

COLUMBIA UNIVERSITY

MECE 4510 EVOLUTIONARY COMPUTATION AND DESIGN AUTOMATION

Assignment 3 - Phase B

Hanwen Zhao

UNI: hz2547

supervised by
Dr. Hod LIPSON

Grace Hours Used: 129
Grace Hours Accumulated: 70
Grace Hours Remaining: 37

Tuesday 4th December, 2018 03:34

1 Result Summary

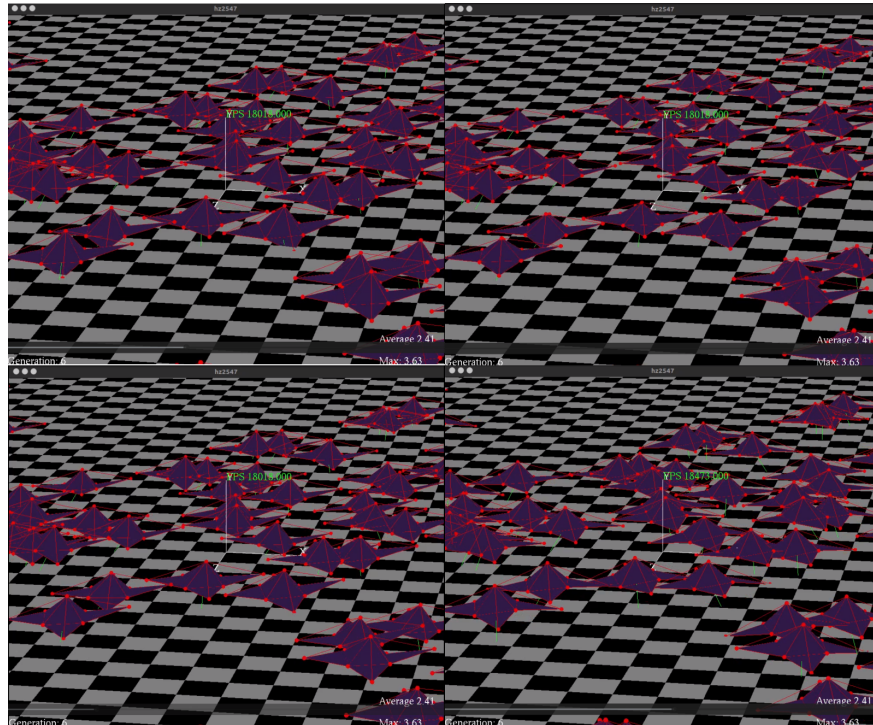


Figure 1: Group of Fastest Robots Moving 0.0163m/s

url: <https://goo.gl/2Gmhao>

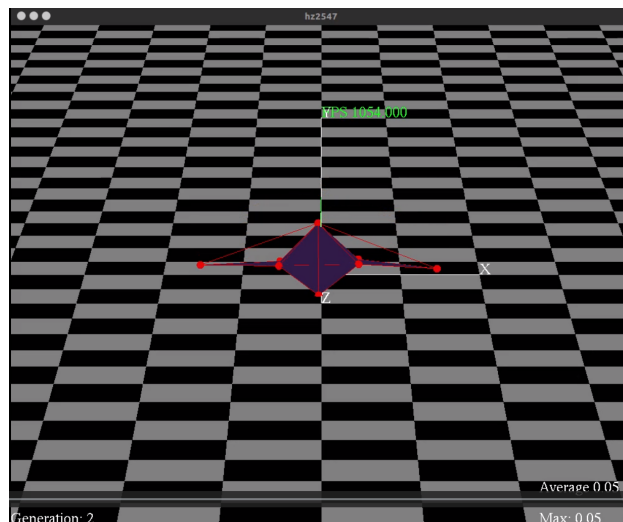


Figure 2: Bouncing Test

url: <https://goo.gl/fsHNQN>

2 Methods

2.1 Description of Design

At the second phase of the project, we are going to evolve a controller for a robot with fixed morphology. First, I created a robot based on a tetrahedron. The robot is shown in the following picture. The tetrahedron base contains 5 masses and 10 springs with high spring constant. Then, I added four legs to the tetrahedron with four softer springs to control them. Hope these four legs could help the robot to move faster. From the previous phase of assignment 3, I built a simulation based on C++ and OpenGL. In this part of the project, the main goal is to code the classes for building the robot and implement evolutionary algorithm.

2.1.1 EA Representation

There are four legs in each robot, and each leg is controlled by one soft spring. The rest length of each soft spring is controlled by the following equation:

$$L = L_0 + A * \sin(B * t + C) \quad (1)$$

Therefore, there are 3 parameters for each leg, 12 parameters for each robot. In this assignment, I am using direct encoding to see how it works, which means each robot is evolved by a 12 numbers gene.

2.1.2 Design Parameters

The following are the parameter I used for this phase of the assignment:
Simulation Parameters:

- simulation time = 200 s (frame time)
- time step = 0.001 s
- ground restoration constant = 10000 N/m
- dampening constant = 0.99
- ground friction coefficient = 0.7
- gravity = 9.81 N/kg

Robot Parameters:

- mass = 0.5 kg
- length of tetrahedron = 1 m
- soft spring constant = 1000 N/m
- hard spring constant = 5000 N/m
- Gene A range = -1 - 1

- Gene B range = $-\pi/2 - \pi/2$
- Gene C range = $-2*\pi - 2*\pi$

Evolutionary Parameters:

- population size = 128
- mutation probability = 0.9
- number of points for crossover = 2

2.2 Analysis

Simulation:

Overall working with C++ and OpenGL gets more smoothly. The overall speed of simulation is satisfying. However, there is major issue with simulation is that the simulation returns different distance for the robots with the same gene. This might due to the calculation error due to the quick length change in the robot arm causing big forces. When the time step is not small enough to update the following acceleration and velocity change, this error could happen. This also reflect to the learning curve plot where the best fitness in each generation could go down.

Robot:

The tetrahedron was a good way to start, however with simple spring-mass simulation, it is hard to build a "hard" robot. The final result proofed that the robot eventually wriggling on the ground to move instead of using arms to "walk".

Evolutionary Algorithm:

In this part of the project, I am only using two point crossover and simple mutation to evolve the gene for each generation. This actually shows that my EA has poor linkage and diversity. This mainly due to that the structure I designed for simulation is hard to calculate the fitness for one single robot so that deterministic crowding could be used to maintain diversity.

