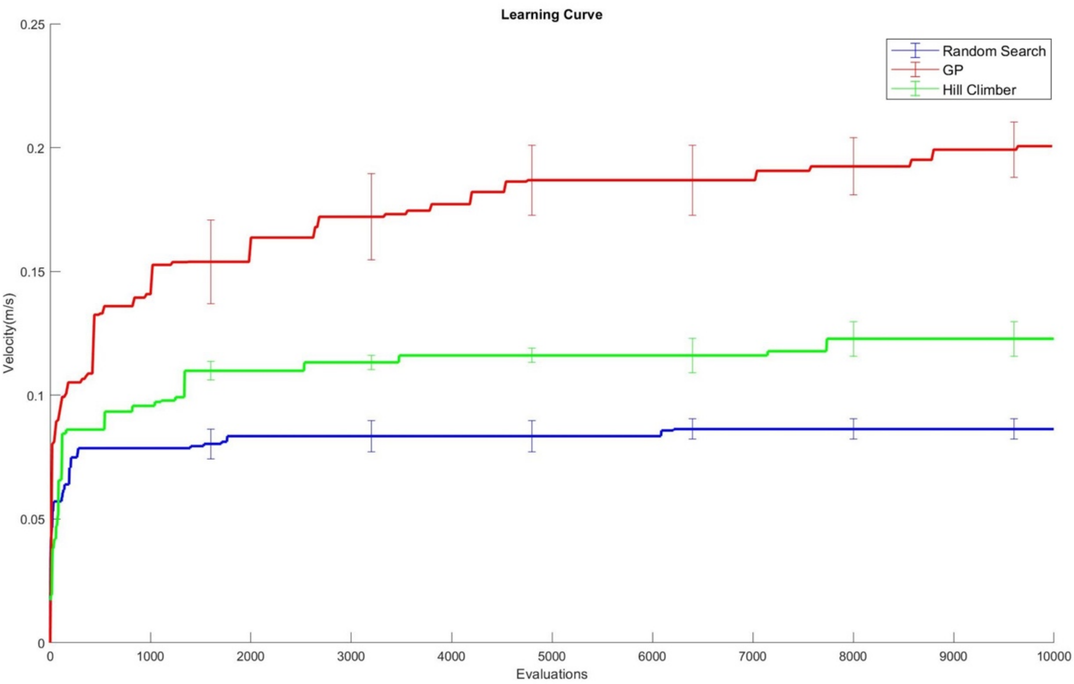
The outcome is a better than Hill climber, but cost more time to calculate

Analysis:

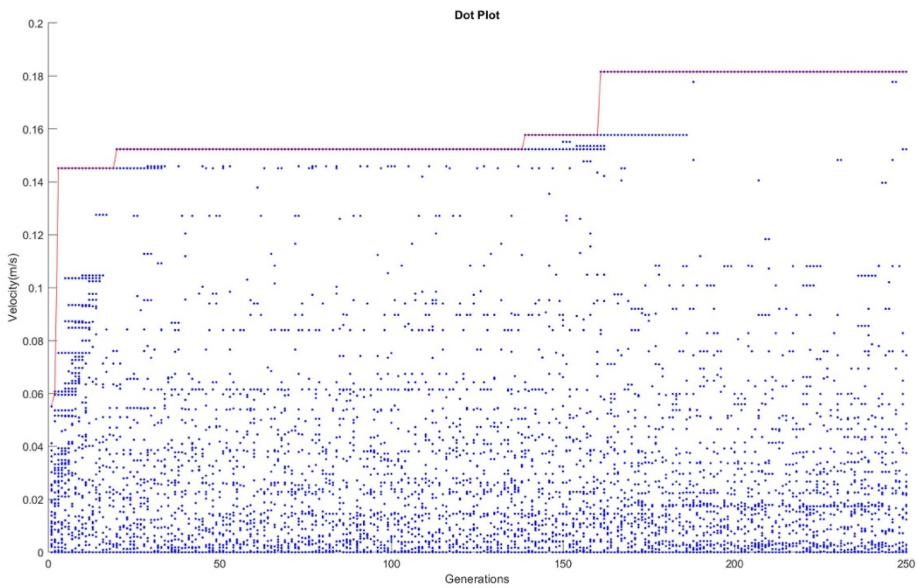
Method	evaluations	Best robot's speed
Random search	10000	0.07343m/s
Hill climber	10000	0.11225m/s
GP	10000	0.21167m/s

Performance plots

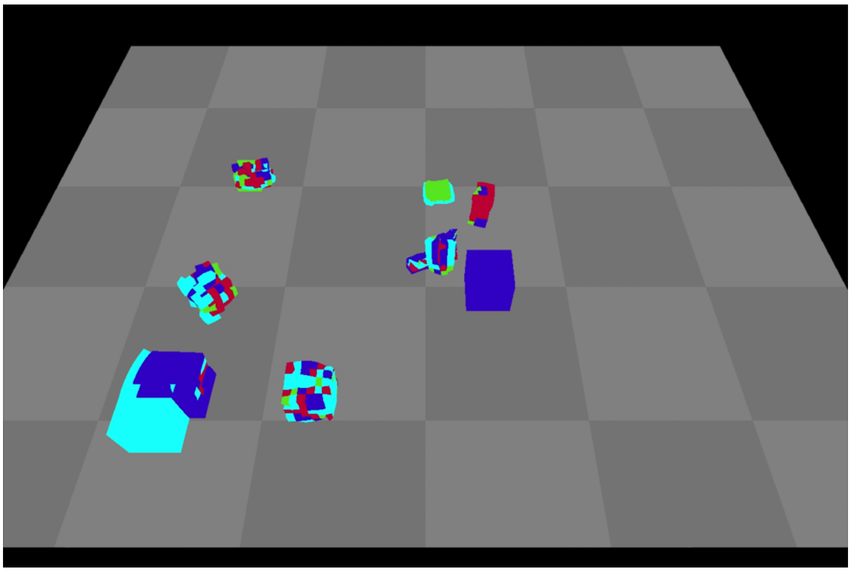
Learning curve



Dot plot



Robot Zoo



Code

```
#define _USE_MATH_DEFINES
#include <iostream>
#include <ctime>
#include <GL/freeglut.h>
#include <GL/glut.h>
#include <vector>
#include <stdarg.h>
#include <fstream>
#include "omp.h"
#include <algorithm>
#include <math.h>
#include <cmath>
#include <numeric>
#include <string>
#include <chrono>
#include <random>

#define _MATH_DEFINES_DEFINED
//#define GRAPHICS

#define LEN 8192 //  Maximum length of text string



using namespace std;



double mass = 0.2;
double length = 0.05;
double gravity = 9.81;
double T = 0;
double timestep = 0.001;
double k_c = 100000;
double damping_constant = 0.99;
double friction_coefficient = 0.8;
double W = 4 * M_PI;

int generationNumber = 1;
int robotNumber = 20;
int simulationTime = 2;

static int Layer_Number = 8;
static int Cube_Size = 6;
```

```
static double Threshold = 8.0;

float cross_rate = 1.0;
float mutation_rate = 0.1;
double maxpath = 0.0;
vector<double> Fitness;

double asp = 1;    // aspect ratio
int fov = 45;        //  Field of view (for perspective)
double dim = 1;  // size of the workd
int moveX, moveY;
int spinX = 0;
int spinY = 0;
int des = 0;
GLfloat world_rotation[16] = { 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1 };
clock_t start = clock();
double duration;
double start_time = 0.0;

double shiny = 1.0;
int mode = 1;
double th = 0.0 * M_PI / 180.0;          //  Azimuth of view angle
double ph = 45.0 * M_PI / 180.0;         //  Elevation of view angle

double view = 1000;
double viewlr = 90 * M_PI / 180.0;

static GLint Frames = 0;
static GLfloat fps = -1;
static GLint T0 = 0;
ofstream bestGene;
ofstream popDis("Best.txt");

struct Mass
{
    double m;
    double p[3];
    double v[3];
    double a[3];
    int Spring;
};

struct Spring
{
```

```
    double k;
    double l0;
    int m1;
    int m2;
    int Spring_type = 0;
    double Original;

};

struct Cube
{
    bool Is_Cube = false;
    int Cube_type = 0;
};

string operator_dic[10] = { "+","-","*","/","sin","cos","x","y","z","a" };
string calculator[4] = { "+","-","*","/" };
string tri[2] = { "sin","cos" };

double distance(Mass a, Mass b)
{
    return sqrt(pow(a.p[0] - b.p[0], 2) + pow(a.p[1] - b.p[1], 2) + pow(a.p[2] - b.p[2],
2));
}



vector<int> sort_indexes(const std::vector<double>& v)
{
    // initialize original index locations
    vector<int> idx(v.size());
    iota(idx.begin(), idx.end(), 0);
    // sort indexes based on comparing values in v
    sort(idx.begin(), idx.end(),
        [&v](int i1, int i2) {return v[i1] > v[i2]; });
    return idx;
}

class ROBOT
{
private:
    vector<string> gene;

public:
```

```
double initialLocation[3] = { 0,0,0 };
vector<Mass> Masses;
vector<Mass> M;
vector<Spring> Springs;
vector<Cube> map;
vector<Cube> map1;

int biggest_number = 0;
int biggest_index = 0;

void generate_Cube(double x, double y, double z)
{
     //string Function = To_string(1);
     //cout << Function << endl;
     Simplify();
     Generate_Masses(biggest_index, x, y, z);
     Simplify_Masses();
     double x_m = 0.0, y_m = 0.0, z_m = 0.0;
     for (int i = 0; i < Masses.size(); i++)
     {
          x_m += Masses[i].p[0];
          y_m += Masses[i].p[1];
     }
     initialLocation[0] = x_m / Masses.size();
     initialLocation[1] = y_m / Masses.size();
     initialLocation[2] = 0.0;
     Generate_Spring();


}

void Generate_Masses(int i, double a, double b, double c)
{
     if (map[i].Is_Cube)
     {
          double z = c + length / 2.0 + length * ((int)i / (Cube_Size * Cube_Size));
          double y = b + length / 2.0 + length * (int)(((int)i % (Cube_Size *
Cube_Size)) / Cube_Size);
          double x = a + length / 2.0 + length * (i % Cube_Size);

          Masses.push_back({ mass,{x + 0.5 * length,y + 0.5 * length, z + 0.5 *
length},{0,0,0},{0,0,0},map[i].Cube_type });
          Masses.push_back({ mass,{x + 0.5 * length,y + 0.5 * length, z - 0.5 *
length},{0,0,0},{0,0,0},map[i].Cube_type });
```

```
            Masses.push_back({ mass,{x + 0.5 * length,y - 0.5 * length, z + 0.5 *
length},{0,0,0},{0,0,0},map[i].Cube_type });
            Masses.push_back({ mass,{x + 0.5 * length,y - 0.5 * length, z - 0.5 *
length},{0,0,0},{0,0,0},map[i].Cube_type });
            Masses.push_back({ mass,{x - 0.5 * length,y + 0.5 * length, z + 0.5 *
length},{0,0,0},{0,0,0},map[i].Cube_type });
            Masses.push_back({ mass,{x - 0.5 * length,y + 0.5 * length, z - 0.5 *
length},{0,0,0},{0,0,0},map[i].Cube_type });
            Masses.push_back({ mass,{x - 0.5 * length,y - 0.5 * length, z + 0.5 *
length},{0,0,0},{0,0,0},map[i].Cube_type });
            Masses.push_back({ mass,{x - 0.5 * length,y - 0.5 * length, z - 0.5 *
length},{0,0,0},{0,0,0},map[i].Cube_type });
            map[i].Is_Cube = false;
        }

        if ((i + 1) % Cube_Size != 0 && map[i + 1].Is_Cube)
            Generate_Masses(i + 1, a, b, c);
        if ((i + Cube_Size) % (Cube_Size * Cube_Size) >= Cube_Size && map[i +
Cube_Size].Is_Cube)
            Generate_Masses(i + Cube_Size, a, b, c);
        if ((i + Cube_Size * Cube_Size) < Cube_Size * Cube_Size * Cube_Size && map[i +
Cube_Size * Cube_Size].Is_Cube)
            Generate_Masses(i + Cube_Size * Cube_Size, a, b, c);

        if (i - Cube_Size * Cube_Size >= 0 && map[i - Cube_Size * Cube_Size].Is_Cube)
