Name: Hongbo Zhu Cooperator: Chenxu Yang

UNI:hz2629 UNI:cy2554

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

Instructor: Hod Lipson

Date Submitted:2019/11/23/9 PM.

Grace hours used:0

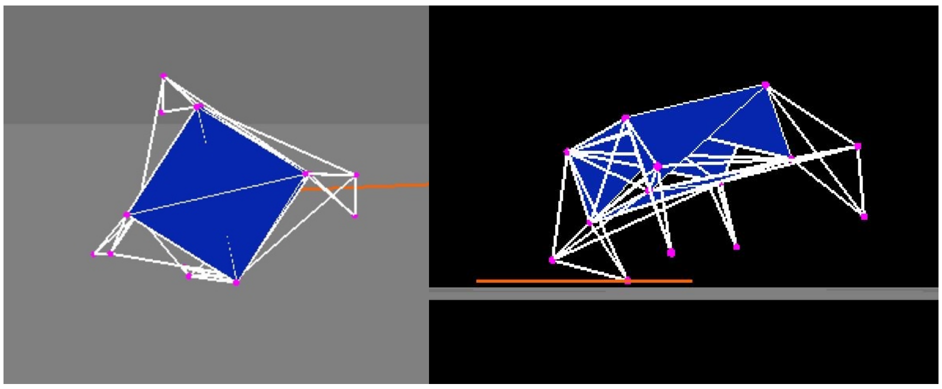
Grace hours remaining: 311.5

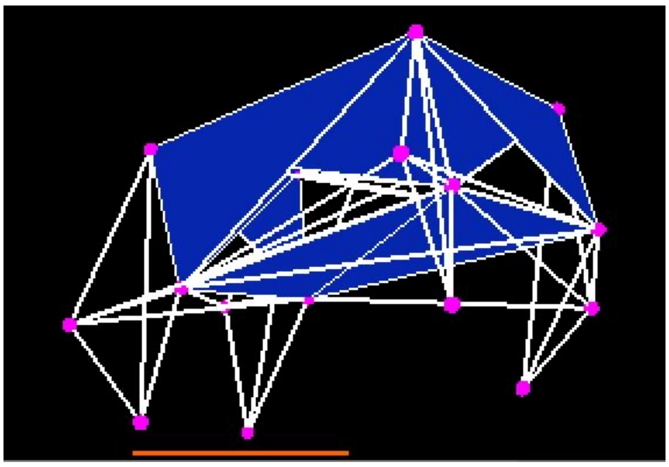
Chenxu Yang(184)

Hongbo Zhu(127.5)

Results summary table

Fastest robot





Speed

0.30865m/s

Video link:

Robot moving:
Link: https://drive.google.com/open?id=1SkuVc5BkN2WXfzK7SjUYWwfzFGwpgZ_M

Robot bouncing:
Link: https://drive.google.com/open?id=1wd6lWqv8n_loPZ-6zahhlnl_sY6SP9aK

method:

Representation :

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

Firstly, we designed a robot with four legs, each leg has 3 springs, the body part is made of 44 springs, we use different k to represent the springs, for the 12 springs of the 4 legs, we set k as 2000N/m, for the rest 44 springs, we set k as 6000N/m, our goal is to evolve the 4 legs, we use a 48_long list to represent the gene, each spring has 4 parameters which represent how the spring will change.

Random search:

for every time, generate a 48_long list which represents the parameters for all leg springs, 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 48_long list which represent the parameters for all leg springs, for every time, change one part of the gene, 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 0.19m/s

EA:

1) generate a population which includes 100 individuals, each individual is a random 48_long list which represent the parameters for all leg springs, the list 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 choose springs, and exchange their parameters.
   (2) Mutation: select an individual, use the method in Hill Climber, make a small change in the gene,
   (3) Select: sort all population according to every individual's fitness, keep the top 50% of them,
   (4) Always keep the best individual in the previous population.

