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

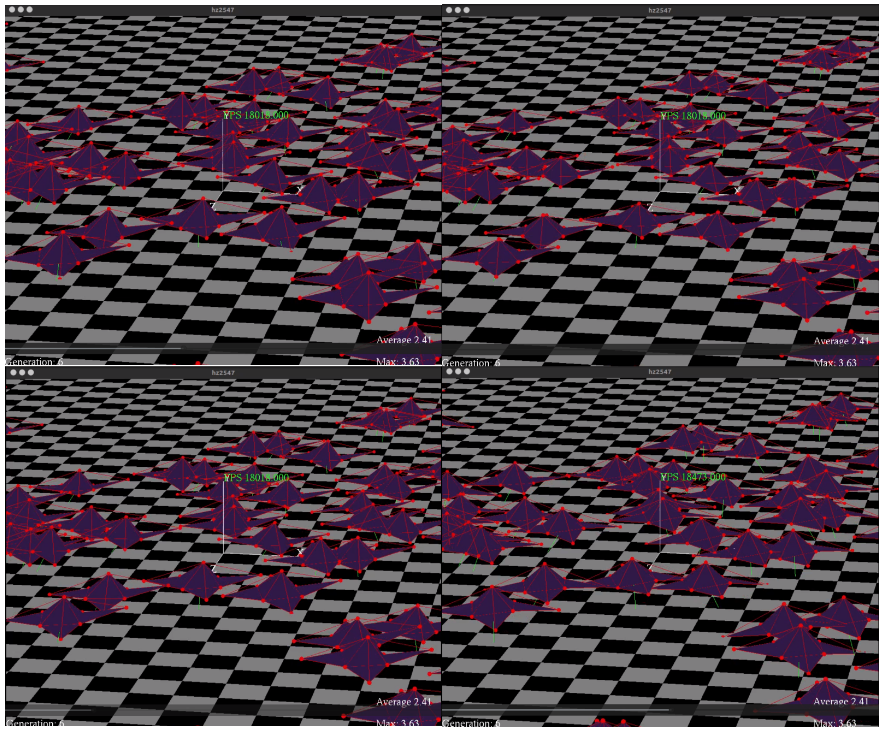

Figure 1: Group of Fastest Robots Moving 0.0163m/s

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

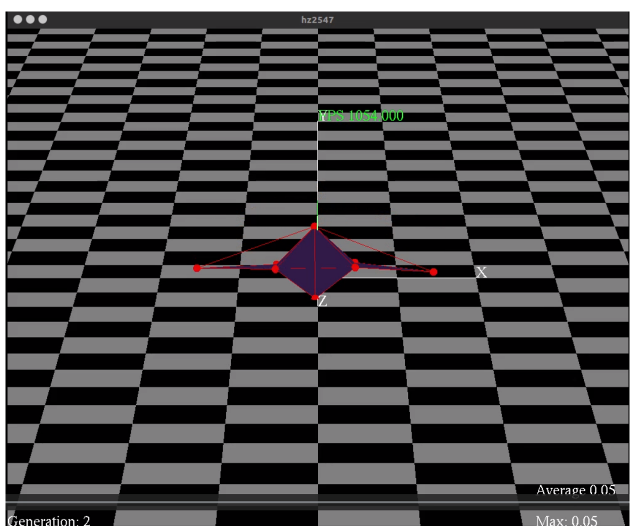

Figure 2: Bouncing Test

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:

$$L = L_0 + A * sin(B * t + C) \tag{1}$$

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 = -π/2 - π/2
- Gene C range = -2*π - 2*π