Number of springs evaluations: 810000/s

3 Performance

3.1 Performance Plots

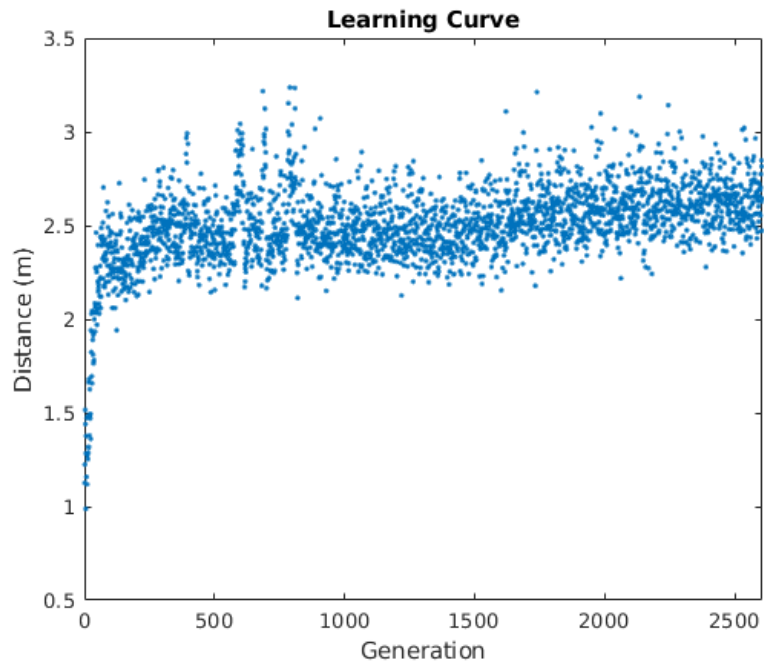


Figure 3: Learning Curve

4 Additional Tasks

4.1 Dot Chart



Figure 4: Dot Chart

4.2 Diversity Chart

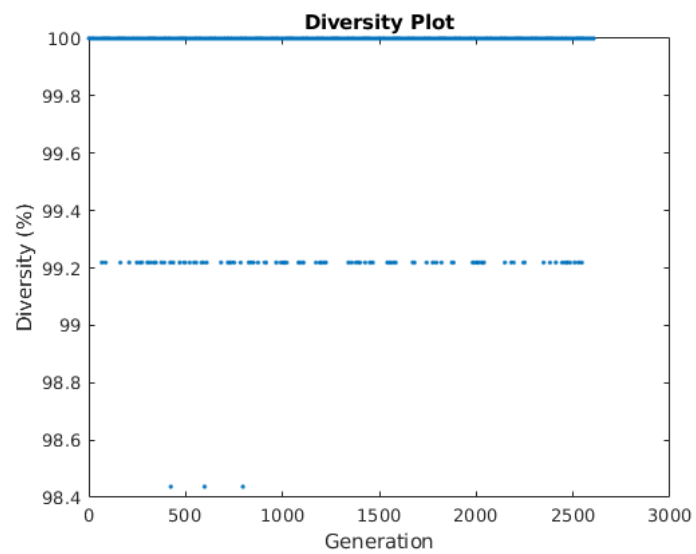


Figure 5: Diversity Chart

5 Appendix

```
1 #include "main.h"
2
3
4 // OPENGGL Variables
5 int th = 0;           // Azimuth of view angle
6 int ph = 0;           // Elevation of view angle
7 int axes = 1;         // Display axes
8 int light = 0;
9 double asp = 1;       // aspect ratio
10 int fov = 40;         // Field of view (for perspective)
11 double dim = 30.0;    // size of the workd
12 double skyBoxScale = 1.0;
13 double cx=0;          // Camera Location
14 double cy=5;
15 double cz=4;
16 double view=1000;
17 double viewlr=90;
18 int mode = 1;
19
20 int generationNumber = 1;
21 int robotNumber = 128;
22 int simulationTime = 400;
23
24 float emission = 60; // Emission intensity (%)
25 float ambient = 60; // Ambient intensity (%)
26 float diffuse = 60; // Diffuse intensity (%)
27 float specular = 60; // Specular intensity (%)
28 float shininess = 64; // Shininess (power of two)
29 float shiny = 1; // Shininess (value)
30 float white[] = {1,1,1,1};
31 float black[] = {0,0,0,1};
32
33 unsigned int grassTexture;
34 unsigned int slimeTexture;
35 unsigned int skyBoxTexture[10]; // Texture for Sky Box
36
37 // Physics Simluator Variables
38 double mass = 0.5;
39 double length = 1;
40 double gravity = 9.8;
41 double T = 0;
42
43 double timeStep = 0.001;
44 double restoreConstant = 10000;
45 double springConstant = 5000;
46 double dampingConstant = 0.99;
47 double frictionCoefficient = 0.7;
48
49 static GLint Frames = 0;
50 static GLfloat fps = -1;
51 static GLint T0 = 0;
52
53 GLfloat worldRotation[16] = {1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1};
54
55 std::ofstream bestGene;
56 std::ofstream popDis;
57
58 // calcualte norm for vector
59 double norm( double x[], std::size_t sz )
60 {
61     return std::sqrt( std::inner_product( x, x+sz, x, 0.0 ) );
62 }
63
64 std::vector<int> sort_indexes(const std::vector<double> &v) {
65
```

```

66     // initialize original index locations
67     std::vector<int> idx(v.size());
68     iota(idx.begin(), idx.end(), 0);
69     // sort indexes based on comparing values in v
70     sort(idx.begin(), idx.end(),
71         [&v](int i1, int i2) {return v[i1] > v[i2];});
72     return idx;
73 }
74
75 class ROBOT
76 {
77 private:
78     std::vector<GENE> gene;
79 public:
80     double initialLocation[3] = {0,0,0};
81     std::vector<MASS> robotMasses;
82     std::vector<SPRING> robotSprings;
83
84     ROBOT(double initialX, double initialY, double initialZ, std::vector<GENE> legGene)
85     {
86         // default constructor
87         initialLocation[0] = initialX; initialLocation[1] = initialY; initialLocation[2] =
            initialZ;
88         gene = legGene;
89         generateRobotMasses(initialX, initialY, initialZ);
90         generateRobotSprings();
91     }
92
93     void generateRobotMasses(double initialX, double initialY, double initialZ)
94     {
95         robotMasses.push_back({mass, {initialX, initialY, initialZ+0.5*length}, {0, 0, 0}, {0, 0,
            0}});
96
97         robotMasses.push_back({mass, {initialX-0.5*length, initialY+0.5*length, initialZ}, {0, 0,
            0}, {0, 0, 0}});
98         robotMasses.push_back({mass, {initialX-0.5*length, initialY-0.5*length, initialZ}, {0, 0,
            0}, {0, 0, 0}});
99         robotMasses.push_back({mass, {initialX+0.5*length, initialY-0.5*length, initialZ}, {0, 0,
            0}, {0, 0, 0}});
100        robotMasses.push_back({mass, {initialX+0.5*length, initialY+0.5*length, initialZ}, {0, 0,
            0}, {0, 0, 0}});
101
102        robotMasses.push_back({mass, {initialX-0.0*length, initialY+1.5*length, initialZ}, {0, 0,
            0}, {0, 0, 0}});
103        robotMasses.push_back({mass, {initialX-1.5*length, initialY-0.0*length, initialZ}, {0, 0,
            0}, {0, 0, 0}});
104        robotMasses.push_back({mass, {initialX+0.0*length, initialY-1.5*length, initialZ}, {0, 0,
            0}, {0, 0, 0}});
105        robotMasses.push_back({mass, {initialX+1.5*length, initialY+0.0*length, initialZ}, {0, 0,
            0}, {0, 0, 0}});
106    }
107
108    void generateRobotSprings()
109    {
110        for (int i = 0; i < robotMasses.size() - 1; i++){
111            for (int j = 1; j < robotMasses.size(); j++){
112                double positionDiff[3] = {robotMasses[j].p[0] - robotMasses[i].p[0], robotMasses[j]
                    .p[1] - robotMasses[i].p[1], robotMasses[j].p[2] - robotMasses[i].p[2]};
113                if (norm(positionDiff, 3) < 1.6*length){
114                    robotSprings.push_back({springConstant, norm(positionDiff, 3), i, j});
115                }
116            }
117        }
118
119        double positionDiff[3] = {robotMasses[0].p[0] - robotMasses[5].p[0], robotMasses[0].p[1] -
            robotMasses[5].p[1], robotMasses[0].p[2] - robotMasses[5].p[2]};
120        robotSprings.push_back({1000, norm(positionDiff, 3), 0, 5});
121        robotSprings.push_back({1000, norm(positionDiff, 3), 0, 6});

