COLUMBIA UNIVERSITY

MECE 4510 EVOLUTIONARY COMPUTATION AND DESIGN AUTOMATION

Evolved robot

Yi Jiang & Chiqu Li

UNI:cl3895

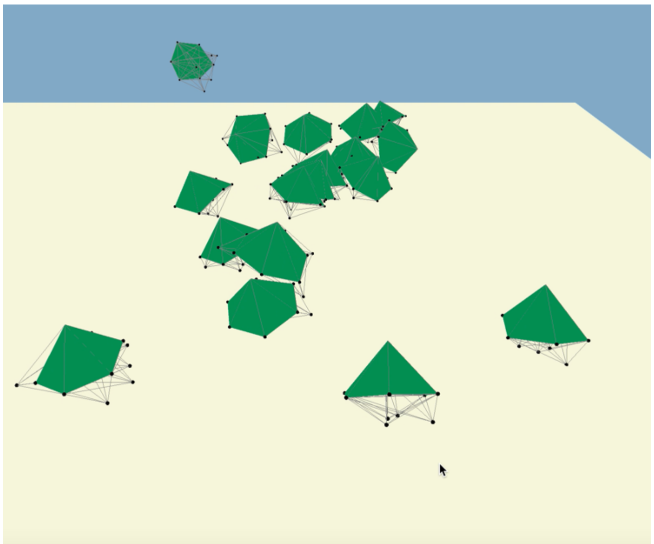
Instructor: Dr. Hod Lipson

Grace Hours Used: 20

Grace Hours Remaining: 126h

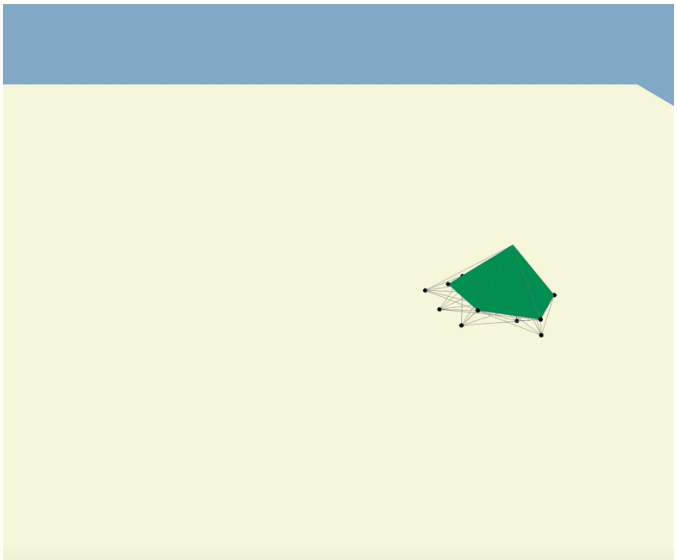
Dec 8, 2019

Result

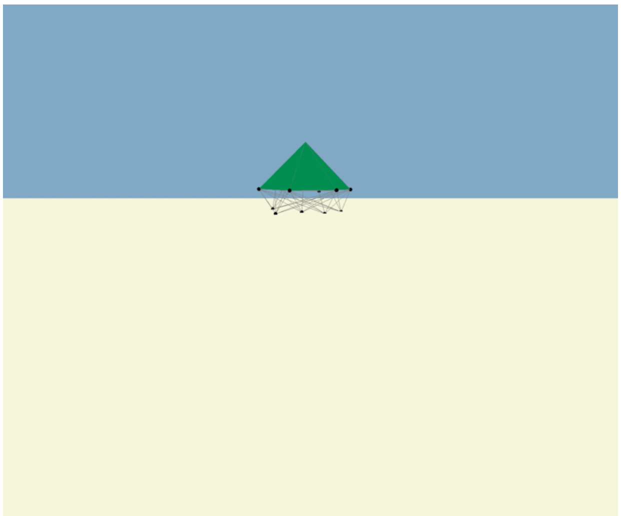


https://youtu.be/hAcLQ7_2fcwhttps://youtu.be/ppW31Lb8cPg

the fast robot



https://youtu.be/rpAglpyxgjo



https://www.youtube.com/watch?v=oryK-FSM-dM

Method

Description

In this phase, we should evolve a robot with a variable morphology. Unlike last phase of assignment 3, we change the number of feet of the robot. This base robot contains 13 masses and 72 springs, the picture as followed, just similar to last phase. We choose the all springs connected to each foot to evolve. And then after 20 second, we delete the worst robot and randomly create a new robot with different number of foot and different gene, and evolve them.

Evolution

Like last phase of assignment 3, the population size is 20, the definition of fitness is the distance away from the initial position at x axis. Selection is keeping top 50% population by fitness to the next generation. Unlike last phase, crossover is choosing two parents from top 50% population and exchange the part of them even though their length is different. And then deleting the worst gene and robot, and adding a new gene and a new robot based on this new gene. The robot has randomly number of feet and randomly position of the feet. At last we mutate the whole generation with a small probability. Each generation we let the robot move for 20 second and then evolve them.

Parameter

1. simulation parameters

timeStep = 0.002;
Nground = 50000;
dampening = 1;
frictionCoefficient = 0.5;
generationT = 6s; // the time in one generation

2. robot parameters

k = 9000; // spring constant to other spring that are not evolved
```
mass = 2; // mass of one joint of robot
length = 0.1;
gravity = 9.81;
```