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

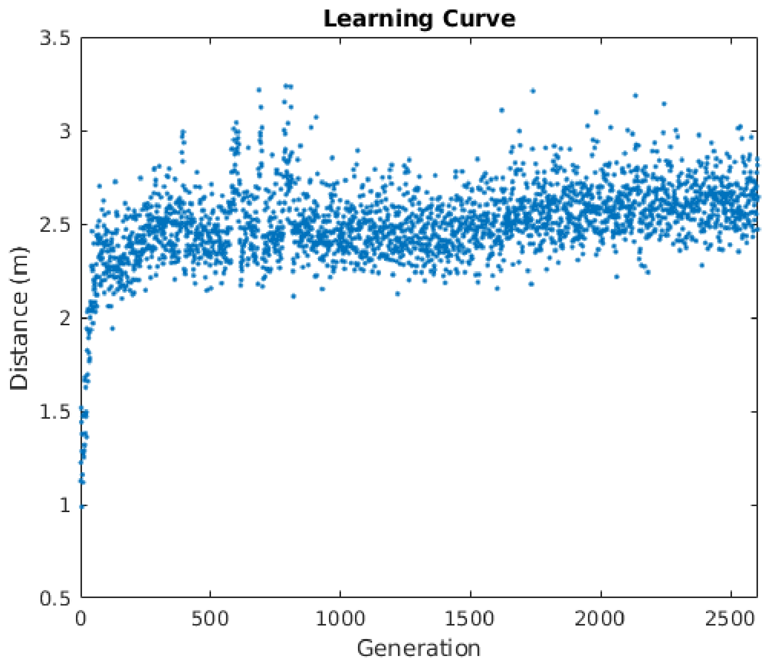

Figure 3: Learning Curve

4 Additional Tasks

4.1 Dot Chart

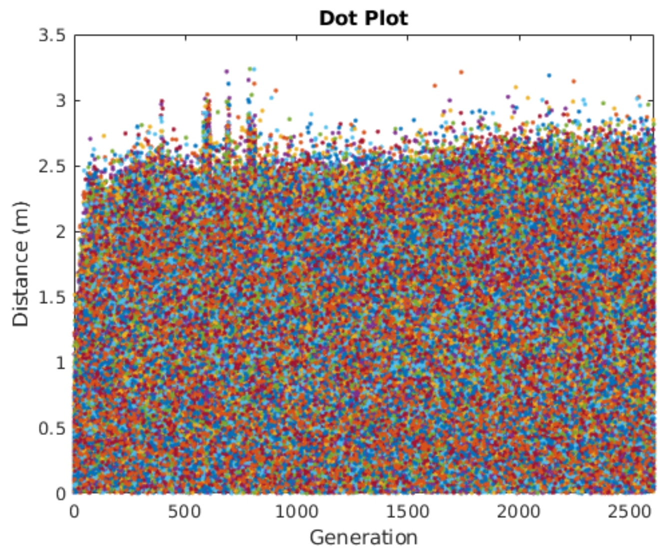

Figure 4: Dot Chart

4.2 Diversity Chart

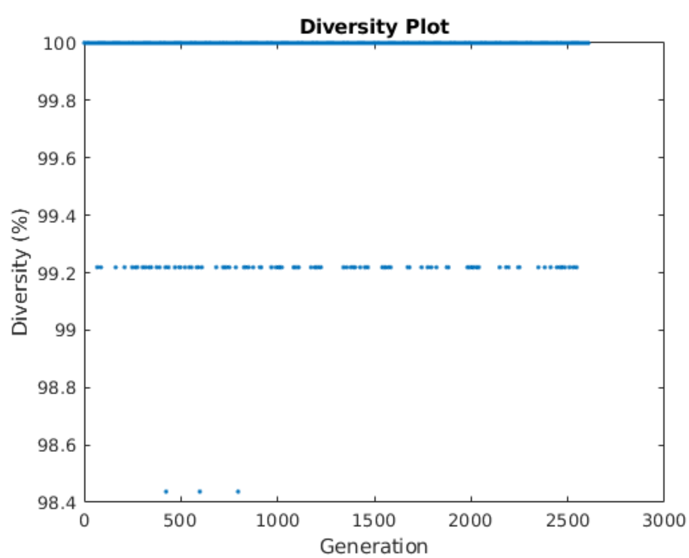

Figure 5: Diversity Chart

5 Appendix

```
#include "main.h"


// OPENGL Variables
int th = 0;             //  Azimuth of view angle
int ph = 0;            //  Elevation of view angle
int axes = 1;          //  Display axes
int light = 0;
double asp = 1;      // aspect ratio
int fov = 40;          //  Field of view (for perspective)
double dim = 30.0;  // size of the workd
double skyBoxScale = 1.0;
double cx=0;             //  Camera Location
double cy=5;
double cz=4;
double view=1000;
double viewlr=90;
int mode = 1;

int generationNumber = 1;
int robotNumber = 128;
int simulationTime = 400;

float emission  = 60;  // Emission intensity (%)
float ambient   = 60;  // Ambient intensity (%)
float diffuse   = 60;  // Diffuse intensity (%)
float specular  = 60;  // Specular intensity (%)
float shininess = 64;  // Shininess (power of two)
float shiny   =   1;    // Shininess (value)
float white[] = {1,1,1,1};
float black[] = {0,0,0,1};

unsigned int grassTexture;
unsigned int slimeTexture;
unsigned int skyBoxTexture[10]; // Texture for Sky Box

// Physics Simluator Variables
double mass = 0.5;
double length = 1;
double gravity = 9.8;
double T = 0;

double timeStep = 0.001;
double restoreConstant = 10000;
double springConstant = 5000;
double dampingConstant = 0.99;
double frictionCoefficient = 0.7;

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

GLfloat worldRotation[16] = {1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1};

std::ofstream bestGene;
std::ofstream popDis;

// calcualte norm for vector
double norm( double x[], std::size_t sz )
{
    return std::sqrt( std::inner_product( x, x+sz, x, 0.0 ) ) ;
}

std::vector<int> sort_indexes(const std::vector<double> &v) {

```

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

class ROBOT
{
private:
    std::vector<GENE> gene;
public:
    double initialLocation[3] = {0,0,0};
    std::vector<MASS> robotMasses;
    std::vector<SPRING> robotSprings;