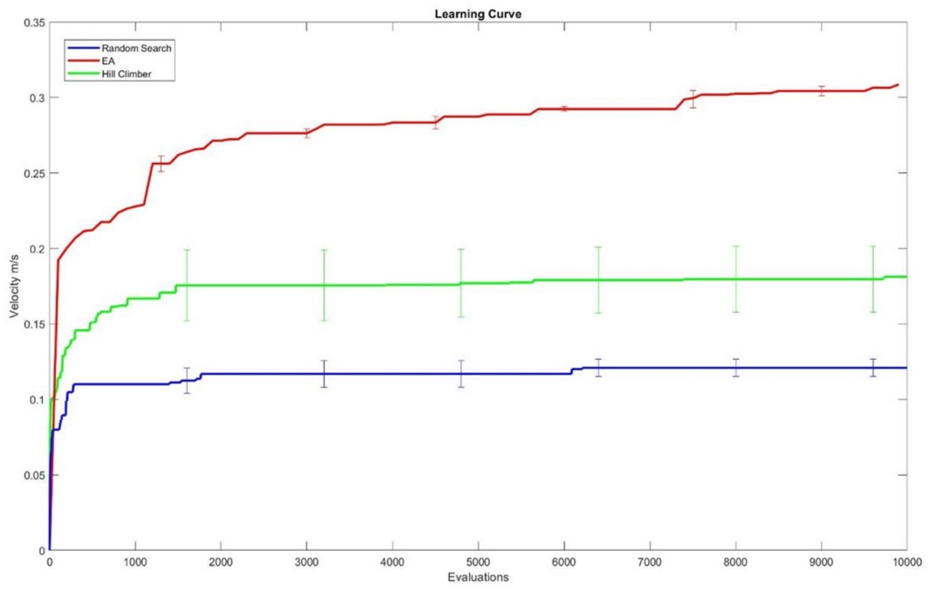
The outcome is a better than Hill climber, but cost more time to calculate

Analysis:

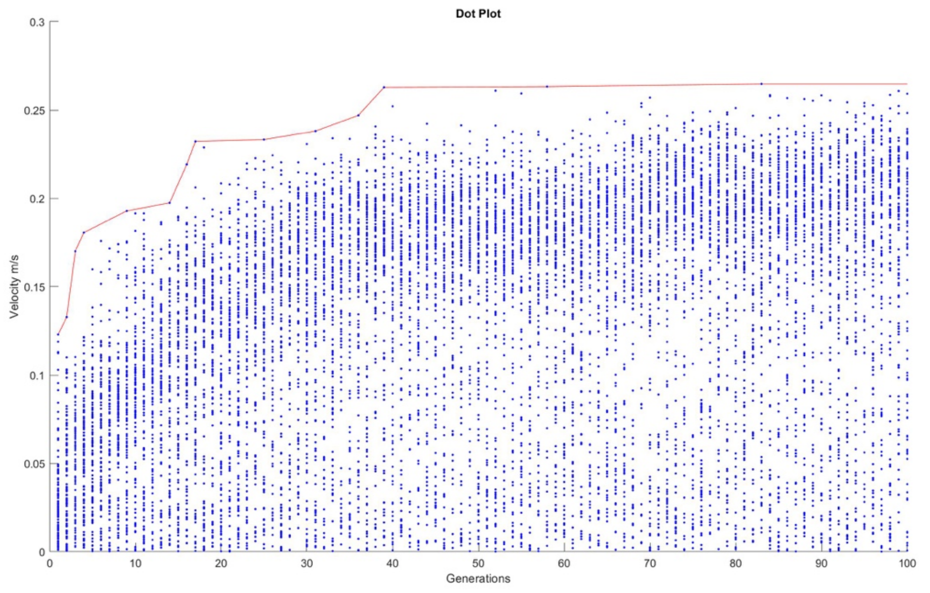
Method	evaluations	Best robot's speed
Random search	10000	0.12144m/s
Hill climber	10000	0.18985m/s
GP	10000	0.30865m/s

Performance plots

Learning curve



Dot plot



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>
#define _MATH_DEFINES_DEFINED
#define Plot



using namespace std;
int mass_number = 16;
int spring_number = 56;
double mass = 0.2;
double length = 0.1;
double gravity = 9.81;
double T = 0;
double timestep = 0.001;
double k_c = 100000;
double k = 6000;
double damping_constant = 0.99;
double friction_coefficient = 0.8;
double W = 2 * M_PI;
double OriLength[16];
int evospring = 12;

int generationNumber = 1;
int number = 100;
int simulationTime = 8;

float cross_rate = 0.6;
float mutation_rate = 0.1;
double maxpath = 0.0;
vector<double> Fitness;
```