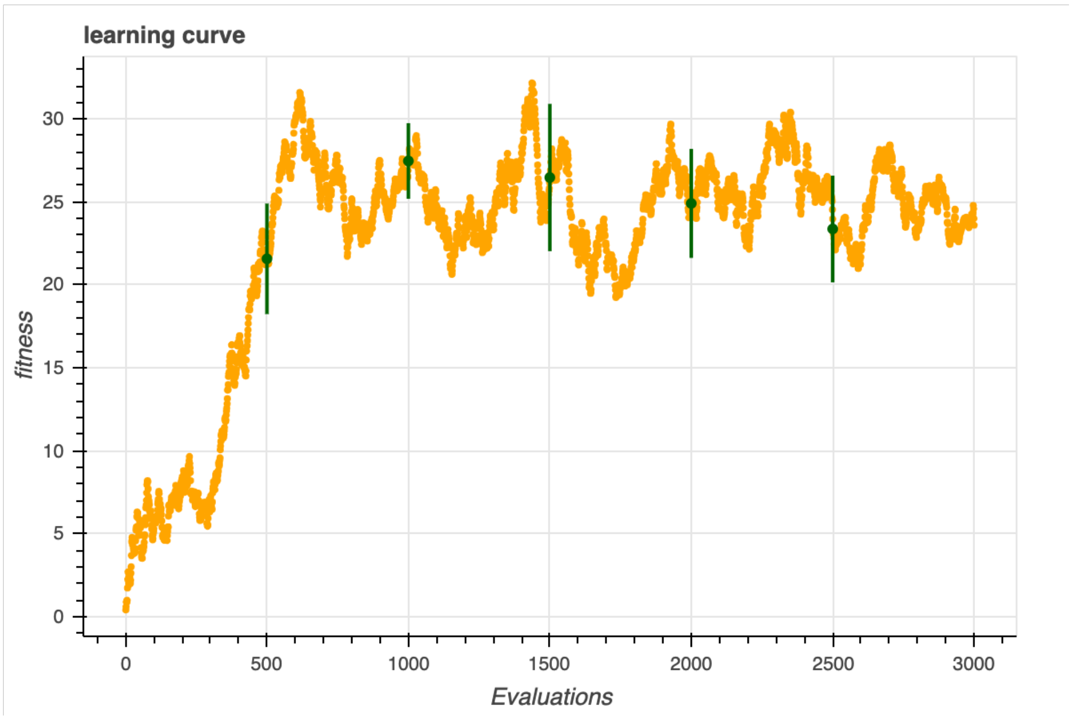
3. evolutionary parameters

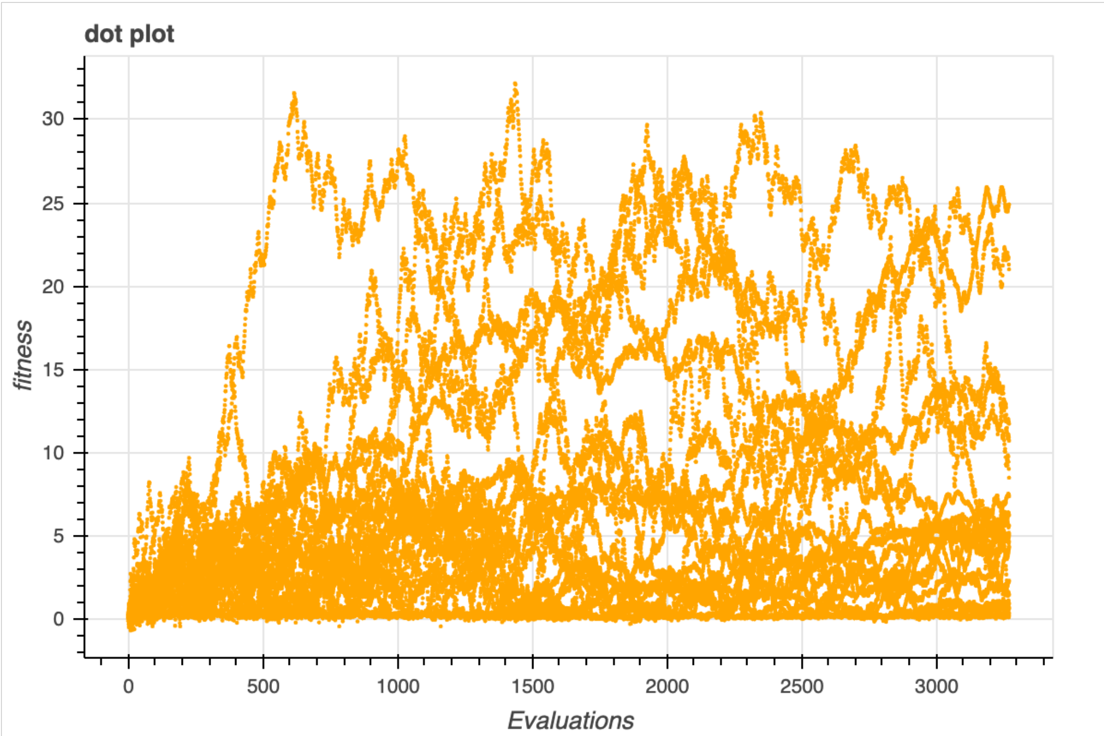
k : 600-5000; the range of evolved spring
b: -1——1; the range of b
c: -2PI——2PI;the range of c
size = 20; //population size
generation = 3000; // evaluation
mutation probability = 0.5