    ROBOT(double initialX, double initialY, double initialZ, std::vector<GENE> legGene)
    {
        // default constructor
        initialLocation[0] = initialX; initialLocation[1] = initialY; initialLocation[2] =
            initialZ;
        gene = legGene;
        generateRobotMasses(initialX, initialY, initialZ);
        generateRobotSprings();
    }

    void generateRobotMasses(double initialX, double initialY, double initialZ)
    {
        robotMasses.push_back({mass, {initialX, initialY, initialZ+0.5*length}, {0, 0, 0}, {0, 0,
             0}});

        robotMasses.push_back({mass, {initialX-0.5*length, initialY+0.5*length, initialZ}, {0, 0,
             0}, {0, 0, 0}});
        robotMasses.push_back({mass, {initialX-0.5*length, initialY-0.5*length, initialZ}, {0, 0,
             0}, {0, 0, 0}});
        robotMasses.push_back({mass, {initialX+0.5*length, initialY-0.5*length, initialZ}, {0, 0,
             0}, {0, 0, 0}});
        robotMasses.push_back({mass, {initialX+0.5*length, initialY+0.5*length, initialZ}, {0, 0,
             0}, {0, 0, 0}});

        robotMasses.push_back({mass, {initialX-0.0*length, initialY+1.5*length, initialZ}, {0, 0,
             0}, {0, 0, 0}});
        robotMasses.push_back({mass, {initialX-1.5*length, initialY-0.0*length, initialZ}, {0, 0,
             0}, {0, 0, 0}});
        robotMasses.push_back({mass, {initialX+0.0*length, initialY-1.5*length, initialZ}, {0, 0,
             0}, {0, 0, 0}});
        robotMasses.push_back({mass, {initialX+1.5*length, initialY+0.0*length, initialZ}, {0, 0,
             0}, {0, 0, 0}});
    }

    void generateRobotSprings()
    {
        for (int i = 0; i < robotMasses.size() - 1; i++){
            for (int j = 1; j < robotMasses.size(); j++){
                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]};
                if (norm(positionDiff,3) < 1.6*length){
                    robotSprings.push_back({springConstant,norm(positionDiff,3),i,j});
                }
            }
        }

        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]};
        robotSprings.push_back({1000,norm(positionDiff,3),0,5});
        robotSprings.push_back({1000,norm(positionDiff,3),0,6});
```

```
    robotSprings.push_back({1000,norm(positionDiff,3),0,7});
    robotSprings.push_back({1000,norm(positionDiff,3),0,8});

}

void robotDraw()
{
    glColor3f(1, 0, 0);

    GLUquadric *quad;
    quad = gluNewQuadric();
    for (int i = 0; i < (int)robotMasses.size(); i++) {
        glPushMatrix();
        glMultMatrixf(worldRotation);
        glTranslated(robotMasses[i].p[0], robotMasses[i].p[1], robotMasses[i].p[2]);
        gluSphere(quad, length / 20, 10, 10);
        glPopMatrix();
    }

    for (int i = 0; i < (int)robotSprings.size(); i++) {
        glPushMatrix();
        glMultMatrixf(worldRotation);
        glBegin(GL_LINES);
        glVertex3f(GLfloat(robotMasses[robotSprings[i].m1].p[0]), GLfloat(robotMasses[
            robotSprings[i].m1].p[1]), GLfloat(robotMasses[robotSprings[i].m1].p[2]));
        glVertex3f(GLfloat(robotMasses[robotSprings[i].m2].p[0]), GLfloat(robotMasses[
            robotSprings[i].m2].p[1]), GLfloat(robotMasses[robotSprings[i].m2].p[2]));
        glEnd();
        glPopMatrix();
    }