```



```

122     robotSprings.push_back({1000,norm(positionDiff,3),0,7});
123     robotSprings.push_back({1000,norm(positionDiff,3),0,8});
124
125 }
126
127 void robotDraw()
128 {
129     glColor3f(1, 0, 0);
130
131     GLUQuadric *quad;
132     quad = gluNewQuadric();
133     for (int i = 0; i < (int)robotMasses.size(); i++) {
134         glPushMatrix();
135         glMultMatrixf(worldRotation);
136         glTranslated(robotMasses[i].p[0], robotMasses[i].p[1], robotMasses[i].p[2]);
137         gluSphere(quad, length / 20, 10, 10);
138         glPopMatrix();
139     }
140
141     for (int i = 0; i < (int)robotSprings.size(); i++) {
142         glPushMatrix();
143         glMultMatrixf(worldRotation);
144         glBegin(GL_LINES);
145         glVertex3f(GLfloat(robotMasses[robotSprings[i].m1].p[0]), GLfloat(robotMasses[
            robotSprings[i].m1].p[1]), GLfloat(robotMasses[robotSprings[i].m1].p[2]));
146         glVertex3f(GLfloat(robotMasses[robotSprings[i].m2].p[0]), GLfloat(robotMasses[
            robotSprings[i].m2].p[1]), GLfloat(robotMasses[robotSprings[i].m2].p[2]));
147         glEnd();
148         glPopMatrix();
149     }
150
151     // draw planes
152     glPushMatrix();
153     glMultMatrixf(worldRotation);
154     glBegin(GL_QUADS);
155     glColor3f(0.2, 0.1, 0.3);
156     glVertex3f(GLfloat(robotMasses[1].p[0]), GLfloat(robotMasses[1].p[1]), GLfloat(
        robotMasses[1].p[2]));
157     glVertex3f(GLfloat(robotMasses[2].p[0]), GLfloat(robotMasses[2].p[1]), GLfloat(
        robotMasses[2].p[2]));
158     glVertex3f(GLfloat(robotMasses[3].p[0]), GLfloat(robotMasses[3].p[1]), GLfloat(
        robotMasses[3].p[2]));
159     glVertex3f(GLfloat(robotMasses[4].p[0]), GLfloat(robotMasses[4].p[1]), GLfloat(
        robotMasses[4].p[2]));
160     glEnd();
161     glBegin(GL_TRIANGLES);
162     glColor3f(0.2, 0.1, 0.3);
163     glVertex3f(GLfloat(robotMasses[0].p[0]), GLfloat(robotMasses[0].p[1]), GLfloat(
        robotMasses[0].p[2]));
164     glVertex3f(GLfloat(robotMasses[1].p[0]), GLfloat(robotMasses[1].p[1]), GLfloat(
        robotMasses[1].p[2]));
165     glVertex3f(GLfloat(robotMasses[2].p[0]), GLfloat(robotMasses[2].p[1]), GLfloat(
        robotMasses[2].p[2]));
166     glEnd();
167     glBegin(GL_TRIANGLES);
168     glColor3f(0.2, 0.1, 0.3);
169     glVertex3f(GLfloat(robotMasses[0].p[0]), GLfloat(robotMasses[0].p[1]), GLfloat(
        robotMasses[0].p[2]));
170     glVertex3f(GLfloat(robotMasses[2].p[0]), GLfloat(robotMasses[2].p[1]), GLfloat(
        robotMasses[2].p[2]));
171     glVertex3f(GLfloat(robotMasses[3].p[0]), GLfloat(robotMasses[3].p[1]), GLfloat(
        robotMasses[3].p[2]));
172     glEnd();
173     glBegin(GL_TRIANGLES);
174     glColor3f(0.2, 0.1, 0.3);
175     glVertex3f(GLfloat(robotMasses[0].p[0]), GLfloat(robotMasses[0].p[1]), GLfloat(
        robotMasses[0].p[2]));