```
double asp = 1;
int fov = 45;
double dim = 1.0;
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;
double ph = 45.0 * M_PI / 180.0;

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

static GLint Frames = 0;
static GLfloat fps = -1;
static GLint T0 = 0;

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

struct Spring
{
    double k;
    double l0;
    int m1;
    int m2;
};

struct GENE
{
    // L = L*L_0 + A*sin(W*t+B) + C*sin(W*t+D)
    double A;
    double B;
    double C;
    double D;
};

struct Cube
{
    vector<Mass> Masses;
    vector<Spring> Springs;
};


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));
}


class Individual
{
private:
    vector<GENE> gene;
public:
    double initialLocation[3] = { 0,0,0 };
    vector<Mass> Masses;
    vector<Spring> Springs;

    void generate_Masses(double X, double Y, double Z)
    {
        Masses.push_back({ mass,{X + 1.5 * length,Y + 1.5 * length,Z},{0,0,0},{0,0,0} });
        Masses.push_back({ mass,{X + 1.5 * length,Y + 1.5 * length,Z +
length},{0,0,0},{0,0,0} });
        Masses.push_back({ mass,{X - 1.5 * length,Y + 1.5 * length,Z},{0,0,0},{0,0,0} });
        Masses.push_back({ mass,{X - 1.5 * length,Y + 1.5 * length,Z +
length},{0,0,0},{0,0,0} });
        Masses.push_back({ mass,{X - 1.5 * length,Y - 1.5 * length,Z},{0,0,0},{0,0,0} });
        Masses.push_back({ mass,{X - 1.5 * length,Y - 1.5 * length,Z +
length},{0,0,0},{0,0,0} });
        Masses.push_back({ mass,{X + 1.5 * length,Y - 1.5 * length,Z},{0,0,0},{0,0,0} });
        Masses.push_back({ mass,{X + 1.5 * length,Y - 1.5 * length,Z +
length},{0,0,0},{0,0,0} });

        Masses.push_back({ mass,{X + 1 * length,Y + 1 * length,Z +
length},{0,0,0},{0,0,0} });
        Masses.push_back({ mass,{X + 1 * length,Y + 1 * length,Z + 2.0 *
length},{0,0,0},{0,0,0} });
        Masses.push_back({ mass,{X - 1 * length,Y + 1 * length,Z +
length},{0,0,0},{0,0,0} });
        Masses.push_back({ mass,{X - 1 * length,Y + 1 * length,Z + 2.0 *
length},{0,0,0},{0,0,0} });
        Masses.push_back({ mass,{X - 1 * length,Y - 1 * length,Z +
length},{0,0,0},{0,0,0} });
        Masses.push_back({ mass,{X - 1 * length,Y - 1 * length,Z + 2.0 *
length},{0,0,0},{0,0,0} });
        Masses.push_back({ mass,{X + 1 * length,Y - 1 * length,Z +
length},{0,0,0},{0,0,0} });
        Masses.push_back({ mass,{X + 1 * length,Y - 1 * length,Z + 2.0 *
length},{0,0,0},{0,0,0} });
    }

    void generate_Springs()
    {
        int count = 0;
        Springs.push_back({ 2000,distance(Masses[1],Masses[9]), 1,9 });
        Springs.push_back({ 2000,distance(Masses[1],Masses[10]), 1,10 });
        Springs.push_back({ 2000,distance(Masses[1],Masses[14]), 1,14 });
        Springs.push_back({ 2000,distance(Masses[3],Masses[8]), 3,8 });
        Springs.push_back({ 2000,distance(Masses[3],Masses[11]), 3,11 });
        Springs.push_back({ 2000,distance(Masses[3],Masses[12]), 3,12 });
        Springs.push_back({ 2000,distance(Masses[5],Masses[10]), 5,10 });
        Springs.push_back({ 2000,distance(Masses[5],Masses[13]), 5,13 });
        Springs.push_back({ 2000,distance(Masses[5],Masses[14]), 5,14 });
        Springs.push_back({ 2000,distance(Masses[7],Masses[8]), 7,8 });
        Springs.push_back({ 2000,distance(Masses[7],Masses[12]), 7,12 });
        Springs.push_back({ 2000,distance(Masses[7],Masses[15]), 7,15 });