    // draw planes
    glPushMatrix();
    glMultMatrixf(worldRotation);
    glBegin(GL_QUADS);
    glColor3f(0.2, 0.1, 0.3);
    glVertex3f(GLfloat(robotMasses[1].p[0]), GLfloat(robotMasses[1].p[1]), GLfloat(
        robotMasses[1].p[2]));
    glVertex3f(GLfloat(robotMasses[2].p[0]), GLfloat(robotMasses[2].p[1]), GLfloat(
        robotMasses[2].p[2]));
    glVertex3f(GLfloat(robotMasses[3].p[0]), GLfloat(robotMasses[3].p[1]), GLfloat(
        robotMasses[3].p[2]));
    glVertex3f(GLfloat(robotMasses[4].p[0]), GLfloat(robotMasses[4].p[1]), GLfloat(
        robotMasses[4].p[2]));
    glEnd();
    glBegin(GL_TRIANGLES);
    glColor3f(0.2, 0.1, 0.3);
    glVertex3f(GLfloat(robotMasses[0].p[0]), GLfloat(robotMasses[0].p[1]), GLfloat(
        robotMasses[0].p[2]));
    glVertex3f(GLfloat(robotMasses[1].p[0]), GLfloat(robotMasses[1].p[1]), GLfloat(
        robotMasses[1].p[2]));
    glVertex3f(GLfloat(robotMasses[2].p[0]), GLfloat(robotMasses[2].p[1]), GLfloat(
        robotMasses[2].p[2]));
    glEnd();
    glBegin(GL_TRIANGLES);
    glColor3f(0.2, 0.1, 0.3);
    glVertex3f(GLfloat(robotMasses[0].p[0]), GLfloat(robotMasses[0].p[1]), GLfloat(
        robotMasses[0].p[2]));
    glVertex3f(GLfloat(robotMasses[2].p[0]), GLfloat(robotMasses[2].p[1]), GLfloat(
        robotMasses[2].p[2]));
    glVertex3f(GLfloat(robotMasses[3].p[0]), GLfloat(robotMasses[3].p[1]), GLfloat(
        robotMasses[3].p[2]));
    glEnd();
    glBegin(GL_TRIANGLES);
    glColor3f(0.2, 0.1, 0.3);
    glVertex3f(GLfloat(robotMasses[0].p[0]), GLfloat(robotMasses[0].p[1]), GLfloat(
        robotMasses[0].p[2]));
```

```
    glVertex3f(GLfloat(robotMasses[3].p[0]), GLfloat(robotMasses[3].p[1]), GLfloat(
        robotMasses[3].p[2]));
    glVertex3f(GLfloat(robotMasses[4].p[0]), GLfloat(robotMasses[4].p[1]), GLfloat(
        robotMasses[4].p[2]));
    glEnd();
    glBegin(GL_TRIANGLES);
    glColor3f(0.2, 0.1, 0.3);
    glVertex3f(GLfloat(robotMasses[0].p[0]), GLfloat(robotMasses[0].p[1]), GLfloat(
        robotMasses[0].p[2]));
    glVertex3f(GLfloat(robotMasses[4].p[0]), GLfloat(robotMasses[4].p[1]), GLfloat(
        robotMasses[4].p[2]));
    glVertex3f(GLfloat(robotMasses[1].p[0]), GLfloat(robotMasses[1].p[1]), GLfloat(
        robotMasses[1].p[2]));
    glEnd();
    glBegin(GL_TRIANGLES);
    glColor3f(0.2, 0.1, 0.3);
    glVertex3f(GLfloat(robotMasses[1].p[0]), GLfloat(robotMasses[1].p[1]), GLfloat(
        robotMasses[1].p[2]));
    glVertex3f(GLfloat(robotMasses[2].p[0]), GLfloat(robotMasses[2].p[1]), GLfloat(
        robotMasses[2].p[2]));
    glVertex3f(GLfloat(robotMasses[6].p[0]), GLfloat(robotMasses[6].p[1]), GLfloat(
        robotMasses[6].p[2]));
    glEnd();
    glBegin(GL_TRIANGLES);
    glColor3f(0.2, 0.1, 0.3);
    glVertex3f(GLfloat(robotMasses[2].p[0]), GLfloat(robotMasses[2].p[1]), GLfloat(
        robotMasses[2].p[2]));
    glVertex3f(GLfloat(robotMasses[3].p[0]), GLfloat(robotMasses[3].p[1]), GLfloat(
        robotMasses[3].p[2]));
    glVertex3f(GLfloat(robotMasses[7].p[0]), GLfloat(robotMasses[7].p[1]), GLfloat(
        robotMasses[7].p[2]));
    glEnd();
    glBegin(GL_TRIANGLES);
    glColor3f(0.2, 0.1, 0.3);
    glVertex3f(GLfloat(robotMasses[3].p[0]), GLfloat(robotMasses[3].p[1]), GLfloat(
        robotMasses[3].p[2]));
    glVertex3f(GLfloat(robotMasses[4].p[0]), GLfloat(robotMasses[4].p[1]), GLfloat(
        robotMasses[4].p[2]));
    glVertex3f(GLfloat(robotMasses[8].p[0]), GLfloat(robotMasses[8].p[1]), GLfloat(
        robotMasses[8].p[2]));
    glEnd();
    glBegin(GL_TRIANGLES);
    glColor3f(0.2, 0.1, 0.3);
    glVertex3f(GLfloat(robotMasses[1].p[0]), GLfloat(robotMasses[1].p[1]), GLfloat(
        robotMasses[1].p[2]));
    glVertex3f(GLfloat(robotMasses[4].p[0]), GLfloat(robotMasses[4].p[1]), GLfloat(
        robotMasses[4].p[2]));
    glVertex3f(GLfloat(robotMasses[5].p[0]), GLfloat(robotMasses[5].p[1]), GLfloat(
        robotMasses[5].p[2]));
    glEnd();
    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);
```