```

```

176     glVertex3f(GLfloat(robotMasses[3].p[0]), GLfloat(robotMasses[3].p[1]), GLfloat(
177         robotMasses[3].p[2]));
178     glVertex3f(GLfloat(robotMasses[4].p[0]), GLfloat(robotMasses[4].p[1]), GLfloat(
179         robotMasses[4].p[2]));
180     glEnd();
181     glBegin(GL_TRIANGLES);
182     glColor3f(0.2, 0.1, 0.3);
183     glVertex3f(GLfloat(robotMasses[0].p[0]), GLfloat(robotMasses[0].p[1]), GLfloat(
184         robotMasses[0].p[2]));
185     glVertex3f(GLfloat(robotMasses[4].p[0]), GLfloat(robotMasses[4].p[1]), GLfloat(
186         robotMasses[4].p[2]));
187     glVertex3f(GLfloat(robotMasses[1].p[0]), GLfloat(robotMasses[1].p[1]), GLfloat(
188         robotMasses[1].p[2]));
189     glEnd();
190     glBegin(GL_TRIANGLES);
191     glColor3f(0.2, 0.1, 0.3);
192     glVertex3f(GLfloat(robotMasses[1].p[0]), GLfloat(robotMasses[1].p[1]), GLfloat(
193         robotMasses[1].p[2]));
194     glVertex3f(GLfloat(robotMasses[2].p[0]), GLfloat(robotMasses[2].p[1]), GLfloat(
195         robotMasses[2].p[2]));
196     glVertex3f(GLfloat(robotMasses[6].p[0]), GLfloat(robotMasses[6].p[1]), GLfloat(
197         robotMasses[6].p[2]));
198     glEnd();
199     glBegin(GL_TRIANGLES);
200     glColor3f(0.2, 0.1, 0.3);
201     glVertex3f(GLfloat(robotMasses[2].p[0]), GLfloat(robotMasses[2].p[1]), GLfloat(
202         robotMasses[2].p[2]));
203     glVertex3f(GLfloat(robotMasses[3].p[0]), GLfloat(robotMasses[3].p[1]), GLfloat(
204         robotMasses[3].p[2]));
205     glVertex3f(GLfloat(robotMasses[7].p[0]), GLfloat(robotMasses[7].p[1]), GLfloat(
206         robotMasses[7].p[2]));
207     glEnd();
208     glBegin(GL_TRIANGLES);
209     glColor3f(0.2, 0.1, 0.3);
210     glVertex3f(GLfloat(robotMasses[3].p[0]), GLfloat(robotMasses[3].p[1]), GLfloat(
211         robotMasses[3].p[2]));
212     glVertex3f(GLfloat(robotMasses[4].p[0]), GLfloat(robotMasses[4].p[1]), GLfloat(
213         robotMasses[4].p[2]));
214     glVertex3f(GLfloat(robotMasses[8].p[0]), GLfloat(robotMasses[8].p[1]), GLfloat(
215         robotMasses[8].p[2]));
216     glEnd();
217     glBegin(GL_TRIANGLES);
218     glColor3f(0.2, 0.1, 0.3);
219     glVertex3f(GLfloat(robotMasses[1].p[0]), GLfloat(robotMasses[1].p[1]), GLfloat(
220         robotMasses[1].p[2]));
221     glVertex3f(GLfloat(robotMasses[4].p[0]), GLfloat(robotMasses[4].p[1]), GLfloat(
222         robotMasses[4].p[2]));
223     glVertex3f(GLfloat(robotMasses[5].p[0]), GLfloat(robotMasses[5].p[1]), GLfloat(
224         robotMasses[5].p[2]));
225     glEnd();
226     glPopMatrix();

    // draw line between middle point and initial position
    double x = 0; double y = 0; double z = 0;
    for (int j = 1; j < 5; j++){
        x = x + robotMasses[j].p[0];
        y = y + robotMasses[j].p[1];
        z = z + robotMasses[j].p[2];
    }
    x = x / 4;
    y = y / 4;
    z = z / 4;
    glPushMatrix();
    glMultMatrixf(worldRotation);
    glBegin(GL_LINES);
    glColor3f(0.0, 1.0, 0.0);

```

```

227     glVertex3f(GLfloat(x), GLfloat(y), GLfloat(z));
228     glVertex3f(GLfloat(initialLocation[0]), GLfloat(initialLocation[1]), GLfloat(
        initialLocation[2]));
229     glEnd();
230     glPopMatrix();
231     Frames++;
232     GLint t = glutGet(GLUT_ELAPSED_TIME);
233     if (t - T0 >= 1000) {
234         GLfloat seconds = (t - T0) / 1000.0;
235         fps = Frames / seconds;
236         //printf("%d frames in %6.3f seconds = %6.3f FPS\n", Frames, seconds, fps);
237         T0 = t;
238         Frames = 0;
239     }
240 }
241
242 void robotUpdate()
243 {
244     // initialize the force vector with value 0
245     std::vector<std::vector<double>> robotForces((int)robotMasses.size(), std::vector<double
        >(3));
246     if (T > 1.0) {
247         robotSprings[robotSprings.size()-4].L_0 = 1.58114 + gene[0].A / (2 * M_PI) * length *
            cos(gene[0].B / 4 * T + gene[0].C);
248         robotSprings[robotSprings.size()-3].L_0 = 1.58114 + gene[1].A / (2 * M_PI) * length *
            cos(gene[1].B / 4 * T + gene[1].C);
249         robotSprings[robotSprings.size()-2].L_0 = 1.58114 + gene[2].A / (2 * M_PI) * length *
            cos(gene[2].B / 4 * T + gene[2].C);
250         robotSprings[robotSprings.size()-1].L_0 = 1.58114 + gene[3].A / (2 * M_PI) * length *
            cos(gene[3].B / 4 * T + gene[3].C);
251     }
252
253     // loop through all springs to calculate spring forces
254     // #pragma omp parallel for
255     for (int i = 0; i < (int)robotSprings.size(); i++) {
256         MASS mass1 = robotMasses[robotSprings[i].m1];
257         MASS mass2 = robotMasses[robotSprings[i].m2];
258         double positionDiff[3] = {mass2.p[0] - mass1.p[0], mass2.p[1] - mass1.p[1], mass2.p
            [2] - mass1.p[2]};
259         double L = norm(positionDiff, 3);
260         double force = robotSprings[i].k * fabs(robotSprings[i].L_0 - L);
261         double direction[3] = {positionDiff[0] / L, positionDiff[1] / L, positionDiff[2] / L
            };
262         // contraction case
263         if (L > robotSprings[i].L_0) {
264             robotForces[robotSprings[i].m1][0] = robotForces[robotSprings[i].m1][0] +
                direction[0] * force;
265             robotForces[robotSprings[i].m1][1] = robotForces[robotSprings[i].m1][1] +
                direction[1] * force;
266             robotForces[robotSprings[i].m1][2] = robotForces[robotSprings[i].m1][2] +
                direction[2] * force;
267             robotForces[robotSprings[i].m2][0] = robotForces[robotSprings[i].m2][0] -
                direction[0] * force;
268             robotForces[robotSprings[i].m2][1] = robotForces[robotSprings[i].m2][1] -
                direction[1] * force;
269             robotForces[robotSprings[i].m2][2] = robotForces[robotSprings[i].m2][2] -
                direction[2] * force;
270         }
271         // expansion case
272         else if (L < robotSprings[i].L_0) {
273             robotForces[robotSprings[i].m1][0] = robotForces[robotSprings[i].m1][0] -
                direction[0] * force;
274             robotForces[robotSprings[i].m1][1] = robotForces[robotSprings[i].m1][1] -
                direction[1] * force;
275             robotForces[robotSprings[i].m1][2] = robotForces[robotSprings[i].m1][2] -
                direction[2] * force;
276             robotForces[robotSprings[i].m2][0] = robotForces[robotSprings[i].m2][0] +
                direction[0] * force;