        Springs.push_back({ k,distance(Masses[0],Masses[8]), 0,8 });
        Springs.push_back({ k,distance(Masses[2],Masses[10]), 2,10 });
        Springs.push_back({ k,distance(Masses[4],Masses[12]), 4,12 });
        Springs.push_back({ k,distance(Masses[6],Masses[14]), 6,14 });

        Springs.push_back({ k,distance(Masses[0],Masses[9]), 0,9 });
        Springs.push_back({ k,distance(Masses[2],Masses[11]), 2,11 });
        Springs.push_back({ k,distance(Masses[4],Masses[13]), 4,13 });
        Springs.push_back({ k,distance(Masses[6],Masses[15]), 6,15 });

        Springs.push_back({ k,distance(Masses[0],Masses[1]), 0,1 });
        Springs.push_back({ k,distance(Masses[2],Masses[3]), 2,3 });
        Springs.push_back({ k,distance(Masses[4],Masses[5]), 4,5 });
        Springs.push_back({ k,distance(Masses[6],Masses[7]), 6,7 });
        Springs.push_back({ k,distance(Masses[1],Masses[8]), 1,8 });
        Springs.push_back({ k,distance(Masses[3],Masses[10]), 3,10 });
        Springs.push_back({ k,distance(Masses[5],Masses[12]), 5,12 });
        Springs.push_back({ k,distance(Masses[7],Masses[14]), 7,14 });

        for (int i = 8; i < Masses.size() - 1; i++)
        {
            for (int j = i + 1; j < Masses.size(); j++)
            {
                Springs.push_back({ k,distance(Masses[i],Masses[j]), i,j });
            }
        }
        for (int i = 0; i < 12; i++)
        {
            OriLength[i] = Springs[i].l0;
        }
    }

    Individual(double X, double Y, double Z, vector<GENE> gene)
    {
        // default constructor
        initialLocation[0] = X; initialLocation[1] = Y; initialLocation[2] = Z;
        gene = gene;
        generate_Masses(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();
        }



        glPushMatrix();
        glMultMatrixf(world_rotation);
        glBegin(GL_QUADS);
        glColor3f(0.2, 0.1, 0.3);
        glVertex3f(GLfloat(Masses[8].p[0]), GLfloat(Masses[8].p[1]),
GLfloat(Masses[8].p[2]));
        glVertex3f(GLfloat(Masses[9].p[0]), GLfloat(Masses[9].p[1]),
GLfloat(Masses[9].p[2]));
        glVertex3f(GLfloat(Masses[10].p[0]), GLfloat(Masses[10].p[1]),
GLfloat(Masses[10].p[2]));
        glVertex3f(GLfloat(Masses[11].p[0]), GLfloat(Masses[11].p[1]),
GLfloat(Masses[11].p[2]));
        glEnd();
        glBegin(GL_QUADS);
        glColor3f(0.2, 0.1, 0.3);
        glVertex3f(GLfloat(Masses[10].p[0]), GLfloat(Masses[10].p[1]),
GLfloat(Masses[10].p[2]));
        glVertex3f(GLfloat(Masses[11].p[0]), GLfloat(Masses[11].p[1]),
GLfloat(Masses[11].p[2]));
        glVertex3f(GLfloat(Masses[12].p[0]), GLfloat(Masses[12].p[1]),
GLfloat(Masses[12].p[2]));
        glVertex3f(GLfloat(Masses[13].p[0]), GLfloat(Masses[13].p[1]),
GLfloat(Masses[13].p[2]));
        glEnd();
        glBegin(GL_QUADS);
        glColor3f(0.2, 0.1, 0.3);
        glVertex3f(GLfloat(Masses[12].p[0]), GLfloat(Masses[12].p[1]),
GLfloat(Masses[12].p[2]));
         glVertex3f(GLfloat(Masses[13].p[0]), GLfloat(Masses[13].p[1]),
GLfloat(Masses[13].p[2]));
         glVertex3f(GLfloat(Masses[14].p[0]), GLfloat(Masses[14].p[1]),
GLfloat(Masses[14].p[2]));
         glVertex3f(GLfloat(Masses[15].p[0]), GLfloat(Masses[15].p[1]),
GLfloat(Masses[15].p[2]));
         glEnd();
         glBegin(GL_QUADS);
         glColor3f(0.2, 0.1, 0.3);
         glVertex3f(GLfloat(Masses[8].p[0]), GLfloat(Masses[8].p[1]),
GLfloat(Masses[8].p[2]));
         glVertex3f(GLfloat(Masses[10].p[0]), GLfloat(Masses[10].p[1]),
GLfloat(Masses[10].p[2]));
         glVertex3f(GLfloat(Masses[12].p[0]), GLfloat(Masses[12].p[1]),
GLfloat(Masses[12].p[2]));
         glVertex3f(GLfloat(Masses[14].p[0]), GLfloat(Masses[14].p[1]),
GLfloat(Masses[14].p[2]));
         glEnd();
         glBegin(GL_QUADS);
         glColor3f(0.2, 0.1, 0.3);
         glVertex3f(GLfloat(Masses[9].p[0]), GLfloat(Masses[9].p[1]),
GLfloat(Masses[9].p[2]));
         glVertex3f(GLfloat(Masses[11].p[0]), GLfloat(Masses[11].p[1]),
GLfloat(Masses[11].p[2]));
         glVertex3f(GLfloat(Masses[13].p[0]), GLfloat(Masses[13].p[1]),
GLfloat(Masses[13].p[2]));
         glVertex3f(GLfloat(Masses[15].p[0]), GLfloat(Masses[15].p[1]),
GLfloat(Masses[15].p[2]));
         glEnd();
         glPopMatrix();