```
        glVertex3f(GLfloat(x), GLfloat(y), GLfloat(z));
        glVertex3f(GLfloat(initialLocation[0]), GLfloat(initialLocation[1]), GLfloat(initialLocation[2]));
        glEnd();
        glPopMatrix();
        Frames++;
        GLint t = glutGet(GLUT_ELAPSED_TIME);
        if (t - T0 >= 1000) {
            GLfloat seconds = (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;
        }
    }

    void robotUpdate()
    {
        // initialize the force vector with value 0
        std::vector<std::vector<double>> robotForces((int)robotMasses.size(),std::vector<double>(3));
        if (T > 1.0){
            robotSprings[robotSprings.size()-4].L_0 = 1.58114 + gene[0].A / (2 * M_PI) * length * cos(gene[0].B / 4 * T + gene[0].C);
            robotSprings[robotSprings.size()-3].L_0 = 1.58114 + gene[1].A / (2 * M_PI) * length * cos(gene[1].B / 4 * T + gene[1].C);
            robotSprings[robotSprings.size()-2].L_0 = 1.58114 + gene[2].A / (2 * M_PI) * length * cos(gene[2].B / 4 * T + gene[2].C);
            robotSprings[robotSprings.size()-1].L_0 = 1.58114 + gene[3].A / (2 * M_PI) * length * cos(gene[3].B / 4 * T + gene[3].C);
        }

        // loop through all springs to calculate spring forces
        //#pragma omp parallel for
        for (int i = 0; i < (int)robotSprings.size(); i++) {
            MASS mass1 = robotMasses[robotSprings[i].m1];
            MASS mass2 = robotMasses[robotSprings[i].m2];
            double positionDiff[3] = {mass2.p[0] - mass1.p[0], mass2.p[1] - mass1.p[1], mass2.p[2] - mass1.p[2]};
            double L = norm(positionDiff, 3);
            double force = robotSprings[i].k * fabs(robotSprings[i].L_0 - L);
            double direction[3] = {positionDiff[0] / L, positionDiff[1] / L, positionDiff[2] / L};
            // contraction case
            if (L > robotSprings[i].L_0) {
                robotForces[robotSprings[i].m1][0] = robotForces[robotSprings[i].m1][0] + direction[0] * force;
                robotForces[robotSprings[i].m1][1] = robotForces[robotSprings[i].m1][1] + direction[1] * force;
                robotForces[robotSprings[i].m1][2] = robotForces[robotSprings[i].m1][2] + direction[2] * force;
                robotForces[robotSprings[i].m2][0] = robotForces[robotSprings[i].m2][0] - direction[0] * force;
                robotForces[robotSprings[i].m2][1] = robotForces[robotSprings[i].m2][1] - direction[1] * force;
                robotForces[robotSprings[i].m2][2] = robotForces[robotSprings[i].m2][2] - direction[2] * force;
            }
                // expansion case
            else if (L < robotSprings[i].L_0) {
                robotForces[robotSprings[i].m1][0] = robotForces[robotSprings[i].m1][0] - direction[0] * force;
                robotForces[robotSprings[i].m1][1] = robotForces[robotSprings[i].m1][1] - direction[1] * force;
                robotForces[robotSprings[i].m1][2] = robotForces[robotSprings[i].m1][2] - direction[2] * force;
                robotForces[robotSprings[i].m2][0] = robotForces[robotSprings[i].m2][0] + direction[0] * force;
```

```
                robotForces[robotSprings[i].m2][1] = robotForces[robotSprings[i].m2][1] +
                    direction[1] * force;
                robotForces[robotSprings[i].m2][2] = robotForces[robotSprings[i].m2][2] +
                    direction[2] * force;
            }
        }

        //#pragma omp parallel for
        for (int i = 0; i < (int)robotMasses.size(); i++) {
            // add gravity
            robotForces[i][2] = robotForces[i][2] - robotMasses[i].m * gravity;
            // if the mass is below ground, add restroration force and calculate friction
            if (robotMasses[i].p[2] <= 0) {
                robotForces[i][2] = robotForces[i][2] + restoreConstant * fabs(robotMasses[i].p
                    [2]);
                // calculate horizontal force and vertical force
                double F_h = sqrt(pow(robotForces[i][0], 2) + pow(robotForces[i][1], 2));
                double F_v = robotForces[i][2];
                if (F_h < F_v * frictionCoefficient) {
                    robotForces[i][0] = 0;
                    robotForces[i][1] = 0;
                    robotMasses[i].v[0] = 0;
                    robotMasses[i].v[1] = 0;
                }
            }
            // update acceleration
            robotMasses[i].a[0] = robotForces[i][0] / robotMasses[i].m;
            robotMasses[i].a[1] = robotForces[i][1] / robotMasses[i].m;
            robotMasses[i].a[2] = robotForces[i][2] / robotMasses[i].m;
            // update velocity
            robotMasses[i].v[0] = dampingConstant * (robotMasses[i].v[0] + robotMasses[i].a[0] *
                timeStep);
            robotMasses[i].v[1] = dampingConstant * (robotMasses[i].v[1] + robotMasses[i].a[1] *
                timeStep);
            robotMasses[i].v[2] = dampingConstant * (robotMasses[i].v[2] + robotMasses[i].a[2] *
                timeStep);
            // update position
            robotMasses[i].p[0] = robotMasses[i].p[0] + robotMasses[i].v[0] * timeStep;
            robotMasses[i].p[1] = robotMasses[i].p[1] + robotMasses[i].v[1] * timeStep;
            robotMasses[i].p[2] = robotMasses[i].p[2] + robotMasses[i].v[2] * timeStep;
        }
        // update time
        T = T + timeStep;
    }