```

```

277         robotForces[robotSprings[i].m2][1] = robotForces[robotSprings[i].m2][1] +
            direction[1] * force;
278         robotForces[robotSprings[i].m2][2] = robotForces[robotSprings[i].m2][2] +
            direction[2] * force;
279     }
280 }
281
282 // #pragma omp parallel for
283 for (int i = 0; i < (int)robotMasses.size(); i++) {
284     // add gravity
285     robotForces[i][2] = robotForces[i][2] - robotMasses[i].m * gravity;
286     // if the mass is below ground, add restoration force and calculate friction
287     if (robotMasses[i].p[2] <= 0) {
288         robotForces[i][2] = robotForces[i][2] + restoreConstant * fabs(robotMasses[i].p
            [2]);
289         // calculate horizontal force and vertical force
290         double F_h = sqrt(pow(robotForces[i][0], 2) + pow(robotForces[i][1], 2));
291         double F_v = robotForces[i][2];
292         if (F_h < F_v * frictionCoefficient) {
293             robotForces[i][0] = 0;
294             robotForces[i][1] = 0;
295             robotMasses[i].v[0] = 0;
296             robotMasses[i].v[1] = 0;
297         }
298     }
299     // update acceleration
300     robotMasses[i].a[0] = robotForces[i][0] / robotMasses[i].m;
301     robotMasses[i].a[1] = robotForces[i][1] / robotMasses[i].m;
302     robotMasses[i].a[2] = robotForces[i][2] / robotMasses[i].m;
303     // update velocity
304     robotMasses[i].v[0] = dampingConstant * (robotMasses[i].v[0] + robotMasses[i].a[0] *
        timeStep);
305     robotMasses[i].v[1] = dampingConstant * (robotMasses[i].v[1] + robotMasses[i].a[1] *
        timeStep);
306     robotMasses[i].v[2] = dampingConstant * (robotMasses[i].v[2] + robotMasses[i].a[2] *
        timeStep);
307     // update position
308     robotMasses[i].p[0] = robotMasses[i].p[0] + robotMasses[i].v[0] * timeStep;
309     robotMasses[i].p[1] = robotMasses[i].p[1] + robotMasses[i].v[1] * timeStep;
310     robotMasses[i].p[2] = robotMasses[i].p[2] + robotMasses[i].v[2] * timeStep;
311 }
312 // update time
313 T = T + timeStep;
314 }
315
316 };
317
318 class Simulation{
319 private:
320     int populationSize;
321     std::vector<double> populationDistance;
322     std::vector<std::vector<GENE>> populationGene;
323     std::vector<std::vector<GENE>> newPopulationGene;
324     std::vector<ROBOT> robots;
325 public:
326     double averageDistance;
327     double maxDistance;
328
329     Simulation(int popSize)
330     {
331         populationSize = popSize;
332         generateGenes();
333         //generateBestGene();
334         generateRobots();
335         popDis.open ("populationDistance.txt"); popDis.close();
336         bestGene.open ("bestGene.txt"); bestGene.close();
337         popDis.open ("populationDistance.txt", std::ios_base::app);

