MECS4510 Evolutionary Computation and Design Automation

Project Phase C

Jiaqi Miao

UNI: jm5166

Wanjun Chao

UNI: wc2747

Instructor: Hod Lipson

12/14/2020

Grace hours used: 0

Grace hours remaining: 129+115

Contents

Result:

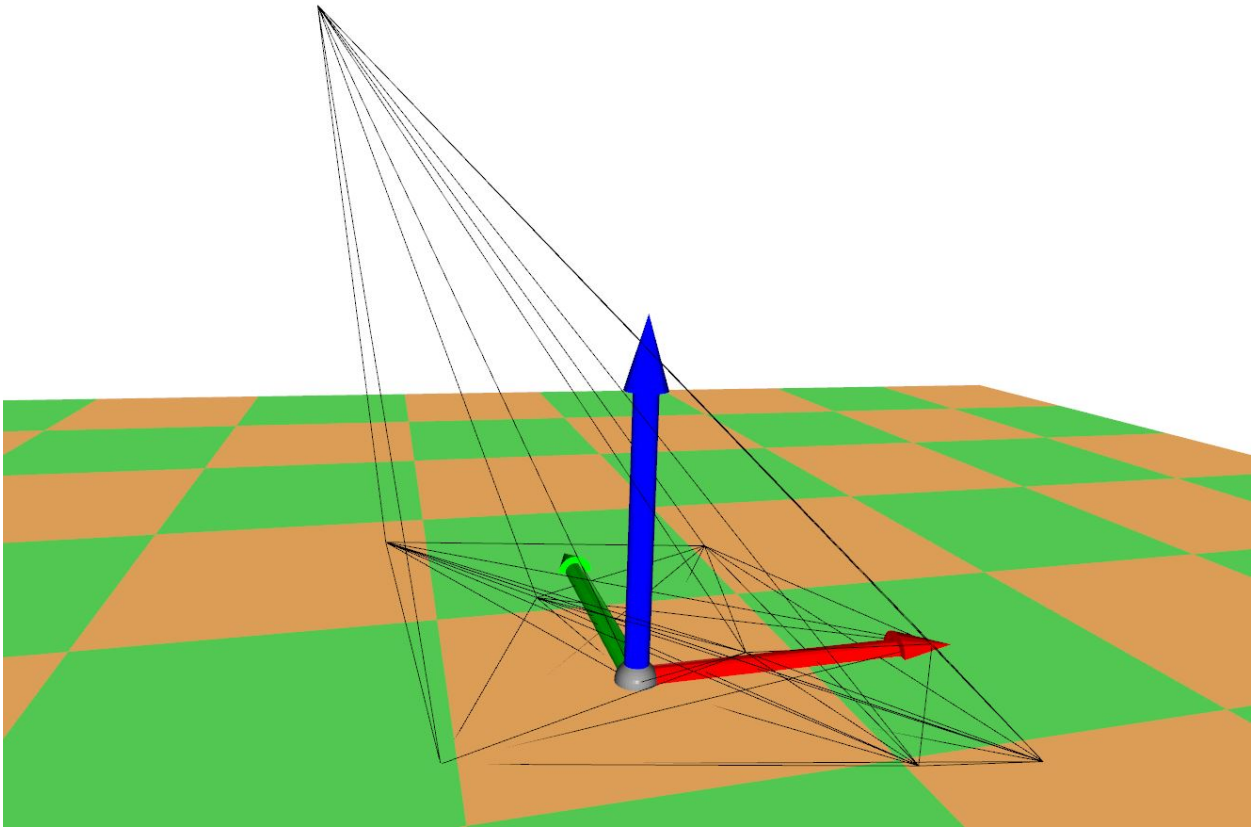


Figure 1 Robot moving

Fastest robot speed = 0.77m/s

3.85 Diameter/s

Fastest Robot

https://youtu.be/OhEQxhTynLo

Fastest robot bounce

https://youtu.be/uJ1q63aLzNs

Methods and Analysis:

For phase C of the project. We use the same physics simulator as the previous phases. The parameters for physics simulator are the same as phase B:

Timesteps dT = 0.0001s

Static coefficient of friction = 0.61(between Aluminum and Steel)

Kinetic coefficient of friction = 0.47

Damping Constant = 0.9999

For phase C's representation encoding, we used direct encoding to maximize the possibility.