            Generate_Masses(i - Cube_Size * Cube_Size, a, b, c);
        if ((i - 1) >= 0 && (i - 1) % Cube_Size != (Cube_Size - 1) && map[i - 1].Is_Cube)
            Generate_Masses(i - 1, a, b, c);
        if ((i - Cube_Size) >= 0 && (i - Cube_Size) % (Cube_Size * Cube_Size) <
(Cube_Size * Cube_Size - Cube_Size) && map[i - Cube_Size].Is_Cube)
            Generate_Masses(i - Cube_Size, a, b, c);
    }

    void Simplify_Masses()
    {
        M.clear();
        bool same;
        for (int i = 0; i < Masses.size(); i++)
        {
            same = true;
            for (int j = i + 1; j < Masses.size(); j++)
            {
                if (distance(Masses[i], Masses[j]) < 0.0001)
                {
```

```
                    same = false;
                    break;
                }
            }
            if (same)
            {
                M.push_back(Masses[i]);
            }

        }
        Masses = M;
    }

    void Generate_Spring()
    {
        for (int i = 0; i < Masses.size(); i++)
        {
            for (int j = i + 1; j < Masses.size(); j++)
            {
                if (distance(Masses[i], Masses[j]) < (1.8 * length))
                {
                    if (Masses[i].Spring == Masses[j].Spring)
                    {
                        if (Masses[i].Spring == 1)
                            Springs.push_back({  4000,distance(Masses[i],Masses[j]),
i, j, 1, distance(Masses[i],Masses[j]) }); //Active, Compress then expand
                        else if (Masses[i].Spring == 2)
                            Springs.push_back({  4000,distance(Masses[i],Masses[j]),
i, j, 2, distance(Masses[i],Masses[j]) }); //Active, Expand then Compress
                        else if (Masses[i].Spring == 3)
                            Springs.push_back({  4000,distance(Masses[i],Masses[j]),
i, j, 3, distance(Masses[i],Masses[j]) }); //Passive
                        else if (Masses[i].Spring == 4)
                            Springs.push_back({  8000,distance(Masses[i],Masses[j]),
i, j, 4, distance(Masses[i],Masses[j]) }); //Passive
                    }
                    else
                    {
                        if (Masses[i].Spring == 1 || Masses[j].Spring == 1)
                            Springs.push_back({  4000,distance(Masses[i],Masses[j]),
i, j, 1, distance(Masses[i],Masses[j]) });
                        else
                        {
                            if (Masses[i].Spring == 2 || Masses[j].Spring == 2)
```

```
    Springs.push_back({     4000,distance(Masses[i],Masses[j]),     i,     j,     2,
distance(Masses[i],Masses[j]) });
                              else if (Masses[i].Spring == 3 || Masses[i].Spring == 4)

    Springs.push_back({     6000,distance(Masses[i],Masses[j]),     i,     j,     3,
distance(Masses[i],Masses[j]) });
                          }
                      }
                  }
              }
          }
      }

    vector<Cube> Is_Cube()
    {
        vector<Cube> draw_cube((int)pow(Cube_Size, 3));
        vector<double> x;
        vector<double> y;
        vector<double> z;

        double j = -Cube_Size / 2 + 0.5;
        for (int i = 0; i < Cube_Size; i++)
        {
            x.push_back(j);
            y.push_back(j);
            z.push_back(j);
            j += 1.0;
        }

        int n = 0;
        for (int i = 0; i < Cube_Size; i++)
        {
            for (int j = 0; j < Cube_Size; j++)
            {
                for (int k = 0; k < Cube_Size; k++)
                {
                    double reference = 10.0 * exp(-pow(Calculate(x[i], y[j], z[k]),
2) / 2.0);
                    //double reference = Calculate(x[i], y[j], z[k]);
                    if (reference >= 0.0 && reference <= Threshold)
                    {

                        draw_cube[n].Is_Cube = true;
```

```
                    if ((int)(reference / 0.05) % 4 == 0)
                        draw_cube[n].Cube_type = 4;
                    else if ((int)(reference / 0.05) % 4 == 1)
                        draw_cube[n].Cube_type = 3;
                    else if ((int)(reference / 0.05) % 4 == 2)
                        draw_cube[n].Cube_type = 2;
                    else
                        draw_cube[n].Cube_type = 1;
                }
                n++;
            }
        }
    }

    bool no = true;
    for (int i = 0; i < draw_cube.size(); i++)
    {
        if (draw_cube[i].Is_Cube)
        {
            no = false;
        }
    }
    if (no)
    {
        for (int i = 0; i < draw_cube.size(); i++)
        {
            draw_cube[i].Is_Cube = true;
            draw_cube[i].Cube_type = 4;
        }
    }

    return draw_cube;
}

void Simplify()
{
    map.clear();
    map = Is_Cube();
    map1.clear();
    map1 = map;
    biggest_number = 0;
    biggest_index = 0;
    int biggest0 = 0;
```

```
        for (int i = 0; i < map.size(); i++)
        {
            if (map[i].Is_Cube)
            {
                Biggest_Cube(i);
            }
            if (biggest0 < biggest_number)
            {
                biggest0 = biggest_number;
                biggest_index = i;
            }
            biggest_number = 0;
        }

        New_Cube(biggest_index);

    }

    void Biggest_Cube(int i)
    {

        if (map[i].Is_Cube)
        {
            biggest_number++;
            map[i].Is_Cube = false;
        }
        if ((i + 1) % Cube_Size != 0 && map[i + 1].Is_Cube)
            Biggest_Cube(i + 1);
        if ((i + Cube_Size) % (Cube_Size * Cube_Size) >= Cube_Size && map[i +
Cube_Size].Is_Cube)
            Biggest_Cube(i + Cube_Size);