         double x = 0; double y = 0; double z = 0;
         for (int j = 0; j < mass_number; j++) {
             x = x + Masses[j].p[0];
             y = y + Masses[j].p[1];
             z = z + Masses[j].p[2];
         }
         x = x / mass_number;
         y = y / mass_number;
         z = z / mass_number;
         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));
    }

    void robot_update()
    {
        bool move = true;

        vector<vector<double>> forces(mass_number, vector<double>(3));



        for (int i = 0; i < 12; i++)
        {
            Springs[i].l0 = OriLength[i] + gene[i].A * OriLength[i] * sin(W * T + gene[i].B)
+
                gene[i].C * OriLength[i] * sin(2 * W * T + gene[i].D);
        }

        for (int i = 0; i < spring_number; 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 l0 = sqrt(pow(position_distance[0], 2) + pow(position_distance[1], 2)
+ pow(position_distance[2], 2));
            double force = Springs[i].k * fabs(Springs[i].l0 - l0);
            int flag = 1;

            if (l0 > Springs[i].l0) {
                flag = -1;
            }
            forces[Springs[i].m1][0] = forces[Springs[i].m1][0] - flag * force *
position_distance[0] / l0;
            forces[Springs[i].m1][1] = forces[Springs[i].m1][1] - flag * force *
position_distance[1] / l0;
            forces[Springs[i].m1][2] = forces[Springs[i].m1][2] - flag * force *
position_distance[2] / l0;
```

```
            forces[Springs[i].m2][0] = forces[Springs[i].m2][0] + flag * force *
position_distance[0] / l0;
            forces[Springs[i].m2][1] = forces[Springs[i].m2][1] + flag * force *
position_distance[1] / l0;
            forces[Springs[i].m2][2] = forces[Springs[i].m2][2] + flag * force *
position_distance[2] / l0;
        }
        for (int i = 0; i < mass_number; 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;
            }

        };
    }
};

class Evolution {
private:
    int population;
    vector<double> distances;
    vector<vector<GENE>> gene_all;
    vector<vector<GENE>> new_gene;
    vector<vector<GENE>> maxgene;
    vector<Individual> robots;
public:
    double average;
    double longest;

    Evolution(int populationnum)
    {
        population = populationnum;
        InitialGene(population);
        Initialize();
    }

    void simulate(double time)
    {
        double time = omp_get_wtime() - start_time;
        fitness();
        selection();
        crossOver();
        mutation();
        gene_all.clear();
        gene_all = new_gene;
        generationNumber++;
        robots.clear();
        Initialize();
        T = 0;
        start_time = omp_get_wtime();