Representation for fixed number of Masses & adjustable Spring number	
Spring Index	Randomly selected from 28 spring pairs of a robot with 8 masses
Number of Spring	In a range between 8 springs and 28 springs
b	A double value between 70 and 100. Part of breathing sine function
c	A double value between 0 and 19, part of breathing sine function
Spring constant	An integer value between 5000 and 20000

Representation for non fixed number of masses	
Number of masses	An integer value between 7 and 16
Number of Spring	In a range between 8 springs and 28 springs
b	A double value between 70 and 100. Part of breathing sine function
c	A double value between 0 and 19, part of breathing sine function
Spring constant	An integer value between 5000 and 20000
Mass position relative to the starting mass point	For x and y coordinate: between 0.1 and -0.1 For z coordinate: between 0 and 0.1

The equation we used to make spring breath and robot moving is showed below.

$$L0 = L0x + \frac{\sin(23 * T + c)}{b}$$

L0 is origin length of spring, it will be changed as time pass. L0x is the initial length of the spring before any changed is applied on it. The reason to use L0x here is to make sure the changing range of the spring is at a safety level so the robot will not explode if length change of spring over the limited. As L0 changes each time, the robot will be able to move in the space.

Different from phase B, the shape of the robot is not constant phase C. Two different types of robots are created in this phase of project. The first type keeps the total number of mass and their position but change the number of springs. For the second type, we generate random number of masses and use springs to connect them and form new robots.

To get the fastest robot by genetic algorithm. We set a 10000 generations evolutionary algorithm. Each generation contain 20 population and the top 50 precent will be selected as parent for the next generation. The reason we pick 20 population is because of efficiency. Due to the limitation of our coding skills, 20 precent is the best population size we can work on to reduce the processing time but still get a good result. Each robot will run 1000 cycles to calculate the speed. The speed is based on the horizontal displacement of center of mass between first cycle and the last cycles.

For the crossover part, crossover rate is set to be 50%. We randomly generate a number between 1 and 20, if it is even number, the crossover will happen. Otherwise, a parent set will be randomly selected from top 10 and

passes to the next generation. In this phase of project. The basic idea of crossover is randomly pick a position in the mass/spring index, the element from 0 to selected index will be from father and the element after elected will be from mother. We tried two types of crossover. The difference between two crossover type is the element size of the children. For first types of crossover, the number of springs for fixed mass number and number of mass for non-fixed mass number are decided by the parent with highest amount of spring/mass. This offer a more stable child generation because the good gene of parents has a higher possibility to be passed to next generation. For the second crossover method, the mass/spring number the children robot will be the mean of mass/spring numbers of parents. The first method ensure the best gene will be passed to the next generation but it lost the diversity. The diversity is depended on the number of spring/mass. Robot with same amount of spring/mass will be seen as the same type even the position of the mass is different because they have same amount of mass. After several generation, the diversity will be reduced to 1 because all robots are having the same number of mass/springs. The second crossover method is intended to solve this problem but it just postpones the loss of diversity to a later generation. Therefore, the mutation is needed here.