        if (i + Cube_Size * Cube_Size < Cube_Size * Cube_Size * Cube_Size && map[i +
Cube_Size * Cube_Size].Is_Cube)
            Biggest_Cube(i + Cube_Size * Cube_Size);

        if (i - Cube_Size * Cube_Size >= 0 && map[i - Cube_Size * Cube_Size].Is_Cube)
            Biggest_Cube(i - Cube_Size * Cube_Size);
        if ((i - 1) >= 0 && (i - 1) % Cube_Size != (Cube_Size - 1) && map[i - 1].Is_Cube)
            Biggest_Cube(i - 1);
        if ((i - Cube_Size) >= 0 && (i - Cube_Size) % (Cube_Size * Cube_Size) <
(Cube_Size * Cube_Size - Cube_Size) && map[i - Cube_Size].Is_Cube)
            Biggest_Cube(i - Cube_Size);
    }
```

```
void New_Cube(int i)
{
    if (map1[i].Is_Cube)
    {
        map[i].Is_Cube = true;
        map1[i].Is_Cube = false;
    }

    if ((i + 1) % Cube_Size != 0 && map1[i + 1].Is_Cube)
        New_Cube(i + 1);
    if ((i + Cube_Size) % (Cube_Size * Cube_Size) >= Cube_Size && map1[i + Cube_Size].Is_Cube)
        New_Cube(i + Cube_Size);
    if (i + Cube_Size * Cube_Size < Cube_Size * Cube_Size * Cube_Size && map1[i + Cube_Size * Cube_Size].Is_Cube)
        New_Cube(i + Cube_Size * Cube_Size);

    if (i - Cube_Size * Cube_Size >= 0 && map1[i - Cube_Size * Cube_Size].Is_Cube)
        New_Cube(i - Cube_Size * Cube_Size);
    if ((i - 1) >= 0 && (i - 1) % Cube_Size != (Cube_Size - 1) && map1[i - 1].Is_Cube)
        New_Cube(i - 1);
    if ((i - Cube_Size) >= 0 && (i - Cube_Size) % (Cube_Size * Cube_Size) < (Cube_Size * Cube_Size - Cube_Size) && map1[i - Cube_Size].Is_Cube)
        New_Cube(i - Cube_Size);
}

string To_string(int x)
{
    string function = "\0";
    if (((int)(2 * x) >= (int)pow(2, Layer_Number)) || (gene[2 * x] == (string)"0"))
        function += gene[x];
    else if (gene[x] == tri[0] || gene[x] == tri[1])
        function = function + gene[x] + (string)"(" + To_string(2 * x) + (string)")";
    else
        function = function + (string)"(" + To_string(2 * x) + gene[x] + To_string(2 * x + 1) + (string)")";
    return function;
}
double Calculate(double x, double y, double z)
{
    vector<double> result(gene.size());

    for (int i = gene.size() - 1; i > 0; i--)
    {
```

```
        if (gene[i] == "x")
            result[i] = x;
        else if (gene[i] == "y")
            result[i] = y;
        else if (gene[i] == "z")
            result[i] = z;
        else if (gene[i] == "+" && i < pow(2, Layer_Number - 1))
            result[i] = result[2 * i] + result[2 * i + 1];

        else if (gene[i] == "-" && i < pow(2, Layer_Number - 1))
            result[i] = result[2 * i] - result[2 * i + 1];

        else if (gene[i] == "*" && i < pow(2, Layer_Number - 1))
            result[i] = result[2 * i] * result[2 * i + 1];

        else if (gene[i] == "/" && i < pow(2, Layer_Number - 1))
            result[i] = result[2 * i] / result[2 * i + 1];

        else if (gene[i] == "sin" && i < pow(2, Layer_Number - 1))
            result[i] = sin(result[2 * i]);

        else if (gene[i] == "cos" && i < pow(2, Layer_Number - 1))
            result[i] = cos(result[2 * i]);

        else if (gene[i] == "0")
            continue;
        else
            result[i] = atof(gene[i].c_str());
    }
    return result[1];

}

ROBOT(double X, double Y, double Z, vector<string> Tree)
{
    // default constructor
    initialLocation[0] = X; initialLocation[1] = Y; initialLocation[2] = Z;

    gene = Tree;
    generate_Cube(X, Y, Z);
    //generate_Springs();
}
```

```
void draw_cube()
{
    GLfloat color[6][3] = { {1.0,0.0,0.0},{0.0,1.0,0.0},{0.0,0.0,1.0},
                            {1.0,1.0,0.0},{1.0,0.0,1.0},{0.0,1.0,1.0} };
    GLUquadric* quad;
    quad = gluNewQuadric();
    for (int i = 0; i < (int)Masses.size(); i++)
    {
        glPushMatrix();
        glMultMatrixf(world_rotation);
        glColor3f(1, 0, 1);
        glTranslated(Masses[i].p[0], Masses[i].p[1], Masses[i].p[2]);
        gluSphere(quad, 0.005, 20, 20);
        glPopMatrix();
    }
    for (int i = 0; i < (int)Springs.size(); i++)
    {
        glPushMatrix();
        glMultMatrixf(world_rotation);
        glLineWidth(2.0f);
        glBegin(GL_LINES);
        glColor3d(1, 1, 1);
        glVertex3f(Masses[Springs[i].m1].p[0],        Masses[Springs[i].m1].p[1],
Masses[Springs[i].m1].p[2]);
        glVertex3f(Masses[Springs[i].m2].p[0],        Masses[Springs[i].m2].p[1],
Masses[Springs[i].m2].p[2]);
        glEnd();
        glPopMatrix();
    }

    // draw line between middle point and initial position
    double x = 0; double y = 0; double z = 0;
    for (int j = 0; j < Masses.size(); j++)
    {
        x = x + Masses[j].p[0];
        y = y + Masses[j].p[1];
        z = z + Masses[j].p[2];
    }
    x = x / Masses.size();
    y = y / Masses.size();
    z = z / Masses.size();
    glPushMatrix();
    glMultMatrixf(world_rotation);
```

```
        glBegin(GL_LINES);
        glColor3f(0.0, 1.0, 0.0);
        glVertex3f(GLfloat(x), GLfloat(y), 0.0);
        glVertex3f(GLfloat(initialLocation[0]),          GLfloat(initialLocation[1]),
GLfloat(0.0));
        glEnd();
        glPopMatrix();
        double distance = sqrt(pow(x, 2) + pow(y, 2));
        //printf("Time: %f, Distance: %f\n", T, distance);
    }

    void robotUpdate()
    {
        bool move = true;

        vector<vector<double>> forces(Masses.size(), vector<double>(3));


        for (int i = 0; i < Springs.size(); i++)
        {
            if (Springs[i].Spring_type == 1)
            {
                Springs[i].l0  =  Springs[i].Original  +  0.4  *  Springs[i].Original  *
sin(W * T);
            }
            else if (Springs[i].Spring_type == 2)
            {
                Springs[i].l0  =  Springs[i].Original  +  0.4  *  Springs[i].Original  *