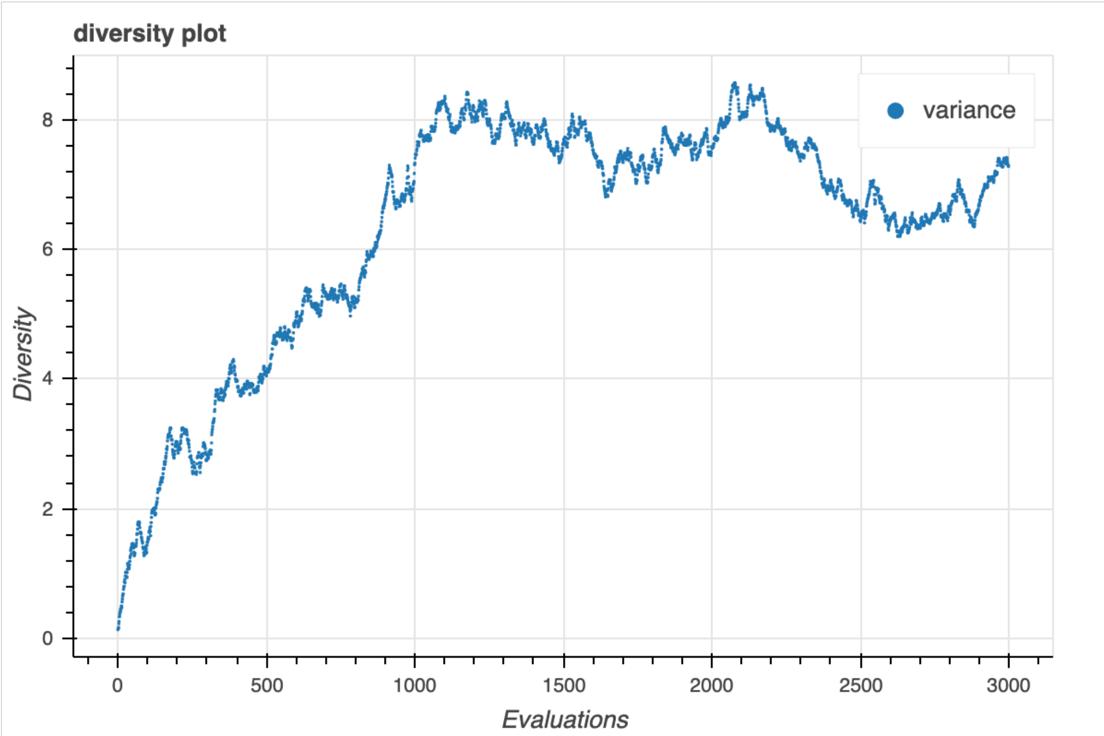
Analysis

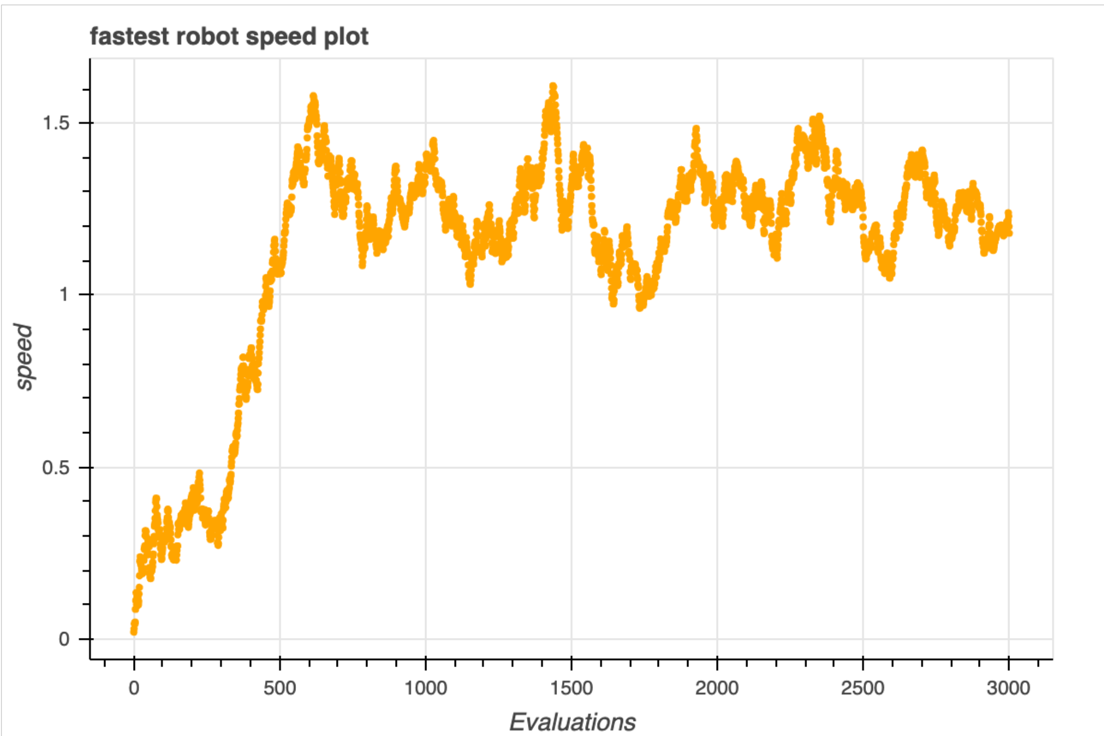
Compared to last phase of assignment, after lots of tempts to get a reasonable range of parameters, we concluded that the shape of robot is important and all of the parameters play a significant role in the robot performance. Among three parameters (k, b, c), it turns out that the k is the key of the robot. If the k is too large or too small, robots would be very unstable. So we choose a range of 600-5000.

Performance






diversity plot
variance
Diversity
Evaluations
0
2
4
6
8
0
500
1000
1500
2000
2500
3000


fastest robot speed plot
speed
Evaluations
0
0.5
1
1.5
0
500
1000
1500
2000
2500
3000