```

```

339     bestGene.open("bestGene.txt", std::ios_base::app);
340 }
341
342 void startSim(double time){
343     if (T < time){
344         simUpdate();
345         simDraw();
346     }
347     else {
348         calculatePopulationDistance();
349         selection();
350         crossOver();
351         populationGene.clear(); populationGene.shrink_to_fit();
352         populationGene = newPopulationGene;
353         generationNumber++;
354         robots.clear(); robots.shrink_to_fit();
355         generateRobots();
356         T = 0;
357     }
358 }
359
360 void selection(){
361     std::vector<int> index = sort_indexes(populationDistance);
362     newPopulationGene.clear();
363     newPopulationGene.shrink_to_fit();
364     for (int i = 0; i < index.size()/2; i++) {
365         //std::cout << index[i] << std::endl;
366         newPopulationGene.push_back(populationGene[index[i]]);
367     }
368     for (int i = 0; i < newPopulationGene[0].size(); i++){
369         bestGene << newPopulationGene[0][i].A << " " << newPopulationGene[0][i].B << " " <<
            newPopulationGene[0][i].C << " ";
370     }
371     bestGene << "\n";
372 }
373
374 void crossOver(){
375     for (int n = 0; n < populationGene.size() / 2; n++){
376         int parentIndex1 = rand() % static_cast<int>(newPopulationGene.size());
377         int parentIndex2 = rand() % static_cast<int>(newPopulationGene.size());
378         std::vector<double> parent1, parent2;
379         for (int i = 0; i < newPopulationGene[parentIndex1].size(); i++){
380             parent1.push_back(newPopulationGene[parentIndex1][i].A);
381             parent1.push_back(newPopulationGene[parentIndex1][i].B);
382             parent1.push_back(newPopulationGene[parentIndex1][i].C);
383             parent2.push_back(newPopulationGene[parentIndex2][i].A);
384             parent2.push_back(newPopulationGene[parentIndex2][i].B);
385             parent2.push_back(newPopulationGene[parentIndex2][i].C);
386         }
387         int crossOverPoint1 = rand() % static_cast<int>(parent1.size());
388         int crossOverPoint2 = rand() % static_cast<int>(parent1.size());
389         if (crossOverPoint2 < crossOverPoint1){
390             int temp = crossOverPoint1;
391             crossOverPoint1 = crossOverPoint2;
392             crossOverPoint2 = temp;
393         }
394         std::vector<double> offSpring1, offSpring2;
395         for (int i = 0; i < crossOverPoint1; i++){
396             offSpring1.push_back(parent1[i]);
397             offSpring2.push_back(parent2[i]);
398         }
399         for (int i = crossOverPoint1; i < crossOverPoint2; i++){
400             offSpring1.push_back(parent2[i]);
401             offSpring2.push_back(parent1[i]);
402         }
403         for (int i = crossOverPoint2; i < parent1.size(); i++){
404             offSpring1.push_back(parent1[i]);
405             offSpring2.push_back(parent2[i]);

```

```

406     }
407     offSpring1 = mutation(offSpring1);
408     offSpring2 = mutation(offSpring2);
409     std::vector<GENE> offSpringGene1, offSpringGene2;
410     GENE temp1, temp2;
411     for (int i = 0; i < offSpring1.size(); i = i + 3){
412         temp1.A = offSpring1[i];
413         temp1.B = offSpring1[i+1];
414         temp1.C = offSpring1[i+2];
415         temp2.A = offSpring2[i];
416         temp2.B = offSpring2[i+1];
417         temp2.C = offSpring2[i+2];
418         offSpringGene1.push_back(temp1);
419         offSpringGene2.push_back(temp2);
420     }
421     newPopulationGene.push_back(offSpringGene1);
422     newPopulationGene.push_back(offSpringGene2);
423 }
424 }
425
426 std::vector<double> mutation(std::vector<double> offSpring){
427     for (int i = 0; i < offSpring.size(); i++){
428         double mutationProbability = static_cast<float>(rand()) / (static_cast<float>(
429             RAND_MAX/1.0));
430         if (mutationProbability > 0.9){
431             offSpring[i] = -2*M_PI + static_cast<float>(rand()) / (static_cast<float>(
432                 RAND_MAX/(4*M_PI)));
433         }
434     }
435     return offSpring;
436 }
437
438 void generateBestGene(){
439     for (int i = 0; i < populationSize; i++) {
440         std::vector<GENE> tempVec1;
441         GENE temp1{6.26142, -6.27729, 1.23343};
442         tempVec1.push_back(temp1);
443         GENE temp2{-0.0439628, -0.281218, 1.16934};
444         tempVec1.push_back(temp2);
445         GENE temp3{-0.00115234, -2.94204, 6.13413};
446         tempVec1.push_back(temp3);
447         GENE temp4{-5.13257, -0.0672038, 1.83822};
448         tempVec1.push_back(temp4);
449         populationGene.push_back(tempVec1);
450     }
451 }
452
453 void generateGenes(){
454     srand(time(0));
455     for (int i = 0; i < populationSize; i++){
456         std::vector<GENE> tempVec;
457         for (int j = 0; j < 4; j++){
458             double A = -2*M_PI + static_cast<float>(rand()) / (static_cast<float>(
459                 RAND_MAX/(4*M_PI))); // actual assign -1 - 1
460             double B = -2*M_PI + static_cast<float>(rand()) / (static_cast<float>(
461                 RAND_MAX/(4*M_PI))); // actual assign -pi/2 - pi/2
462             double C = -2*M_PI + static_cast<float>(rand()) / (static_cast<float>(
463                 RAND_MAX/(4*M_PI))); // acutal assign -2pi - 2pi
464             GENE temp{A,B,C};
465             tempVec.push_back(temp);
466         }
467         populationGene.push_back(tempVec);
468     }
469 }
470
471 void generateRobots(){
472     for (int i = 0; i < populationSize; i++){

```

```

469         double X = -20 + static_cast<float>(rand()) / (static_cast<float>(RAND_MAX/40));
470         double Y = -20 + static_cast<float>(rand()) / (static_cast<float>(RAND_MAX/40));
471         //robots.push_back(ROBOT(0.0, 3.0*(i-populationSize/2), 0.0, populationGene[i]));
472         robots.push_back(ROBOT(X, Y, 0.0, populationGene[i]));
473     }
474 }
475
476 void simUpdate(){
477     #pragma omp parallel for num_threads(32)
478     for (int i = 0; i < populationSize; i++){
479         robots[i].robotUpdate();
480     }
481 }
482
483 void simDraw(){
484     for (int i = 0; i < populationSize; i++){
485         robots[i].robotDraw();
486     }
487 }
488
489 void calculatePopulationDistance(){
490     populationDistance.clear();
491     populationDistance.shrink_to_fit();
492     for (int i = 0; i < populationSize; i++){
493         double x = 0; double y = 0;
494         for (int j = 1; j < 5; j++){
495             x = x + robots[i].robotMasses[j].p[0];
496             y = y + robots[i].robotMasses[j].p[1];
497         }
498         x = x / 4;
499         y = y / 4;
500         double distance[2] = {fabs(x - robots[i].initialLocation[0]), fabs(y - robots[i].
                    initialLocation[1])};
501         double distanceNorm = norm(distance, 2);
502         populationDistance.push_back(distanceNorm);
503     }
504     averageDistance = 0;
505     maxDistance = 0;
506     std::cout << "#####\n" << std::endl;
507     for (int i = 0; i < populationSize; i++){
508         averageDistance = averageDistance + populationDistance[i];
509         maxDistance = std::max(maxDistance, populationDistance[i]);
510         //std::cout << populationDistance[i] << std::endl;
511         popDis << populationDistance[i] << " ";
512     }
513     popDis << "\n";
514     averageDistance = averageDistance/populationSize;
515     std::cout << "Maximum Distance: " << maxDistance << std::endl;
516     std::cout << "Average Distance: " << averageDistance << std::endl;
517 }
518 };
519
520 Simulation sim1(robotNumber);
521
522
523 void Print(const char* format , ...)
524 {
525     char    buf[LEN];
526     char*   ch=buf;
527     va_list args;
528     // Turn the parameters into a character string
529     va_start(args,format);
530     vsnprintf(buf,LEN,format,args);
531     va_end(args);
532     // Display the characters one at a time at the current raster position
533     while (*ch)
534         glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24,*ch++);
535 }