cos(W * T);
            }
        }

        for (int i = 0; i < Springs.size(); i++)
        {
            Mass mass1 = Masses[Springs[i].m1];
            Mass mass2 = Masses[Springs[i].m2];
            double  position_distance[3]  =  {  mass2.p[0]  -  mass1.p[0],mass2.p[1]  -
mass1.p[1],mass2.p[2] - mass1.p[2] };
            double      l_now      =      sqrt(pow(position_distance[0],      2)      +
pow(position_distance[1], 2) + pow(position_distance[2], 2));
            double this_force = Springs[i].k * fabs(Springs[i].l0 - l_now);
            int flag = 1;

            if (l_now > Springs[i].l0) {
```

```
                flag = -1;
            }
            forces[Springs[i].m1][0] = forces[Springs[i].m1][0] - flag * this_force *
position_distance[0] / l_now;
            forces[Springs[i].m1][1] = forces[Springs[i].m1][1] - flag * this_force *
position_distance[1] / l_now;
            forces[Springs[i].m1][2] = forces[Springs[i].m1][2] - flag * this_force *
position_distance[2] / l_now;
            forces[Springs[i].m2][0] = forces[Springs[i].m2][0] + flag * this_force *
position_distance[0] / l_now;
            forces[Springs[i].m2][1] = forces[Springs[i].m2][1] + flag * this_force *
position_distance[1] / l_now;
            forces[Springs[i].m2][2] = forces[Springs[i].m2][2] + flag * this_force *
position_distance[2] / l_now;
        }

        for (int i = 0; i < Masses.size(); i++)
        {
            forces[i][2] = forces[i][2] - Masses[i].m * gravity;
            if (Masses[i].p[2] <= 0)
            {
                forces[i][2] = forces[i][2] + k_c * fabs(Masses[i].p[2]);
                double F_H = sqrt(pow(forces[i][0], 2) + pow(forces[i][1], 2));
                double F_V = forces[i][2];
                if (abs(F_H) < abs(F_V) * friction_coefficient)
                {
                    forces[i][0] = 0;
                    forces[i][1] = 0;
                    Masses[i].v[0] = 0;
                    Masses[i].v[1] = 0;
                }
                /*
                else
                {
                    for (int j = 0; j < 2; j++)
                    {
                        if (forces[i][j] < 0)
                        {
                            forces[i][j]    =    forces[i][j]    +    abs(F_V)    *
friction_coefficient * abs(forces[i][j] / F_H);
                            if (forces[i][j] > 0)
                                forces[i][j] = 0;
                        }
                        else
```

```
                        {
                            forces[i][j]    =    forces[i][j]    -    abs(F_V)    *
friction_coefficient * abs(forces[i][j] / F_H);
                            if (forces[i][j] < 0)
                                forces[i][j] = 0;
                        }
                    }
                }*/

            }
            for (int j = 0; j < 3; j++) {
                Masses[i].a[j] = forces[i][j] / Masses[i].m;
                Masses[i].v[j] = damping_constant * (Masses[i].v[j] + Masses[i].a[j]
* timestep);
                Masses[i].p[j] = Masses[i].p[j] + Masses[i].v[j] * timestep;
            }
            //double velocity = sqrt(pow(Masses[i].v[0], 2) + pow(Masses[i].v[1], 2) +
pow(Masses[i].v[2], 2));
            //cout << "v:" << velocity << endl;

        };
    }
};

class Simulation {
private:
    int populationSize;
    vector<double> populationDistance;
    vector<vector<string>> populationGene;
    vector<vector<string>> newPopulationGene;
    vector<vector<string>> maxgene;
    vector<string> Cage;
    vector<ROBOT> robots;
    vector<int> index;
public:
    double averageDistance;
    double maxDistance;

    Simulation(int popSize)
    {
        populationSize = popSize;
        generateGenes(populationSize);
        generateRobots();
    }
```

```
    void startSim(double time) {
        if (T < time) {
            simUpdate();
#ifdef  GRAPHICS
            simDraw();
#endif
        }
        else {

            printf("### Generation %d ###", generationNumber);
            double time = omp_get_wtime() - start_time;
            printf(" Time: %f ###\n", time);
            calculatePopulationDistance();
            selection();
            crossOver();
            mutation();
            populationGene.clear();
            populationGene.shrink_to_fit();
            populationGene = newPopulationGene;
            generationNumber++;
            robots.clear();
            robots.shrink_to_fit();
            generateRobots();
            T = 0;
            start_time = omp_get_wtime();
        }
    }

    void selection()
    {
        index = sort_indexes(populationDistance);
        newPopulationGene.clear();
        newPopulationGene.shrink_to_fit();
        if (populationDistance[index[0]] > maxpath)
        {
            maxpath = populationDistance[index[0]];
            maxgene.clear();
            maxgene.push_back(populationGene[index[0]]);
        }

        for (int i = 0; i < populationDistance.size(); i++)
        {
            popDis << populationDistance[i] / 2.0 << endl;
```

```
}

cout << To_string(1) << endl;

double all = 0.0;
for (int i = 0; i < index.size(); i++)
{
    all += populationDistance[i];
}

for (int i = 0; i < index.size() / 2 - 1; i++)
{
    double b = 0.0;
    int a = (double)rand() / (double)RAND_MAX * all;
    for (int j = 0; j < index.size(); j++)
    {
        b += populationDistance[j];
        if (b >= a)
        {
            newPopulationGene.push_back(populationGene[index[j]]);
            break;
        }
    }
}
newPopulationGene.push_back(maxgene[0]);

/*