Appendix

```
#include "HW3.h"
GLfloat worldRotation[16] = { 1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1 };

struct MASS
{
    double m;       // mass
    double p[3];    // 3D position
    double v[3];    // 3D velocity
    double a[3];    // 3D acceleration
};

struct SPRING
{
    double k;       // spring constant
    double L_0;     // rest length
    int m1;         // first mass connected
    int m2;         // second mass connected
};

struct GENE {
    double k;
    double b;
    double c;
};
//int num_foot = 2 + rand() % 6;

//ofstream outFile1("distance3.txt");
//ofstream outFile2("bestgene3.txt");
//ofstream outFile3("dot3.txt");

//struct Cube {
//    struct MASS cubemass[8];
//    struct SPRING cubespring[28];
//};

float L(MASS mass1, MASS mass2) {
    double length = sqrt(pow((mass1.p[0] - mass2.p[0]), 2) + pow((mass1.p[1] - 
mass2.p[1]), 2) + pow((mass1.p[2] - mass2.p[2]), 2));

    return length;
}
```

```
//vector<MASS> joint = jointmass(mass, length, 0, 0, 0.01);
//
//
//vector<SPRING> spring = cubespring(length, k);


GLuint tex;
GLUquadric* sphere;
void make_tex(void)
{
    unsigned char data[256][256][3];
    for (int y = 0; y < 255; y++) {
        for (int x = 0; x < 255; x++) {
            unsigned char* p = data[y][x];
            p[0] = p[1] = p[2] = (x ^ y) & 8 ? 255 : 0;
        }
    }
    glGenTextures(1, &tex);
    glBindTexture(GL_TEXTURE_2D, tex);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 256, 256, 0, GL_RGB, GL_UNSIGNED_BYTE, (const
GLvoid*)data);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
}

void init(void)
{
    glEnable(GL_DEPTH_TEST);
    make_tex();
    sphere = gluNewQuadric();
    glEnable(GL_TEXTURE_2D);
}



class Robot {
private:
    vector<GENE> walkgene;

public:
    vector<MASS> joint;
    vector<SPRING> spring;
    int mass_num;
    int num_sp;
```

```
    int num_foot;
    vector<double> original;
    double initl[3] = { 0,0,0 };
    Robot(double x, double y, double z, int num, vector<GENE> newGene)
    {
        mass_num = 7 + num;
        num_foot = num;
        num_sp = 21+ 6 * num_foot;
        initl[0] = x;
        initl[1] = y;
        initl[2] = z;
        walkgene = newGene;
        joint = jointmass(x, y, z, num);
        cubespring(walkgene);

    }

    vector<MASS> jointmass(double x, double y, double z, int num)
    {
        vector<MASS> joint(7 + num);
        double subtle = length / 2;
        joint[0] = { mass, {x + 1.5 * length,y + length * sqrt(3) / 2, z + 1.5 * length},
{0,0,0}, {0,0,0} };
        joint[1] = { mass, {x + length   ,y + sqrt(3) * length,z + 0.5 * length}, {0,0,0},
{0,0,0} };
        joint[2] = { mass, {x + 2 * length  ,y + sqrt(3) * length ,z + 0.5 * length},
{0,0,0}, {0,0,0} };
        joint[3] = { mass, {x + length / 2,y + length / 2 * sqrt(3),z + 0.5 * length},
{0,0,0}, {0,0,0} };
        joint[4] = { mass, {x + length / 2 + 2 * length ,y + length / 2 * sqrt(3),z + 0.5
* length}, {0,0,0}, {0,0,0} };
        joint[5] = { mass, {x + length,y,z + 0.5 * length}, {0,0,0}, {0,0,0} };
        joint[6] = { mass, {x + 2 * length ,y ,z + 0.5 * length}, {0,0,0}, {0,0,0} };
        for (int i = 7; i < 7 + num; i++) {
            double a = -length + static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (2 * length)));
            double b = -length + static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (2 * length)));
            joint[i] = { mass, {joint[0].p[0] + a ,joint[0].p[1] + b, z}, {0,0,0},
{0,0,0} };
        }
        //joint[7] = { mass, {x + 1.5 * length ,y + length * 3 * sqrt(3) / 2 - subtle,z},
{0,0,0}, {0,0,0} };
        //joint[8] = { mass, {x + subtle ,y + sqrt(3) * length - subtle ,z}, {0,0,0},
```

```
{0,0,0} };
        //joint[9] = { mass, {x + 3 * length - subtle,y + sqrt(3) * length - subtle,z },
{0,0,0}, {0,0,0} };
        //joint[10] = { mass, {x + subtle ,y + subtle ,z }, {0,0,0}, {0,0,0} };
        //joint[11] = { mass, {x + 3 * length - subtle, y + subtle ,z}, {0,0,0}, {0,0,0} };
        //joint[12] = { mass, {x + 1.5 * length,y - length * sqrt(3) / 2 + subtle, z},
{0,0,0}, {0,0,0} };
        return joint;
    }
    void cubespring(vector<GENE> walkgene)
    {

        int pointer = 0;
        for (int i = 0; i < 6; i++) {
            for (int j = i + 1; j < 7; j++) {
                original.push_back(L(joint[i], joint[j]));
                spring.push_back({ walkgene[pointer].k,L(joint[i],joint[j]),i,j });
//                cout << spring[i * j].k << endl;
                pointer++;



            }
        }
        for (int i = 7; i < 7 + num_foot; i++) {
            for (int j = 0; j < 6; j++) {
                original.push_back(L(joint[i], joint[j]));
                spring.push_back({ walkgene[pointer].k,L(joint[i],joint[j]),i,j });
                pointer++;
            }
        }



    }

    void drawcube()
    {

        glPushMatrix();
        glMultMatrixf(worldRotation);

        glBegin(GL_TRIANGLES);
        glColor3f(R,G,B);
        glVertex3f(GLfloat(joint[0].p[0]),                      GLfloat(joint[0].p[1]),
GLfloat(joint[0].p[2]));
```

```
	glVertex3f(GLfloat(joint[1].p[0]), 	GLfloat(joint[1].p[1]), 
GLfloat(joint[1].p[2]));
	glVertex3f(GLfloat(joint[2].p[0]), 	GLfloat(joint[2].p[1]), 
GLfloat(joint[2].p[2]));
	glEnd();

	glBegin(GL_TRIANGLES);
	glColor3f(R,G,B);
	glVertex3f(GLfloat(joint[0].p[0]), 	GLfloat(joint[0].p[1]), 
GLfloat(joint[0].p[2]));
	glVertex3f(GLfloat(joint[1].p[0]), 	GLfloat(joint[1].p[1]), 
GLfloat(joint[1].p[2]));
	glVertex3f(GLfloat(joint[3].p[0]), 	GLfloat(joint[3].p[1]), 
GLfloat(joint[3].p[2]));
	glEnd();

	glBegin(GL_TRIANGLES);
	glColor3f(R,G,B);
	glVertex3f(GLfloat(joint[0].p[0]), 	GLfloat(joint[0].p[1]), 
GLfloat(joint[0].p[2]));
	glVertex3f(GLfloat(joint[5].p[0]), 	GLfloat(joint[5].p[1]), 
GLfloat(joint[5].p[2]));
	glVertex3f(GLfloat(joint[3].p[0]), 	GLfloat(joint[3].p[1]), 
GLfloat(joint[3].p[2]));
	glEnd();

	glBegin(GL_TRIANGLES);
	glColor3f(R,G,B);
	glVertex3f(GLfloat(joint[0].p[0]), 	GLfloat(joint[0].p[1]), 
GLfloat(joint[0].p[2]));
	glVertex3f(GLfloat(joint[5].p[0]), 	GLfloat(joint[5].p[1]), 
GLfloat(joint[5].p[2]));
	glVertex3f(GLfloat(joint[6].p[0]), 	GLfloat(joint[6].p[1]), 