};

class Simulation{
private:
    int populationSize;
    std::vector<double> populationDistance;
    std::vector<std::vector<GENE>> populationGene;
    std::vector<std::vector<GENE>> newPopulationGene;
    std::vector<ROBOT> robots;
public:
    double averageDistance;
    double maxDistance;

    Simulation(int popSize)
    {
        populationSize = popSize;
        generateGenes();
        //generateBestGene();
        generateRobots();
        popDis.open ("populationDistance.txt"); popDis.close();
        bestGene.open("bestGene.txt"); bestGene.close();
        popDis.open ("populationDistance.txt", std::ios_base::app);
```

```
        bestGene.open("bestGene.txt", std::ios_base::app);
    }

    void startSim(double time){
        if (T < time){
            simUpdate();
            simDraw();
        }
        else {
            calculatePopulationDistance();
            selection();
            crossOver();
            populationGene.clear(); populationGene.shrink_to_fit();
            populationGene = newPopulationGene;
            generationNumber++;
            robots.clear(); robots.shrink_to_fit();
            generateRobots();
            T = 0;
        }
    }

    void selection(){
        std::vector<int> index = sort_indexes(populationDistance);
        newPopulationGene.clear();
        newPopulationGene.shrink_to_fit();
        for (int i = 0; i < index.size()/2; i++) {
            //std::cout << index[i] << std::endl;
            newPopulationGene.push_back(populationGene[index[i]]);
        }
        for (int i = 0; i < newPopulationGene[0].size(); i++){
            bestGene << newPopulationGene[0][i].A << " " << newPopulationGene[0][i].B << " " <<
                newPopulationGene[0][i].C << " ";
        }
        bestGene << "\n";
    }

    void crossOver(){
        for (int n = 0; n < populationGene.size() / 2; n++){
            int parentIndex1 = rand() % static_cast<int>(newPopulationGene.size());
            int parentIndex2 = rand() % static_cast<int>(newPopulationGene.size());
            std::vector<double> parent1, parent2;
            for (int i = 0; i < newPopulationGene[parentIndex1].size(); i++){
                parent1.push_back(newPopulationGene[parentIndex1][i].A);
                parent1.push_back(newPopulationGene[parentIndex1][i].B);
                parent1.push_back(newPopulationGene[parentIndex1][i].C);
                parent2.push_back(newPopulationGene[parentIndex2][i].A);
                parent2.push_back(newPopulationGene[parentIndex2][i].B);
                parent2.push_back(newPopulationGene[parentIndex2][i].C);
            }
            int crossOverPoint1 = rand() % static_cast<int>(parent1.size());
            int crossOverPoint2 = rand() % static_cast<int>(parent1.size());
            if (crossOverPoint2 < crossOverPoint1){
                int temp = crossOverPoint1;
                crossOverPoint1 = crossOverPoint2;
                crossOverPoint2 = temp;
            }
            std::vector<double> offSpring1, offSpring2;
            for (int i = 0; i < crossOverPoint1; i++){
                offSpring1.push_back(parent1[i]);
                offSpring2.push_back(parent2[i]);
            }
            for (int i = crossOverPoint1; i < crossOverPoint2; i++){
                offSpring1.push_back(parent2[i]);
                offSpring2.push_back(parent1[i]);
            }
            for (int i = crossOverPoint2; i < parent1.size(); i++){
                offSpring1.push_back(parent1[i]);
                offSpring2.push_back(parent2[i]);
```

```
            }
            offSpring1 = mutation(offSpring1);
            offSpring2 = mutation(offSpring2);
            std::vector<GENE> offSpringGene1, offSpringGene2;
            GENE temp1, temp2;
            for (int i = 0; i < offSpring1.size(); i = i + 3){
                temp1.A = offSpring1[i];
                temp1.B = offSpring1[i+1];
                temp1.C = offSpring1[i+2];
                temp2.A = offSpring2[i];
                temp2.B = offSpring2[i+1];
                temp2.C = offSpring2[i+2];
                offSpringGene1.push_back(temp1);
                offSpringGene2.push_back(temp2);
            }
            newPopulationGene.push_back(offSpringGene1);
            newPopulationGene.push_back(offSpringGene2);
        }
    }

    std::vector<double> mutation(std::vector<double> offSpring){
        for (int i = 0; i < offSpring.size(); i++){
            double mutationProbability = static_cast <float> (rand()) / (static_cast <float> (
                RAND_MAX/1.0));
            if (mutationProbability > 0.9){
                offSpring[i] = -2*M_PI + static_cast <float> (rand()) / (static_cast <float> (
                    RAND_MAX/(4*M_PI)));
            }
        }
        return offSpring;