```

```

536
537 void drawGrid(){
538     for (int i = -dim/2; i < dim/2 + 1; i++) {
539         for (int j = -dim / 2; j < dim / 2 + 1; j++) {
540             float white[] = {1,1,1,1};
541             float black[] = {0,0,0,1};
542             glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, shiny);
543             glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, white);
544             glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, black);
545
546             glPushMatrix();
547             glTranslatef(i*2, 0, j*2);
548             glBegin(GL_QUADS);
549             //
550             glNormal3f(0, 1, 0);
551             glColor3f(0.0, 0.0, 0.0);
552             glVertex3f(+0, -0.01, +0);
553             glVertex3f(+1, -0.01, +0);
554             glVertex3f(+1, -0.01, +1);
555             glVertex3f(+0, -0.01, +1);
556             //
557             glNormal3f(0, 1, 0);
558             glColor3f(0.5, 0.5, 0.5);
559             glVertex3f(-1, -0.01, +0);
560             glVertex3f(+0, -0.01, +0);
561             glVertex3f(+0, -0.01, +1);
562             glVertex3f(-1, -0.01, +1);
563             //
564             glNormal3f(0, 1, 0);
565             glColor3f(0.0, 0.0, 0.0);
566             glVertex3f(-1, -0.01, -1);
567             glVertex3f(+0, -0.01, -1);
568             glVertex3f(+0, -0.01, +0);
569             glVertex3f(-1, -0.01, +0);
570             //
571             glNormal3f(0, 1, 0);
572             glColor3f(0.5, 0.5, 0.5);
573             glVertex3f(-0, -0.01, -1);
574             glVertex3f(+1, -0.01, -1);
575             glVertex3f(+1, -0.01, +0);
576             glVertex3f(-0, -0.01, +0);
577             glEnd();
578             glPopMatrix();
579         }
580     }
581 }
582
583
584 static void ball(double x,double y,double z,double r)
585 {
586     // Save transformation
587     glPushMatrix();
588     // Offset, scale and rotate
589     glTranslated(x,y,z);
590     glScaled(r,r,r);
591     // White ball
592     glColor3f(1,1,1);
593     glutSolidSphere(1.0,16,16);
594     // Undo transformations
595     glPopMatrix();
596 }
597
598 void display()
599 {
600     const double len=2.0; // Length of axes
601     // Erase the window and the depth buffer
602     glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
603     // Enable Z-buffering in OpenGL

```



```

604     glEnable(GL_DEPTH_TEST);
605     // Undo previous transformations
606     glLoadIdentity();
607     // Eye position
608     if (mode == 1) {
609         // Eye position
610         double Ex = -2*dim*sin(th)*cos(ph);
611         double Ey = +2*dim*cos(th)*sin(ph);
612         double Ez = +2*dim*cos(th)*cos(ph);
613         gluLookAt(Ex,Ey,Ez, 0,0,0, 0,cos(ph),0);
614     }
615     if (mode == 2) {
616         gluLookAt(cx,cy,cz,cx+view*cos(viewlr),cy,cz+view*sin(viewlr),0,1,0);
617     }
618
619     if (light)
620     {
621         // Translate intensity to color vectors
622         float Ambient[] = {(float)0.01*ambient, (float)0.01*ambient, (float)0.01*ambient ,2.0};
623         float Diffuse[] = {(float)0.01*diffuse, (float)0.01*diffuse, (float)0.01*diffuse ,2.0};
624         float Specular[] = {(float)0.01*specular, (float)0.01*specular, (float)0.01*specular,2.0};
625         // Light direction
626         //float Position[] = {0,10,0};
627         //float Position[] = {5*cos(zh),ylight,5*sin(zh),1};
628         float Position[] = {95,90,60,1};
629         // Draw light position as ball (still no lighting here)
630         glColor3f(16,16,16);
631         ball(Position[0],Position[1],Position[2], 5);
632         // OpenGL should normalize normal vectors
633         glEnable(GL_NORMALIZE);
634         // Enable lighting
635         glEnable(GL_LIGHTING);
636         // glColor sets ambient and diffuse color materials
637         glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);
638         glEnable(GL_COLOR_MATERIAL);
639         // Enable light 0
640         glEnable(GL_LIGHT0);
641         // Set ambient, diffuse, specular components and position of light 0
642         glLightfv(GL_LIGHT0, GL_AMBIENT ,Ambient);
643         glLightfv(GL_LIGHT0, GL_DIFFUSE ,Diffuse);
644         glLightfv(GL_LIGHT0, GL_SPECULAR, Specular);
645         glLightfv(GL_LIGHT0, GL_POSITION, Position);
646     }
647     else {
648         glDisable(GL_LIGHTING);
649     }
650
651     drawGrid();
652     glColor3f(1,1,1);
653     glWindowPos2i(0,0);
654     Print("Generation: %d", generationNumber);
655     glWindowPos2i(850, 0);
656     Print("Max: %4.2f", sim1.maxDistance);
657     glWindowPos2i(850, 50);
658     Print("Average %4.2f", sim1.averageDistance);
659     sim1.startSim(simulationTime);
660
661     glRasterPos3d(0.0,2,0.0);
662     if (fps>0) Print("FPS %.3f", fps);
663     //drawGrass();
664     //skyBox(skyBoxScale);
665     // Draw axes
666     glColor3f(1,1,1);
667     if (axes)
668     {
669         glBegin(GL_LINES);
670         glVertex3d(0.0,0.0,0.0);
671         glVertex3d(len,0.0,0.0);