GLfloat(joint[6].p[2]));
	glEnd();

	glBegin(GL_TRIANGLES);
	glColor3f(R,G,B);
	glVertex3f(GLfloat(joint[0].p[0]), 	GLfloat(joint[0].p[1]), 
GLfloat(joint[0].p[2]));
	glVertex3f(GLfloat(joint[4].p[0]), 	GLfloat(joint[4].p[1]), 
GLfloat(joint[4].p[2]));
	glVertex3f(GLfloat(joint[6].p[0]), 	GLfloat(joint[6].p[1]), 
GLfloat(joint[6].p[2]));
```

```
        glEnd();

        glBegin(GL_TRIANGLES);
        glColor3f(R,G,B);
        glVertex3f(GLfloat(joint[0].p[0]),                              GLfloat(joint[0].p[1]),
GLfloat(joint[0].p[2]));
        glVertex3f(GLfloat(joint[4].p[0]),                              GLfloat(joint[4].p[1]),
GLfloat(joint[4].p[2]));
        glVertex3f(GLfloat(joint[2].p[0]),                              GLfloat(joint[2].p[1]),
GLfloat(joint[2].p[2]));
        glEnd();
        glPopMatrix();



        //
        GLUquadric* quad[mass_num];
        for (int i = 1; i < mass_num; i++) {

            glColor3f(R1,G1, B1);
            quad[i] = gluNewQuadric();
            glPushMatrix();
            glMultMatrixf(worldRotation);
            glTranslated(joint[i].p[0], joint[i].p[1], joint[i].p[2]);

            gluSphere(quad[i], 1.0 / 250, 10, 10);
            //          if (i == 6) { gluSphere(quad[i], 1.0 / 100, 10, 10); }
            glPopMatrix();

        }

        for (int i = 0; i < num_sp; i++) {
            int a = spring[i].m1;
            int b = spring[i].m2;

            glColor3f(0.5, 0.5, 0.5);
            glPushMatrix();
            glMultMatrixf(worldRotation);
            glBegin(GL_LINES);
            glLineWidth(20);
            glVertex3f(joint[a].p[0], joint[a].p[1], joint[a].p[2]);
            glVertex3f(joint[b].p[0], joint[b].p[1], joint[b].p[2]);
            glEnd();
            glPopMatrix();
```

```
        }
    }

    void drawground()
    {
        glPushMatrix();
        glColor3f(0.96078, 0.96078, 0.86274);
        glBegin(GL_QUADS);
        glNormal3f(0, 1, 0);
        glTexCoord2f(0.0, 0.0);  glVertex3f(-1.5, +0.0, -1.5);
        glTexCoord2f(0.0, 1);  glVertex3f(+1.5, +0.0, -1.5);
        glTexCoord2f(1, 1);  glVertex3f(+1.5, +0.0, +1.5);
        glTexCoord2f(1, 0.0);  glVertex3f(-1.5, +0.0, +1.5);
        glEnd();
        glPopMatrix();
        glDisable(GL_TEXTURE_2D);
        for (int i = 0; i < 10; i++) {
            for (int j = -9; j < 10; j++) {
                glColor3f(0, 1, 0);
                glPushMatrix();
                glMultMatrixf(worldRotation);
                glBegin(GL_LINES);
                glLineWidth(10);
                glVertex3f(-0.5 * i / 10, 0.02, 0.5 * j / 10);
                //
glV/Users/chiqu/class2019fall/EA/assignment3/HW3/HW3.cppertex3f(0.5*i/10,    0.5*j/10,
0.01);
                glEnd();
                glPopMatrix();
            }
        }
    }

    void simulate() {
        vector<vector<double>> cForces(num_foot + 7, vector<double>(3));
        for (int i = 0; i < mass_num; i++) {
            cForces[i][0] = 0.0;
            cForces[i][1] = 0.0;
            cForces[i][2] = -joint[i].m * gravity;
        }

        for (int i = 0; i < num_sp; i++) {
            MASS mass1 = joint[spring[i].m1];
            MASS mass2 = joint[spring[i].m2];
```

```
            //if (T < 1) {
//            if (i == 18 or i == 37 or i == 17 or i == 27 or i == 29 or i == 39 or i ==
46 or i == 48 or i == 54 or i == 56 or i == 62 or i == 64) {
//
//                  spring[i].L_0 = original[i] * (1 + walkgene[i].b * sin(500 * T +
walkgene[i].c));
//            if (i > num_sp / 2) {
//                  spring[i].L_0 = original[i] * (1 + walkgene[i].b * cos(500 * T +
walkgene[i].c));
//            }}
            if (i > 21 && i < 21+ 6 * num_foot) {
                spring[i].L_0 = original[i] * (1 + walkgene[i].b * sin(500 * T +
walkgene[i].c));
            }

            double pd[3] = { mass2.p[0] - mass1.p[0],mass2.p[1] - mass1.p[1],mass2.p[2] -
mass1.p[2] };
            double new_L = L(mass1, mass2);
            double L_0 = spring[i].L_0;
            double force = spring[i].k * fabs(new_L - L_0);
            //springenergy += k * pow((new_L - L_0), 2) / 2;
            //cout <<i<<"---new_L---" <<new_L << endl;
            double norm_pd[3] = { pd[0] / new_L, pd[1] / new_L, pd[2] / new_L };
            //cout << i << "---force---" << force << endl;
            //compression
            if (new_L < spring[i].L_0) {
                cForces[spring[i].m1][0] -= norm_pd[0] * force;
                cForces[spring[i].m1][1] -= norm_pd[1] * force;
                cForces[spring[i].m1][2] -= norm_pd[2] * force;
                cForces[spring[i].m2][0] += norm_pd[0] * force;
                cForces[spring[i].m2][1] += norm_pd[1] * force;
                cForces[spring[i].m2][2] += norm_pd[2] * force;
            }

            //tension
            else {
                cForces[spring[i].m1][0] += norm_pd[0] * force;
                cForces[spring[i].m1][1] += norm_pd[1] * force;
                cForces[spring[i].m1][2] += norm_pd[2] * force;
                cForces[spring[i].m2][0] -= norm_pd[0] * force;
                cForces[spring[i].m2][1] -= norm_pd[1] * force;
                cForces[spring[i].m2][2] -= norm_pd[2] * force;
            }
        }
```

```
        //cout << T << "tan " << springenergy << endl;
        for (int i = 0; i < mass_num; i++) {
            if (joint[i].p[2] <= 0) {
                cForces[i][2] -= Nground * joint[i].p[2];
                //cForces[i][2] -= joint[i].m*0.981;
                //groundenergy += Nground * pow(joint[i].p[2], 2) / 2;
                double Fh = sqrt(pow(cForces[i][0], 2) + pow(cForces[i][1], 2));
                double Fv = cForces[i][2];
                if (Fh < Fv * frictionCoefficient) {
                    cForces[i][0] = 0;
                    cForces[i][1] = 0;
                    joint[i].v[0] = 0;
                    joint[i].v[1] = 0;
                }
                else {
                    double Fh_new = Fh - Fv * frictionCoefficient;
                    cForces[i][0] = cForces[i][0] * Fh_new / Fh;
                    cForces[i][1] = cForces[i][1] * Fh_new / Fh;
                }
            }