    }

    void generateBestGene(){
        for (int i = 0; i < populationSize; i++) {
            std::vector<GENE> tempVec1;
            GENE temp1{6.26142, -6.27729, 1.23343};
            tempVec1.push_back(temp1);
            GENE temp2{-0.0439628, -0.281218, 1.16934};
            tempVec1.push_back(temp2);
            GENE temp3{-0.00115234, -2.94204, 6.13413};
            tempVec1.push_back(temp3);
            GENE temp4{-5.13257, -0.0672038, 1.83822};
            tempVec1.push_back(temp4);
            populationGene.push_back(tempVec1);
        }
    }

    void generateGenes(){
        srand(time(0));
        for (int i = 0; i < populationSize; i++){
            std::vector<GENE> tempVec;
            for (int j = 0; j < 4; j++){
                double A = -2*M_PI + static_cast <float> (rand()) / (static_cast <float> (
                    RAND_MAX/(4*M_PI))); // actual assign -1 - 1
                double B = -2*M_PI + static_cast <float> (rand()) / (static_cast <float> (
                    RAND_MAX/(4*M_PI))); // actual assign -pi/2 - pi/2
                double C = -2*M_PI + static_cast <float> (rand()) / (static_cast <float> (
                    RAND_MAX/(4*M_PI))); // acutal assign -2pi - 2pi
                GENE temp{A,B,C};
                tempVec.push_back(temp);
            }
            populationGene.push_back(tempVec);
        }
    }

    void generateRobots(){
        for (int i = 0; i < populationSize; i++){
```

```
            double X = -20 + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX/40));
            double Y = -20 + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX/40));
            //robots.push_back(ROBOT(0.0, 3.0*(i-populationSize/2), 0.0, populationGene[i]));
            robots.push_back(ROBOT(X, Y, 0.0, populationGene[i]));
        }
    }

    void simUpdate(){
        #pragma omp parallel for num_threads(32)
        for (int i = 0; i < populationSize; i++){
            robots[i].robotUpdate();
        }
    }

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

    void calculatePopulationDistance(){
        populationDistance.clear();
        populationDistance.shrink_to_fit();
        for (int i = 0; i < populationSize; i++){
            double x = 0; double y = 0;
            for (int j = 1; j < 5; j++){
                x = x + robots[i].robotMasses[j].p[0];
                y = y + robots[i].robotMasses[j].p[1];
            }
            x = x / 4;
            y = y / 4;
            double distance[2] = {fabs(x - robots[i].initialLocation[0]), fabs(y - robots[i].
                initialLocation[1])};
            double distanceNorm = norm(distance, 2);
            populationDistance.push_back(distanceNorm);
        }
        averageDistance = 0;
        maxDistance = 0;
        std::cout << "###################" << std::endl;
        for (int i = 0; i < populationSize; i++){
            averageDistance = averageDistance + populationDistance[i];
            maxDistance = std::max(maxDistance, populationDistance[i]);
            //std::cout << populationDistance[i] << std::endl;
            popDis << populationDistance[i] << " ";
        }
        popDis << "\n";
        averageDistance = averageDistance/populationSize;
        std::cout << "Maximum Distance: " << maxDistance << std::endl;
        std::cout << "Average Distance: " << averageDistance << std::endl;
    }
};

Simulation sim1(robotNumber);