    }
    void InitialGene(int number)
    {
        srand(time(0));
        for (int i = 0; i < number; i++) {
            vector<GENE> gene_tank;
            for (int j = 0; j < evospring; j++) {
                double A = 0.5 * (double)rand() / ((double)RAND_MAX);;
                double B = -2 * M_PI + static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (4 * M_PI)));
                double C = 0.5 * (double)rand() / ((double)RAND_MAX);
                double D = -2 * M_PI + static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (4 * M_PI)));
                GENE tank;
                tank.A = A;
                tank.B = B;
                tank.C = C;
                tank.D = D;
                gene_tank.push_back(tank);
            }
            gene_all.push_back(gene_tank);
        }
    }
    void Initialize()
    {
        for (int i = 0; i < population; i++) {
            double X = 2.0 * (double)rand() / (double)RAND_MAX;
            double Y = 2.0 * (double)rand() / (double)RAND_MAX;
            robots.push_back(Individual(X, Y, 0.0, gene_all[i]));
        }
    }
    void fitness()
    {
        distances.clear();
        for (int i = 0; i < population; i++) {
            double x = 0; double y = 0;
            for (int j = 0; j < mass_number; j++) {
                x = x + robots[i].Masses[j].p[0];
                y = y + robots[i].Masses[j].p[1];
            }
            x = x / mass_number;
            y = y / mass_number;
            double distance = fabs(x - robots[i].initialLocation[0]);
            distances.push_back(distance);
        }
        average = 0;
        longest = 0;
        int m = 0;
        for (int i = 0; i < population; i++)
        {
            average = average + distances[i];
            if (longest < distances[i])
            {
                longest = distances[i];
            }
        }
        average = average / population;
    }
    void selection() {
        sort(distances.begin(), distances.end());
        new_gene.clear();
        new_gene.shrink_to_fit();
        if (distances[0] > maxpath)
        {
            maxpath = distances[0];
            maxgene.clear();
            maxgene.push_back(gene_all[0]);
        }
        new_gene.push_back(maxgene[0]);

        for (int i = 0; i < population / 2 - 1; i++) {
            new_gene.push_back(gene_all[i]);
        }
    }
    void crossOver() {
        for (int n = 0; n < gene_all.size() / 4; n++)
        {
            int mom = rand() % (gene_all.size() / 2);
            int dad = rand() % (gene_all.size() / 2);

            new_gene.push_back(new_gene[mom]);
            new_gene.push_back(new_gene[dad]);
            double cross_Probability = (double)rand() / (double)RAND_MAX;
            if (cross_Probability < cross_rate) {
                int point1, point2, point;
                while (true)
                {
                    point1 = rand() % evospring;
                    point2 = rand() % evospring;
                    if (point1 < point2)
                    {
                        break;
                    }
                }
                int l = point2 - point1;
                while (true)
                {
                    point = rand() % evospring;
                    if (point + l < evospring)
                    {
                        break;
                    }
                }
                GENE tank;
                for (int i = point1; i <= point2; i++)
                {
                    tank = new_gene[mom][i];
                    new_gene[mom][i] = new_gene[dad][point];
                    new_gene[dad][point] = tank;
                    point += 1;
                }
            }
        }
    }
    void mutation()
    {
        for (int i = 0; i < new_gene.size(); i++)
        {
            for (int j = 0; j < evospring; j++)
            {
                double p = (double)rand() / (double)RAND_MAX;
                if (p < mutation_rate)
                {
                    new_gene[i][j].A = new_gene[i][j].A * (1.0 + (0.1 * (double)rand()
/ ((double)RAND_MAX) - 0.05));
                    if (abs(new_gene[i][j].A) > 0.5)
                        new_gene[i][j].A = 0.25;

                    new_gene[i][j].B = new_gene[i][j].B * (1.0 + (0.1 * (double)rand()
/ ((double)RAND_MAX) - 0.05));
```

```
                    new_gene[i][j].C = new_gene[i][j].C * (1.0 + (0.1 * (double)rand()
/ ((double)RAND_MAX) - 0.05));
                    if (abs(new_gene[i][j].C) > 0.5)
                        new_gene[i][j].C = 0.25;

                    new_gene[i][j].D = new_gene[i][j].D * (1.0 + (0.1 * (double)rand()
/ ((double)RAND_MAX) - 0.05));
                }
            }
        }

    }
    void update()
    {
        for (int i = 0; i < population; i++) {
            robots[i].robot_update();
        }
    }

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


};

Evolution first(number);


#if 1

void drawfloor() {
    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);

    drawfloor();
    first.simulate(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
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[])
{
#ifdef Plot
    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 Plot
    while (1) {
        double start_time = omp_get_wtime();
        first.startSim(simulationTime);
        T = T + timestep;
        double time = omp_get_wtime() - start_time;
    }
#endif
};
```