            //cout << T << " " << groundenergy << endl;
            for (int j = 0; j < 3; j++) {
                joint[i].a[j] = cForces[i][j] / joint[i].m;
                joint[i].v[j] += joint[i].a[j] * timeStep;
                joint[i].v[j] *= dampening;
                joint[i].p[j] += joint[i].v[j] * timeStep;
            }
            //cout << cube[i].p[j] << endl;
        //gravityenergy
            //gravityenergy += joint[i].m * gravity * joint[i].p[2];
            //kinetic
            double  norm_v  =  sqrt(pow(joint[i].v[0],  2)  +  pow(joint[i].v[1],  2)  +
pow(joint[i].v[2], 2));
            //kineticenergy += joint[i].m * pow(norm_v, 2) / 2;


            //          cout << i << joint[i].p[2] << endl;
            //          cout << cForces[1][2] << endl;
        }
        //cout << "hhhhh"<< endl;

        //cout << "wtfh" << endl;

        drawground();
```

```
        T = T + timeStep;

    }

};

vector<vector<GENE>> populationGene;
vector<Robot> robot;
vector<double> totald2;
vector<vector<GENE>> newgenepop;
vector<vector<GENE>> nextpop;
vector<GENE> bestGene;
int populationsize = size;



vector<double> bestdistance;

//double maxdistance;
vector<vector<GENE>> bestgene;


vector<GENE> crossover(vector<GENE> p1, vector<GENE> p2) {
    if (p1.size() < p2.size()) {
        int crossposition = 7 + rand() % static_cast<int>(p1.size());

        vector<GENE> offspring = p1;
        //cout << "crossover" << endl;
        for (int i = 0; i < crossposition; i++) {
            offspring[i] = p2[i];
        }

        return offspring;
    }

    else {
        int crossposition = 7 + rand() % static_cast<int>(p2.size());
        vector<GENE> offspring = p2;
        //cout << "crossover" << endl;
        for (int i = 0; i < crossposition; i++) {
            offspring[i] = p1[i];
        }
```

```
        return offspring;
    }
}

vector<vector<GENE>> genereateGene(int num_foot) {
    vector<vector<GENE>> populationGene1;
    for (int i = 0; i < populationsize; i++) {
        vector<GENE> temp;
        GENE temp_2;
//        vector<GENE> temp_4 = {{9000,0,0},{900,0, 0},{9000,0, 0},{900,0, 0},{9000, 0, 
0},{9000, 0, 0},{9000,0, 0},{9000, 0, 0},{9000, 0, 0},{9000, 0, 0},{9000, 0, 0},{9000, 
0, 0},
//        {9000, 0, 0},
//        {9000, 0, 0},
//        {9000, 0, 0},
//        {9000, 0, 0},
//        {9000, 0, 0},
//        {9000, 0, 0},
//        {9000, 0, 0},
//        {9000,0, 0},
//        {9000, 0, 0},
//        {9000, 0, 0},
//        {1149.03, 0.89871, 1.59599},
//        {3387.09, 0.0564407, 1.23423},
//        {2002.62, 0.699834, -4.80743},
//        {2914.94, 0.553911, 1.1057},
//        {817.105, 0.321366, -3.82508},
//        {4385, 0.813266, 0.71422},
//        {1138.03, 0.402719, -0.0315316},
//        {896.933, 0.0647815, 3.55796},
//        {1613.19, 0.143684, 5.02053},
//        {1201.31, 0.177645, 2.19434},
//        {682.43, 0.771142, 1.13884},
//        {1140.98, 0.416475, 2.54433},
//        {1074.05, 0.766175,-5.02824},
//        {2116.14, 0.297064, 3.23736},
//        {2055.34, 0.0556062, 0.919644},
//        {794.846, 0.637075, -2.20964},
//        {2764.1, 0.367395, 3.83164},
//        {1421.09, 0.473704, 0.641933},
//        {2274.68, 0.883705, -0.953366},
//        {1192.37, 0.17468, 4.34046},
//        {1291.49, 0.273336, 5.86253},
//        {1497.38, 0.0790328, -2.46469},
```

```
//       {2897.75, 0.861622, -2.79981},
//       {932.343, 0.309218, -5.87749},
//       {1368.84, 0.606336, 2.27619},
//       {963.681, 0.530644, 0.467611},
//       {870.933, 0.522875, 5.70392},
//       {4467.87, 0.359379, -5.15406},
//           {993.408, 0.668242, -4.40994}};
        double k_1 = 9000;
        double b_1 = 0;
        double c_1 = 0;
        int num_sp = 21 + 6 * num_foot;
        for (int j = 0; j < num_sp; j++) {
//              double k_1 = 100 + static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (1000 - 100)));
//           //              if (j == 18 or j == 37 or j == 17 or j == 27 or j == 29 or
j == 39 or j == 46 or j == 48 or j == 54 or j == 56 or j == 62 or j == 64) {
//           if (j < num_sp && j > 21) {
//                   k_1 = 600 + static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (5000 - 600)));
//                b_1 = static_cast <float> (rand()) / static_cast <float> (RAND_MAX /
1);
//               c_1 = -2 * M_PI + static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (4 * M_PI)));
//           }
//           else {
//               k_1 = 9000;
//               b_1 = 0;
//               c_1 = 0;
            k_1 = 9000;
            b_1 = 0;
            c_1 = 0;


            temp_2 = { k_1, b_1, c_1 };

            temp.push_back(temp_2);}
            //              tempVec.push_back(k1);
            //              tempVec.push_back(b1);
            //              tempVec.push_back(c1);


        populationGene1.push_back(temp);

    }
```

```
    return populationGene1;
}

vector<Robot> sortrobot(vector<Robot> robot, vector<double>totald){

//    vector<int> Index;
//        for (int i = 0; i < populationsize; i++) {
//            cout<<"totald "<<totald[i]<<endl;
//            double temp_min = totald[i];
//            int pointer = i;
//            for (int j = i; j < populationsize; j++) {
//                if (totald[j] > temp_min) {
//                    temp_min = totald[j];
//                    pointer = j;
//
//                }
//
//            }
//            sort(totald.begin(),totald.end());
//            double a = *max_element (totald.begin(), totald.end());
//            k = distance(totald,a);
//            cout<<"pointer"<<pointer<<endl;
//            Index.push_back(pointer);
//        }
        vector<size_t> idx(totald.size());
        iota(idx.begin(), idx.end(), 0);

        // sort indexes based on comparing values in v
        sort(idx.begin(), idx.end(),
             [totald](size_t i1, size_t i2) {return totald[i1] > totald[i2];});

    vector<Robot> tempRobot;
    for(int i = 0; i<populationsize;i++){




        Robot tempp = robot[idx[i]];
        tempRobot.push_back(tempp);
    }

    return tempRobot;

}
```

```
vector<vector<GENE>>  selection(vector<vector<GENE>>  populationGene,  vector<double>