newPopulationGene.push_back(maxgene[0]);
for (int i = 0; i < index.size() / 2 - 1; i++) {
    newPopulationGene.push_back(populationGene[index[i]]);
}*/

/*
for (int i = 0; i < (index.size() - (index.size() / 5) * 1); i++)
{
    if ((int)(i / 2) % 2 == 0)
    {
        newPopulationGene.push_back(populationGene[index[i]]);
    }
}

generateGenes(3);
for (int i = 0; i < 3; i++)
{
```

```
            newPopulationGene.push_back(populationGene[populationGene.size() - i - 1]);
        }
        newPopulationGene.push_back(maxgene[0]);
        */
    }

    void crossOver()
    {
        double Pc_;
        int x, y;

        vector<int> cross;
        cross.clear();
        for (int i = 0; i < populationSize / 2; i++)
            cross.push_back(i);
        unsigned                                seed                                =
(unsigned)chrono::system_clock::now().time_since_epoch().count();
        shuffle(cross.begin(), cross.end(), default_random_engine(seed));

        srand((int)time(0));
        for (int n = 0; n < populationSize / 2; n += 2)
        {
            Pc_ = (double)rand() / (double)RAND_MAX;
            x = cross[n];
            y = cross[n + 1];

            newPopulationGene.push_back(newPopulationGene[x]);
            newPopulationGene.push_back(newPopulationGene[y]);

            if (Pc_ < cross_rate)
            {

                int node1, node2;
                do
                {
                    do
                    {
                        node1 = rand() % (int)pow(2, Layer_Number);
                    } while (newPopulationGene[x][node1] == "0");

                    do
                    {
                        node2 = rand() % (int)pow(2, Layer_Number);
                    } while (newPopulationGene[y][node2] == "0");
```

```
                } while (IsLong(x, node1) + (int)(log(node2) / log(2)) > Layer_Number
|| IsLong(y, node2) + (int)(log(node1) / log(2)) > Layer_Number);

                Swap(node1, node2, x, y);
            }
        }

        /*
        for (int i = 0; i < newPopulationGene.size(); i++)
        {
            cout <<"Size : "<< newPopulationGene[i].size() << endl;
        }*/
    }

    void mutation()
    {
        double Pm_ = 0.0;

        int ind;
        for (ind = 0; ind < newPopulationGene.size(); ind++)
        {
            Pm_ = (double)rand() / (double)RAND_MAX;

            if (Pm_ <= mutation_rate)
            {
                while (true)
                {
                    int node = rand() % (int)pow(2, Layer_Number);
                    if (newPopulationGene[ind][node] == "0")
                        continue;
                    else
                    {
                        double x = (double)rand() / (double)RAND_MAX;
                        if (x < 1.0)
                        {
                            if ((newPopulationGene[ind][node] == "x" || \
                                newPopulationGene[ind][node] == "y" || \
                                newPopulationGene[ind][node] == "z" || \
                                newPopulationGene[ind][node] == "+" || \
                                newPopulationGene[ind][node] == "-" || \
                                newPopulationGene[ind][node] == "*" || \
                                newPopulationGene[ind][node] == "/" || \
```

```
                                newPopulationGene[ind][node] == "sin" || \
                                newPopulationGene[ind][node]  ==  "cos")  &&  node  <
pow(2, Layer_Number - 1))
                            {
                                int pin = rand() % 3;
                                int len = IsLong(ind, node);
                                if  ((pin  ==  0)  &&  (log(node)  /  log(2)  +  len  <
Layer_Number))
                                {
                                    Cage.clear();
                                    for (int i = 0; i < pow(2, Layer_Number); i++)
                                        Cage.push_back("0");
                                    Save(ind, node);
                                    Clear(ind, node);
                                    Copy(ind, 2 * node, node);
                                    newPopulationGene[ind][node] = "+";
                                    char buffer[20];
                                    sprintf_s(buffer,  "%.10f",  ((double)rand()  /
(double)RAND_MAX - 0.5) * 0.1);
                                    string str = buffer;
                                    newPopulationGene[ind][2 * node + 1] = str;
                                }
                                else if ((pin == 1) && (log(node) / log(2) + len <
Layer_Number))
                                {
                                    Cage.clear();
                                    for (int i = 0; i < pow(2, Layer_Number); i++)
                                        Cage.push_back("0");
                                    Save(ind, node);
                                    Clear(ind, node);
                                    Copy(ind, 2 * node, node);
                                    newPopulationGene[ind][node] = "*";
                                    char buffer[20];
                                    sprintf_s(buffer,  "%.10f",  ((double)rand()  /
(double)RAND_MAX - 0.5) * 0.1 + 1);
                                    string str = buffer;
                                    newPopulationGene[ind][2 * node + 1] = str;
                                }
                                else if (pin == 2)
                                {
                                    Clear(ind, node);