```

```

672         glVertex3d(0.0,0.0,0.0);
673         glVertex3d(0.0,len,0.0);
674         glVertex3d(0.0,0.0,0.0);
675         glVertex3d(0.0,0.0,len);
676         glEnd();
677         // Label axes
678         glRasterPos3d(len,0.0,0.0);
679         Print("X");
680         glRasterPos3d(0.0,len,0.0);
681         Print("Y");
682         glRasterPos3d(0.0,0.0,len);
683         Print("Z");
684     }
685     // Render the scene
686     glFlush();
687     // Make the rendered scene visible
688     glutSwapBuffers();
689 }
690
691 /*
692  * GLUT calls this routine when an arrow key is pressed
693  */
694 void special(int key,int x,int y)
695 {
696     // Right arrow key - increase angle by 5 degrees
697     if (key == GLUT_KEY_RIGHT)
698         th += 5;
699     // Left arrow key - decrease angle by 5 degrees
700     else if (key == GLUT_KEY_LEFT)
701         th -= 5;
702     // Up arrow key - increase elevation by 5 degrees
703     else if (key == GLUT_KEY_UP)
704     {
705         if (ph +5 < 90)
706         {
707             ph += 5;
708         }
709     }
710     // Down arrow key - decrease elevation by 5 degrees
711     else if (key == GLUT_KEY_DOWN)
712     {
713         if (ph-5>0)
714         {
715             ph -= 5;
716         }
717     }
718     // Keep angles to +/-360 degrees
719     th %= 360;
720     ph %= 360;
721     // Tell GLUT it is necessary to redisplay the scene
722     glutPostRedisplay();
723 }
724
725 /*
726  * Set projection
727  */
728 void Project(double fov,double asp,double dim)
729 {
730     // Tell OpenGL we want to manipulate the projection matrix
731     glMatrixMode(GL_PROJECTION);
732     // Undo previous transformations
733     glLoadIdentity();
734     // Perspective transformation
735     if (fov)
736         gluPerspective(fov,asp,dim/16,16*dim);
737     // Orthogonal transformation
738     else
739         glOrtho(-asp*dim,asp*dim,-dim,+dim,-dim,+dim);

```

```

740     // Switch to manipulating the model matrix
741     glMatrixMode(GL_MODELVIEW);
742     // Undo previous transformations
743     glLoadIdentity();
744 }
745
746 /*
747  * GLUT calls this routine when a key is pressed
748  */
749 void key(unsigned char ch,int x,int y)
750 {
751     // Exit on ESC
752     if (ch == 27)
753         exit(0);
754     // Reset view angle
755     else if (ch == '0')
756         th = ph = 0;
757     // Toggle axes
758     else if (ch == 'a' || ch == 'A')
759         //axes = 1-axes;
760         int x;
761     // Change field of view angle
762     else if (ch == '-' && ch>1)
763         fov++;
764     else if (ch == '=' && ch<179)
765         fov--;
766     // PageUp key - increase dim
767     else if (ch == GLUT_KEY_PAGE_DOWN){
768         dim += 0.1;
769     }
770     // PageDown key - decrease dim
771     else if (ch == GLUT_KEY_PAGE_UP && dim>1){
772         dim -= 0.1;
773     }
774     else if (ch == 'w' || ch == 'W'){
775         cx += 0.3*cos(viewlr);
776         cz += 0.3*sin(viewlr);
777     }
778     else if (ch == 'a' || ch == 'A'){
779         cx += 0.3*sin(viewlr);
780         cz -= 0.3*cos(viewlr);
781     }
782     else if (ch == 's' || ch == 'S'){
783         cx -= 0.3*cos(viewlr);
784         cz -= 0.3*sin(viewlr);
785     }
786     else if (ch == 'd' || ch == 'D'){
787         cx -= 0.3*sin(viewlr);
788         cz += 0.3*cos(viewlr);
789     }
790     else if (ch == 'r' || ch == 'R'){
791         if (cy+0.3 < skyBoxScale*10){
792             cy += 0.3;
793         }
794     }
795     else if (ch == 'f' || ch == 'F'){
796         if (cy-0.3 > 0){
797             cy -= 0.3;
798         }
799     }
800     else if (ch == 'q' || ch=='Q'){
801         viewlr-=15;
802     }
803     else if (ch == 'e' || ch=='E'){
804         viewlr+=15;
805     }
806
807     if (ch == 'l')

```

```

808     {
809         mode = 1;
810     }
811     else if (ch == '2')
812     {
813         mode = 2;
814     }
815     // Keep angles to +/-360 degrees
816     th %= 360;
817     ph %= 360;
818     // Reproject
819     Project(fov,asp,dim);
820     // Tell GLUT it is necessary to redisplay the scene
821     glutPostRedisplay();
822 }
823
824 /*
825  * GLUT calls this routine when the window is resized
826  */
827 void reshape(int width,int height)
828 {
829     // Ratio of the width to the height of the window
830     asp = (height>0) ? (double)width/height : 1;
831     // Set the viewport to the entire window
832     glViewport(0,0, width,height);
833     // Set projection
834     Project(fov,asp,dim);
835 }
836
837 /*
838  * GLUT calls this routine when there is nothing else to do
839  */
840 void idle()
841 {
842     glutPostRedisplay();
843 }
844
845 int main(int argc,char* argv[]) {
846     // Initialize GLUT and process user parameters
847     glutInit(&argc, argv);
848     glWindowPos2i = (PFNGLWINDOWPOS2IPROC) glutGetProcAddress("glWindowPos2i");
849     // double buffered, true color 600*600
850     glutInitWindowSize(1000,800);
851     glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
852     // create the window
853     glutCreateWindow("hz2547");
854     // Tell GLUT to call "idle" when there is nothing else to do
855     glutIdleFunc(idle);
856     // Tell GLUT to call "display" when the scene should be drawn
857     glutDisplayFunc(display);
858     // Tell GLUT to call "reshape" when the window is resized
859     glutReshapeFunc(reshape);
860     // Tell GLUT to call "special" when an arrow key is pressed
861     glutSpecialFunc(special);
862     // Tell GLUT to call "key" when a key is pressed
863     glutKeyboardFunc(key);
864     // Pass control to GLUT so it can interact with the user
865     glutMainLoop();
866
867     //std::cout << "Hello" << std::endl;
868     //vecAdd_wrapper();
869     return 0;
870 }

```