totald) {
    vector<vector<GENE>> newgenepop;
    vector<size_t> idx(totald.size());
    iota(idx.begin(), idx.end(), 0);

    // sort indexes based on comparing values in v
    sort(idx.begin(), idx.end(),
         [totald](size_t i1, size_t i2) {return totald[i1] > totald[i2];});

   vector<size_t> Index = idx;
//
//       for (int i = 0; i < populationsize; i++) {
//           double temp_min = totald[0];
//           int pointer = 0;
//           for (int j = i; j < populationsize; j++) {
//               if (totald[j] > temp_min) {
//                   temp_min = totald[j];
//                   pointer = j;
//               }
//
//           }
//           Index.push_back(pointer);
//       }

    newgenepop.clear();
    newgenepop.shrink_to_fit();
    for (int i = 0; i < Index.size() / 2; i++) {
        newgenepop.push_back(populationGene[Index[i]]);
    }
    bestgene.push_back(populationGene[Index[0]]);
    bestdistance.push_back(totald[Index[0]]);
    cout << "best distance" << totald[Index[0]] << endl;
    evaluation++;
    //
//    outFile1<<evaluation<<" "<< totald[Index[0]]<<endl;

    cout << "evaluation: " << evaluation << endl;
          //cout<<"worst"<<totald[index[-1]]<<endl;
```

```
//    for (int i=0;i<newgenepop[0].size(); i++){
//          outFile2<<i <<" "<< newgenepop[0][i].k << " " <<newgenepop[0][i].b<< " "
<<newgenepop[0][i].c << " ";
//    }
//
    return newgenepop;
}
vector<GENE> genereateNewGene(int num_foot) {

        vector<GENE> temp;
    GENE temp_2;

//       double k_1 = 9000;
//       double b_1 = 0;
//       double c_1 = 0;
//       int num_sp = 21+ 6 * num_foot;
//
//       for (int j = 0; j < num_sp; j++) {
//             double k_1 = 100 + static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (1000 - 100)));
//          //             if (j == 18 or j == 37 or j == 17 or j == 27 or j == 29 or
j == 39 or j == 46 or j == 48 or j == 54 or j == 56 or j == 62 or j == 64) {
//          if (j < num_sp && j > 21) {
//                   k_1 = 600 + static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (5000 - 600)));
//               b_1 = static_cast <float> (rand()) / static_cast <float> (RAND_MAX /
1);
//              c_1 = -2 * M_PI + static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (4 * M_PI)));
//          }
//          else {
//              k_1 = 9000;
//              b_1 = 0;
//              c_1 = 0;
//          }

//          temp_2 = { k_1, b_1, c_1 };
          temp.push_back(temp_2);



    return temp;
}
Robot getnewrobot(vector<GENE> gene, int num_foot){
```

```
//double x = -2 + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX / 4));
double y = -1 + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX / 2));
//   cout<<"getnewgenenenene"<<gene[9].k<<endl;
Robot robot2(0, y, 0.1, num_foot, gene);



return robot2;
}

vector<GENE> mutation(vector<GENE> p1) {
   vector<int> list;
   for (int i = 21; i < p1.size(); i++) {
      list.push_back(i);
   }
   int i = 21 + rand() % static_cast<int>(p1.size() - 21);
   int i1 = list[i];
   vector<GENE> offspring = p1;
   GENE temp;
   temp.k = 600 + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX / (1000
- 100)));
   temp.b = static_cast <float> (rand()) / static_cast <float> (RAND_MAX / 1);
   temp.c = -2 * M_PI + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX
/ (4 * M_PI)));

   offspring[i] = temp;
   list.clear();
   return offspring;
}
vector<vector<GENE>>     nextgeneration(vector<Robot>     robots,vector<vector<GENE>>
newgenepop) {
   vector<vector<GENE>> nextpop = newgenepop;
   vector<GENE> p3;
   for (int a = 0; a<int(newgenepop.size())-1; a++) {
      int i = rand() % static_cast<int>(newgenepop.size());
      int j = rand() % static_cast<int>(newgenepop.size());

      vector<GENE> p1 = nextpop[i];
      vector<GENE> p2 = newgenepop[j];
      if (p1.size() < p2.size()) {
         p1 = crossover(p1, p2);
         nextpop.push_back(p1);
      }
```

```
        else {
            p2 = crossover(p1, p2);
            nextpop.push_back(p2);
        }

    }
    int num = 2 + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX / 10));




    vector<GENE> temp3 = genereateNewGene(num);
    nextpop.push_back(temp3);




    robots.pop_back();
    //nextpop[-1] = temp3;
//    cout <<"nextpop[-1].size(2222)" << nextpop[-1][5].k <<  endl;

    Robot a = getnewrobot(nextpop[9], num);
    robots.push_back(a);

//   robots[-1] = getnewrobot(nextpop[-1], num);
    double r = static_cast <float> (rand()) / (static_cast <float> (RAND_MAX));
    if (r < 0.5) {
        vector<GENE> temp_off;
        int ra_ro = 1 + rand() % static_cast<int>(nextpop.size() - 1);
        temp_off = mutation(nextpop[ra_ro]);
        nextpop[ra_ro] = temp_off;
        cout << "mutationing" << endl;



    }
            for(int i =0; i<nextpop[0].size(); i++){
//                          outFile2<<evaluation<<"  "<<i  <<"  "<<  nextpop[0][i].k<<"
"<<nextpop[0][i].b<<" "<<nextpop[0][i].c<<endl;
            }
    return nextpop;
}

//vector<Robot> getrobots(Robot a){
//   robots.pop_back();
//   robots.push_back(a);
//   return robots;
```

```
//}
//




vector<Robot> getinitalrobot(vector<vector<GENE>> nextpop, int num_foot) {
    vector<Robot> robots;
    for (int i = 0; i < populationsize; i++) {

        //double x = -2 + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX
/ 4));
        double y = -1 + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX /
2));
        //robots.push_back(ROBOT(0.0, 3.0*(i-populationSize/2), 0.0, populationGene[i]));
//          int num = 2 + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX
/ 8));

        Robot robot1(0, y, 0.1, num_foot, nextpop[i]);
        robots.push_back(robot1);

    }
    return robots;
}

    double fittness(Robot r, int num_m) {
       // double fit = 0;
        double startpoint;
        double sumdis = 0;
        for (int i = 0; i < num_m-1; i++) {
//          fit += (r.joint[i].p[0] - r.initl[0]);
            sumdis += (r.joint[i].p[0] - r.initl[0]);

        }
        startpoint = sumdis / num_m;
        r.initl[0] = startpoint;