                                    char buffer[20];
                                    sprintf_s(buffer,  "%.10f",  (double)rand()  /
(double)RAND_MAX * 20.0 - 10.0);
```

```
                    string str = buffer;
                    newPopulationGene[ind][node] = str;
                }
            }
            else
            {
                double x = (double)rand() / (double)RAND_MAX;
                if (x < 0.6)
                {
                    double                   c                    =
atof(newPopulationGene[ind][node].c_str());
                    if ((double)rand() / (double)RAND_MAX >= 0.5)
                       c += (double)rand() / (double)RAND_MAX * 0.5;
                    else
                       c -= (double)rand() / (double)RAND_MAX * 0.5;
                    if (c > 10.0)
                        c -= 10.0;
                    else if (c < -10.0)
                        c += 10.0;

                    char buffer[20];
                    sprintf_s(buffer, "%.10f", c);
                    string str = buffer;
                    newPopulationGene[ind][node] = str;
                }
                else
                {
                    int b = rand() % 3;
                    if (b == 0)
                        newPopulationGene[ind][node] = "x";
                    else if (b == 1)
                        newPopulationGene[ind][node] = "y";
                    else
                        newPopulationGene[ind][node] = "z";
                }

            }
        }
        else
        {
            if ((newPopulationGene[ind][node] == "+" \
                || newPopulationGene[ind][node] == "-" \
                || newPopulationGene[ind][node] == "*" \
                || newPopulationGene[ind][node] == "/" \
```

```
                                        || newPopulationGene[ind][node] == "sin" \
                                        || newPopulationGene[ind][node] == "cos") && node <
pow(2, Layer_Number - 1))
                                    {
                                        int r2 = rand() % 10;
                                        if (r2 < 4 && (newPopulationGene[ind][node] == "sin"
|| newPopulationGene[ind][node] == "cos"))
                                        {
                                            int b = rand() % 3;
                                            if (b == 0)
                                                newPopulationGene[ind][2 * node + 1] = "x";
                                            else if (b == 1)
                                                newPopulationGene[ind][2 * node + 1] = "y";
                                            else
                                                newPopulationGene[ind][2 * node + 1] = "z";

                                           newPopulationGene[ind][node] = operator_dic[r2];
                                        }
                                        else   if   (r2   >=   4   &&   r2   <=   5   &&
(newPopulationGene[ind][node]  ==  "+"  ||  newPopulationGene[ind][node]  ==  "-"  ||
newPopulationGene[ind][node] == "*" || newPopulationGene[ind][node] == "/"))
                                        {
                                            Clear(ind, 2 * node + 1);
                                           newPopulationGene[ind][node] = operator_dic[r2];
                                        }

                                        else if (r2 == 6 || r2 == 7 || r2 == 8)
                                        {
                                            if (node > 1 && node < pow(2, Layer_Number))
                                            {
                                                Clear(ind, node);
                                                newPopulationGene[ind][node]              =
operator_dic[r2];

                                            }
                                        }
                                        else if (r2 == 9)
                                        {
                                            continue;

                                            if (node > 1 && node < pow(2, Layer_Number))
                                            {
                                                Clear(ind, node);
                                                char buffer[20];
                                                sprintf_s(buffer, "%.10f", (double)rand() /
```

```
(double)RAND_MAX * 20.0 - 10.0);
                                                string str = buffer;
                                                newPopulationGene[ind][node] = str;
                                            }

                                        }
                                    }
                                }
                                break;
                            }
                        }
                    }
                }
            }

            void Clear(int i, int x)
            {
                newPopulationGene[i][x] = "0";
                if ((int)(2 * x) < (int)pow(2, Layer_Number))
                {
                    Clear(i, 2 * x);
                    Clear(i, 2 * x + 1);
                }
            }

            void Copy(int x, int node1, int node2)
            {
                newPopulationGene[x][node1] = Cage[node2];
                if (2 * max(node1, node2) < (int)pow(2, Layer_Number))
                {
                    Copy(x, 2 * node1, 2 * node2);
                    Copy(x, 2 * node1 + 1, 2 * node2 + 1);
                }
            }

            void Save(int i, int x)
            {