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

```

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

            glPushMatrix();
            glTranslatef(i*2, 0, j*2);
            glBegin(GL_QUADS);
            //
            glNormal3f(0, 1, 0);
            glColor3f(0.0, 0.0, 0.0);
            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(0.0, 0.0, 0.0);
            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();
        }
    }

}

static void ball(double x,double y,double z,double r)
{
    //  Save transformation
    glPushMatrix();
    //  Offset, scale and rotate
    glTranslated(x,y,z);
    glScaled(r,r,r);
    //  White ball
    glColor3f(1,1,1);
    glutSolidSphere(1.0,16,16);
    //  Undo transofrmations
    glPopMatrix();
}

void display()
{
    const double len=2.0;  //  Length of axes
    //  Erase the window and the depth buffer
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    //  Enable Z-buffering in OpenGL
```

```
glEnable(GL_DEPTH_TEST);
//  Undo previous transformations
glLoadIdentity();
//  Eye position
if (mode == 1) {
    //  Eye position
    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);
}
if (mode == 2) {
    gluLookAt(cx,cy,cz,cx+view*Cos(viewlr),cy,cz+view*Sin(viewlr),0,1,0);
}

if (light)
{
    //  Translate intensity to color vectors
    float Ambient[]   = {(float)0.01*ambient,(float)0.01*ambient,(float)0.01*ambient ,2.0};
    float Diffuse[]   = {(float)0.01*diffuse,(float)0.01*diffuse,(float)0.01*diffuse ,2.0};
    float Specular[]  = {(float)0.01*specular,(float)0.01*specular,(float)0.01*specular,2.0};
    //  Light direction
    //float Position[]  = {0,10,0};
    //float Position[]  = {5*Cos(zh),ylight,5*Sin(zh),1};
    float Position[]  = {95,90,60,1};
    //  Draw light position as ball (still no lighting here)
    glColor3f(16,16,16);
    ball(Position[0],Position[1],Position[2], 5);
    //  OpenGL should normalize normal vectors
    glEnable(GL_NORMALIZE);
    //  Enable lighting
    glEnable(GL_LIGHTING);
    //  glColor sets ambient and diffuse color materials
    glColorMaterial(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE);
    glEnable(GL_COLOR_MATERIAL);
    //  Enable light 0
    glEnable(GL_LIGHT0);
    //  Set ambient, diffuse, specular components and position of light 0
    glLightfv(GL_LIGHT0,GL_AMBIENT ,Ambient);
    glLightfv(GL_LIGHT0,GL_DIFFUSE ,Diffuse);
    glLightfv(GL_LIGHT0,GL_SPECULAR,Specular);
    glLightfv(GL_LIGHT0,GL_POSITION,Position);
}
else {
    glDisable(GL_LIGHTING);
}

drawGrid();
glColor3f(1,1,1);
glWindowPos2i(0,0);
Print("Generation: %d", generationNumber);
glWindowPos2i(850, 0);
Print("Max: %4.2f", sim1.maxDistance);
glWindowPos2i(850, 50);
Print("Average %4.2f", sim1.averageDistance);
sim1.startSim(simulationTime);

glRasterPos3d(0.0,2,0.0);
if (fps>0) Print("FPS %.3f", fps);
//drawGrass();
//skyBox(skyBoxScale);
//  Draw axes
glColor3f(1,1,1);
if (axes)
{
    glBegin(GL_LINES);
    glVertex3d(0.0,0.0,0.0);
    glVertex3d(len,0.0,0.0);
```

```c
        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();
}

/*
 *  GLUT calls this routine when an arrow key is pressed
 */
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();
}

/*
 *  Set projection
 */
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();
}

/*
 *  GLUT calls this routine when a key is pressed
 */
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;
        int x;
        //  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;
    }
    else if (ch == 'w' || ch == 'W'){
        cx += 0.3*Cos(viewlr);
        cz += 0.3*Sin(viewlr);
    }
    else if (ch == 'a' || ch == 'A'){
        cx += 0.3*Sin(viewlr);
        cz -= 0.3*Cos(viewlr);
    }
    else if (ch == 's' || ch == 'S'){
        cx -= 0.3*Cos(viewlr);
        cz -= 0.3*Sin(viewlr);
    }
    else if (ch == 'd' || ch == 'D'){
        cx -= 0.3*Sin(viewlr);
        cz += 0.3*Cos(viewlr);
    }
    else if (ch == 'r' || ch == 'R'){
        if (cy+0.3 < skyBoxScale*10){
            cy += 0.3;
        }
    }
    else if (ch == 'f' || ch == 'F'){
        if (cy-0.3 > 0){
            cy -= 0.3;
        }
    }
    else if (ch == 'q' || ch=='Q'){
        viewlr-=15;
    }
    else if (ch == 'e' || ch=='E'){
        viewlr+=15;
    }

    if (ch == '1')
```

```
    {
        mode = 1;
    }
    else if (ch == '2')
    {
        mode = 2;
    }
    //  Keep angles to +/-360 degrees
    th %= 360;
    ph %= 360;
    //  Reproject
    Project(fov,asp,dim);
    //  Tell GLUT it is necessary to redisplay the scene
    glutPostRedisplay();
}

/*
 *  GLUT calls this routine when the window is resized
 */
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);
}

/*
 *  GLUT calls this toutine when there is nothing else to do
 */
void idle()
{
    glutPostRedisplay();
}

int main(int argc,char* argv[]) {
    // Initialize GLUT and process user parameters
    glutInit(&argc, argv);
    glWindowPos2i =  (PFNGLWINDOWPOS2IPROC) glutGetProcAddress("glWindowPos2i");
    // double buffered, true color 600*600
    glutInitWindowSize(1000,800);
    glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
    // create the window
    glutCreateWindow("hz2547");
    //  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);
    //  Pass control to GLUT so it can interact with the user
    glutMainLoop();

    //std::cout << "Hello" << std::endl;
    //vecAdd_wrapper();
    return 0;
}
```