//       outFile3<< evaluation << " " << sumdis/(num_m-1) << endl;
              //cout<<"startpoint"<<startpoint<<endl;
        return sumdis / (num_m-1);
    }


    vector<double> totaldistance(vector<Robot> robots) {
        vector<double> totald;
```

```
    for (int i = 0; i < populationsize; i++) {
        double f = fittness(robots[i], robots[i].num_foot + 7);

        totald.push_back(f);

    }

    return totald;
}
```

```
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_HELVETICA_18, *ch++);
}

/*
 *  OpenGL (GLUT) calls this routine to display the scene
 */


void display()
{
    //drawground();
    double len = 0.2;  //  Length of axes
    //  Erase the window and the de5th buffer
    glClearColor(0.5372549, 0.6549019, 0.760784, 1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    //  Enable Z-buffering in OpenGL
    glEnable(GL_DEPTH_TEST);
```

```
        //  Undo previous transformations
        glLoadIdentity();
        //  Eye position
        double Ex = -1 * dim * Sin(th) * Cos(ph);
        double Ey = +1 * dim * Sin(ph);
        double Ez = +1 * dim * Cos(th) * Cos(ph);
        gluLookAt(Ex, Ey, Ez, 0, 0, 0, 0, Cos(ph), 0);

        //Simulate();

//        double y = -1 + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX
/ 2));
//
//        Robot robot1(0, y, 0.1, 5, populationGene);
//
//        if (T >0) {

            for (int j = 0; j < populationsize; j++) {
//                cout<<robot[j].num_foot<<endl;
                if(j==0){
                robot[j].simulate();

                robot[j].drawcube();
                }
//            }



        }
//        else {
//            totald2 = totaldistance(robot);
//            newgenepop = selection(populationGene, totald2);
//            nextpop = nextgeneration(robot,newgenepop);
//            robot = sortrobot(robot, totald2);
//            robot.pop_back();
//            robot.push_back(getnewrobot(nextpop[9], int (nextpop[9].size()-21)/6));
//            populationGene = nextpop;
//            //cout<<"in loop evolve"<<endl;
//            T = 0;
//        }

        //Draw axes
        //glColor3f(1, 0, 0);
    //    if (axes)
    //    {
```

```
//      glBegin(GL_LINES);
//      glLineWidth(2);
//      glVertex3d(0.0, 0.0, 0.0);
//      glVertex3d(len, 0.0, 0.0);
//      glVertex3d(0.0, 0.0, 0.0);
//      glVertex3d(0.0, len, 0.0);
//      glVertex3d(0.0, 0.0, 0.0);
//      glVertex3d(0.0, 0.0, len);
//      glEnd();
//      //  Label axes
//      glRasterPos3d(len, 0.0, 0.0);
//      Print("X");
//      glRasterPos3d(0.0, len, 0.0);
//      Print("Y");
//      glRasterPos3d(0.0, 0.0, len);
//      Print("Z");
//   }
   //  Render the scene
   glFlush();
   //  Make the rendered scene visible
   glutSwapBuffers();
}


void special(int key, int x, int y)
{
   //  Right arrow key - increase angle by 5 degrees
   if (key == GLUT_KEY_RIGHT)
      th += 5;
   //  Left arrow key - decrease angle by 5 degrees
   else if (key == GLUT_KEY_LEFT)
      th -= 5;
   //  Up arrow key - increase elevation by 5 degrees
   else if (key == GLUT_KEY_UP)
   {
      if (ph + 5 < 90)
      {
         ph += 5;
      }
   }
   //  Down arrow key - decrease elevation by 5 degrees
   else if (key == GLUT_KEY_DOWN)
   {
      if (ph - 5 > 0)
```

```
      {
         ph -= 5;
      }
   }
   //  Keep angles to +/-360 degrees
   th %= 360;
   ph %= 360;
   //  Tell GLUT it is necessary to redisplay the scene
   glutPostRedisplay();
}


void Project(double fov, double asp, double dim)
{
   //  Tell OpenGL we want to manipulate the projection matrix
   glMatrixMode(GL_PROJECTION);
   //  Undo previous transformations
   glLoadIdentity();
   //  Perspective transformation
   if (fov)
      gluPerspective(fov, asp, dim / 16, 16 * dim);
   //  Orthogonal transformation
   else
      glOrtho(-asp * dim, asp * dim, -dim, +dim, -dim, +dim);
   //  Switch to manipulating the model matrix
   glMatrixMode(GL_MODELVIEW);
   //  Undo previous transformations
   glLoadIdentity();
}


void key(unsigned char ch, int x, int y)
{
   //  Exit on ESC
   if (ch == 27)
      exit(0);
   //  Reset view angle
   else if (ch == '0')
      th = ph = 0;
   //  Toggle axes
   else if (ch == 'a' || ch == 'A')
      axes = 1 - axes;
   //  Change field of view angle
   else if (ch == '-' && ch > 1)
```

```
            fov++;
        else if (ch == '=' && ch < 179)
            fov--;
        //  PageUp key - increase dim
        else if (ch == GLUT_KEY_PAGE_DOWN) {
            dim += 0.1;
        }
        //  PageDown key - decrease dim
        else if (ch == GLUT_KEY_PAGE_UP && dim > 1) {
            dim -= 0.1;
        }
        //  Keep angles to +/-360 degrees
        th %= 360;
        ph %= 360;
        //  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 = (height > 0) ? (double)width / height : 1;
        //  Set the viewport to the entire window
        glViewport(0, 0, width, height);
        //  Set projection
        Project(fov, asp, dim);
    }


    void idle()
    {
        glutPostRedisplay();
    }
int foot = 5;


    int main(int argc, char* argv[])
    {
        populationGene = genereateGene(foot);

        robot = getinitalrobot(populationGene,5);
```

```
//       Robot robot1 = getnewrobot(populationGene, 5);

        // Initialize GLUT and process user parameters
        glutInit(&argc, argv);
        // double buffered, true color 600*600
        glutInitWindowSize(1000, 800);
        glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
        // create the window
        glutCreateWindow("evolve");
        //  Tell GLUT to call "idle" when there is nothing else to do
        glutIdleFunc(idle);
        //  Tell GLUT to call "display" when the scene should be drawn
        glutDisplayFunc(display);
        //  Tell GLUT to call "reshape" when the window is resized
        glutReshapeFunc(reshape);
        //  Tell GLUT to call "special" when an arrow key is pressed
        glutSpecialFunc(special);
        //  Tell GLUT to call "key" when a key is pressed
        glutKeyboardFunc(key);
        init();

        //  Pass control to GLUT so it can interact with the user
        glutMainLoop();

        return 0;



    };
```