                Cage[x] = newPopulationGene[i][x];
                if (2 * x < (int)pow(2, Layer_Number))
                {
                    Save(i, 2 * x);
                    Save(i, 2 * x + 1);
                }
            }
```

```
void Swap(int index1, int index2, int ind1, int ind2)
{
    swap(newPopulationGene[ind1][index1], newPopulationGene[ind2][index2]);

    if (index1 * 2 < (int)pow(2, Layer_Number) && index2 * 2 < (int)pow(2, Layer_Number))
    {
        Swap(index1 * 2, index2 * 2, ind1, ind2);
        Swap(index1 * 2 + 1, index2 * 2 + 1, ind1, ind2);
    }
}

int IsLong(int Ind, int node)
{
    int i = 1;
    int len = 1;
    while (node * pow(2, i) < pow(2, Layer_Number))
    {
        int j = node * (int)pow(2, i);
        int k = 1;
        while (k < (int)pow(2, i))
        {
            if (newPopulationGene[Ind][j] != "0")
            {
                len++;
                break;
            }
            j++;
            k++;
        }
        i++;
    }
    return len;
}

void generateGenes(int num)
{
    srand(time(0));
    vector<string> Tree;
    for (int i = 0; i < num; i++)
    {
        Tree.clear();
        for (int j = 0; j < pow(2, Layer_Number); j++)
```

```
                Tree.push_back("0");

            Tree[1] = operator_dic[rand() % 6];
            for (int i = 1; i < Layer_Number - 1; i++)
            {
                for (int j = (int)pow(2, i); j < (int)pow(2, i + 1); j++)
                {
                    for (int k = 0; k < 4; k++)
                    {
                        if (Tree[int(j / 2.0)] == calculator[k])
                        {
                            Tree[j] = operator_dic[rand() % 10];
                            break;
                        }
                    }
                    if (Tree[int(j / 2)] == tri[0] || Tree[int(j / 2)] == tri[1])
                    {
                        if (j % 2 == 0)
                        {

                            Tree[j] = operator_dic[rand() % 10];
                        }
                    }
                }
            }



            for (int i = (int)pow(2, Layer_Number - 1); i < (int)pow(2, Layer_Number);
i++)
            {
                if (Tree[int(i / 2)] != "0" && Tree[int(i / 2)] != "x" && Tree[int(i
/ 2)] != "y" && Tree[int(i / 2)] != "z" && Tree[int(i / 2)] != (string)"a")
                {
                    if (Tree[int(i / 2)] == "cos" || Tree[int(i / 2)] == "sin")
                    {
                        if (i % 2 == 0)
                        {
                            Tree[i] = operator_dic[6 + rand() % 4];
                        }
                    }
                    else
                    {
                        Tree[i] = operator_dic[6 + rand() % 4];
                    }
```

```
                }
            }

            for (int i = 0; i < pow(2, Layer_Number); i++)
            {
                if (Tree[i] == "a")
                {
                    char buffer[20];
                    sprintf_s(buffer, "%.10f", (double)rand() / (double)RAND_MAX *
20.0 - 10.0);
                    string str = buffer;
                    Tree[i] = str;
                }
            }
            populationGene.push_back(Tree);
        }
    }

    void generateRobots()
    {
        for (int i = 0; i < populationSize; i++) {
            double X = 2.0 * (double)rand() / (double)RAND_MAX;
            double Y = 2.0 * (double)rand() / (double)RAND_MAX;
            //double X = 0.0;
            //double Y = 0.0;
            robots.push_back(ROBOT(X, Y, 0.0, populationGene[i]));
        }
    }

    void simUpdate()
    {
#pragma omp parallel for num_threads(16)
        for (int i = 0; i < populationSize; i++)
            robots[i].robotUpdate();
    }

    void simDraw()
    {
        for (int i = 0; i < populationSize; i++)
            robots[i].draw_cube();
    }

    void calculatePopulationDistance()
    {
```

```
        populationDistance.clear();
        populationDistance.shrink_to_fit();
        for (int i = 0; i < populationSize; i++) {
            double x = 0; double y = 0;
            for (int j = 0; j < robots[i].Masses.size(); j++) {
                x = x + robots[i].Masses[j].p[0];
                y = y + robots[i].Masses[j].p[1];
            }
            x = x / robots[i].Masses.size();
            y = y / robots[i].Masses.size();
            double distance = fabs(x - robots[i].initialLocation[0]);
            populationDistance.push_back(distance);
        }
        averageDistance = 0;
        maxDistance = 0;
        for (int i = 0; i < populationSize; i++)
        {
            averageDistance = averageDistance + populationDistance[i];
            if (maxDistance < populationDistance[i])
            {
                maxDistance = populationDistance[i];
            }
        }

        averageDistance = averageDistance / populationSize;
        cout << "Maximum Velocity: " << maxDistance << endl;
        cout << "Average Velocity: " << averageDistance << endl;
    }

    string To_string(int x)
    {
        string function = "\0";
        if ((2 * x >= pow(2, Layer_Number)) || (maxgene[0][2 * x] == (string)"0"))
            function += maxgene[0][x];
        else if (maxgene[0][x] == tri[0] || maxgene[0][x] == tri[1])
            function = function + maxgene[0][x] + (string)"(" + To_string(2 * x) +
(string)")";
        else
            function = function + (string)"(" + To_string(2 * x) + maxgene[0][x] +
To_string(2 * x + 1) + (string)")";
        return function;
    }
};
```

```
Simulation sim1(robotNumber);



#if 1

void Print(const char* format, ...)
{
    char buf[LEN];
    char* ch = buf;
    va_list args;
    //  Turn the parameters into a character string
    va_start(args, format);
    vsnprintf(buf, LEN, format, args);
    va_end(args);
    //  Display the characters one at a time at the current raster position
    while (*ch)
        glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, *ch++);
}

void drawGrid() {
    for (int i = -dim / 2; i < dim / 2 + 1; i++) {
        for (int j = -dim / 2; j < dim / 2 + 1; j++) {
            float white[] = { 1,1,1,1 };
            float black[] = { 0,0,0,1 };
            glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, shiny);
            glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, white);
            glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, black);

            glPushMatrix();
            glTranslatef(i * 2, 0, j * 2);
            glBegin(GL_QUADS);
            //
            glNormal3f(0, 1, 0);
            glColor3f(0.45, 0.45, 0.45);
            glVertex3f(+0, -0.01, +0);
            glVertex3f(+1, -0.01, +0);
            glVertex3f(+1, -0.01, +1);
            glVertex3f(+0, -0.01, +1);
            //
            glNormal3f(0, 1, 0);
            glColor3f(0.5, 0.5, 0.5);
            glVertex3f(-1, -0.01, +0);
            glVertex3f(+0, -0.01, +0);
            glVertex3f(+0, -0.01, +1);
```

```
            glVertex3f(-1, -0.01, +1);
            //
            glNormal3f(0, 1, 0);
            glColor3f(0.45, 0.45, 0.45);
            glVertex3f(-1, -0.01, -1);
            glVertex3f(+0, -0.01, -1);
            glVertex3f(+0, -0.01, +0);
            glVertex3f(-1, -0.01, +0);
            //
            glNormal3f(0, 1, 0);
            glColor3f(0.5, 0.5, 0.5);
            glVertex3f(-0, -0.01, -1);
            glVertex3f(+1, -0.01, -1);
            glVertex3f(+1, -0.01, +0);
            glVertex3f(-0, -0.01, +0);
            glEnd();
            glPopMatrix();
        }
    }

}


void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);
    glLoadIdentity();
    const double len = 2.0;  //  Length of axes

    double Ex = -2 * dim * sin(th) * cos(ph);
    double Ey = +2 * dim * sin(ph);
    double Ez = +2 * dim * cos(th) * cos(ph);
    gluLookAt(Ex, Ey, Ez, 0, 0, 0, 0, cos(ph), 0);
    glPushMatrix();
    glRotated(spinX, 0, 1, 0);
    glRotated(spinY, 1, 0, 0);
    glTranslated(0, 0, des);

    glDisable(GL_LIGHTING);

    drawGrid();
#ifdef GRAPHICS
    glColor3f(1, 1, 1);
```

```
    //glWindowPos2i(0, 0);
    Print("Generation: %d", generationNumber);
    //glWindowPos2i(850, 0);
    Print("Max: %4.2f", sim1.maxDistance);
    //glWindowPos2i(850, 50);
    Print("Average %4.2f", sim1.averageDistance);
#endif 1
    sim1.startSim(simulationTime);
    T = T + timestep;

    Frames++;
    GLint t = glutGet(GLUT_ELAPSED_TIME);
    if (t - T0 >= 1000) {
        GLfloat seconds = ((double)t - T0) / 1000.0;
        fps = Frames / seconds;
        //printf("%d frames in %6.3f seconds = %6.3f FPS\n", Frames, seconds, fps);
        T0 = t;
        Frames = 0;
    }
    glRasterPos3d(0.0, 2, 0.0);

    glColor3f(1, 1, 1);
    glPopMatrix();
    glutSwapBuffers();
}

#endif

#if 0
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);
    glLoadIdentity();
    gluLookAt(-1.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    glPushMatrix();
    glRotated(spinX, 0, 1, 0);
    glRotated(spinY, 1, 0, 0);
    glTranslated(0, 0, des);
    for (int i = 0; i < cube_numbers; i++) {
        Update_cube(cubes[i].Masses, cubes[i].Springs);
    }
    glPopMatrix();
    glutSwapBuffers();
```

```
}
#endif
void Project(double fov, double asp, double dim)
{

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (fov)
        gluPerspective(fov, asp, dim / 1, 16 * dim);
    else
        glOrtho(-asp * dim, asp * dim, -dim, +dim, -dim, +dim);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void mouseMove(int x, int y) {
    int dx = x - moveX;
    int dy = y - moveY;
    printf("dx;%dx,dy:%dy\n", dx, dy);
    spinX += dx;
    spinY += dy;
    glutPostRedisplay();
    moveX = x;
    moveY = y;
}
void key_pressed(unsigned char ch, int x, int y)
{
    //  Exit on ESC
    if (ch == 27)
        exit(0);
    else if (ch == '0')
        th = ph = 0;
    else if (ch == '-' && ch > 1)
        fov++;
    else if (ch == '=' && ch < 179)
        fov--;
    else if (ch == GLUT_KEY_PAGE_DOWN)
        dim += 0.1;
    else if (ch == GLUT_KEY_PAGE_UP && dim > 1)
        dim -= 0.1;
    else if (ch == 'a' || ch == 'A')
        spinX += 5;
    else if (ch == 'd' || ch == 'D')
        spinX -= 5;
```

```
    else if (ch == 'w' || ch == 'W')
        spinY += 5;
    else if (ch == 's' || ch == 'S')
        spinY -= 5;
    else if (ch == 'q' || ch == 'Q')
        viewlr -= 15;
    else if (ch == 'e' || ch == 'E')
        viewlr += 15;
    //  Reproject
    Project(fov, asp, dim);
    //  Tell GLUT it is necessary to redisplay the scene
    glutPostRedisplay();
}
void reshape(int width, int height)
{
    //  Ratio of the width to the height of the window
    asp = (double)width / height;
    glViewport(0, 0, width, height);
    Project(fov, asp, dim);
}
void idle()
{
    glutPostRedisplay();
}

int main(int argc, char* argv[])
{
    srand((int)time(0));
#ifdef GRAPHICS
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(1000, 1200);
    glutCreateWindow("cubes");
    glutIdleFunc(idle);
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMotionFunc(mouseMove);
    glutKeyboardFunc(key_pressed);
    glutMainLoop();
    return 0;
#endif

#ifndef GRAPHICS
```

```
    while (1) {
        double start_time = omp_get_wtime();
        sim1.startSim(simulationTime);
        T = T + timestep;
        //printf("%f\n", T);
        double time = omp_get_wtime() - start_time;
    }
#endif
};
```