The mutation rate used is 50%. If a number selected from 1 to 20 is an odd number, then mutation will happen. For mutation, we also tried two method. The first method is simple method which does not change the number of mass/spring, just randomly pick two mass in the robot and change the position of these two mass and do the same thing to two random spring as well. This is not a good way to change the diversity because after all, the total number of spring/mass are not changed. Thus, we implemented the second mutation method. Different from the first one, second mutation method not only change the parameter of a specific mass/spring but do change to the whole robot. The second mutation will change the total number of mass to a new value. If the new size is greater than pervious size, the extra mass and spring will be added, if the new size if smaller than pervious size, the extra mass and spring will be deleted. This new mutation method will keep the diversity level of each generation at a relative high level. However, as the dot plot shown in figure 3 and figure 4. Figure 3 representing the first mutation, because the diversity of first mutation was very low, the range of robot speed in each generation is very narrow, while figure 4 representing the second mutation shows a wider range of robot speed in each generation. Although second mutation method has wider speed range, its average speed is not as fast as first mutation method. I think the mean reason is because second mutation method break the element of children that was passed from parents, thus some good elements that will make robot move faster is destroyed during the change of mass/spring size. The first mutation prevented that big change but shrink the possibility of getting a faster speed. I think the possible reason is because the current population size we have is not big enough and iteration times is also not enough.

Also figure 2 performance plots is showing a very interesting result. Robot with fixed number/position of mass and non-fixed number of springs has a higher average speed than robot with non-fixed number masses. I think the reason is because robot with eight mass has lower over all weight than other robot. From my observation, robot that has a higher speed usually has a jump action at the beginning cycle which gives 8 mass robot an advantage since lighter weight makes it fly further before hitting the ground.

Performance Plot:

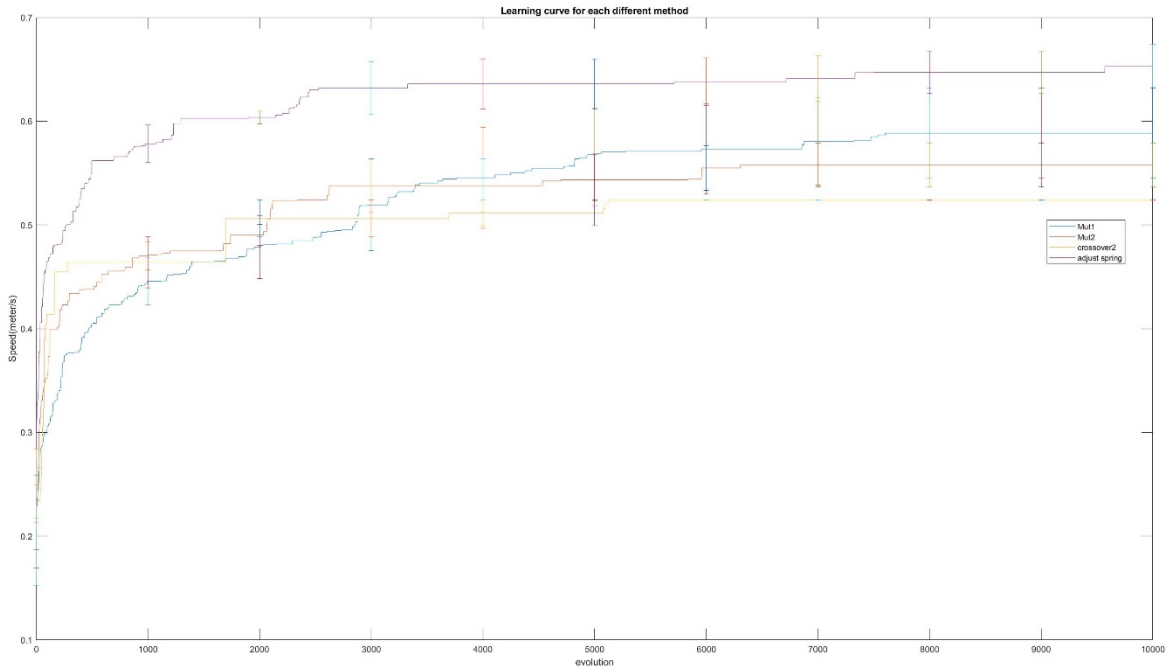


Figure 2 Learning Curve

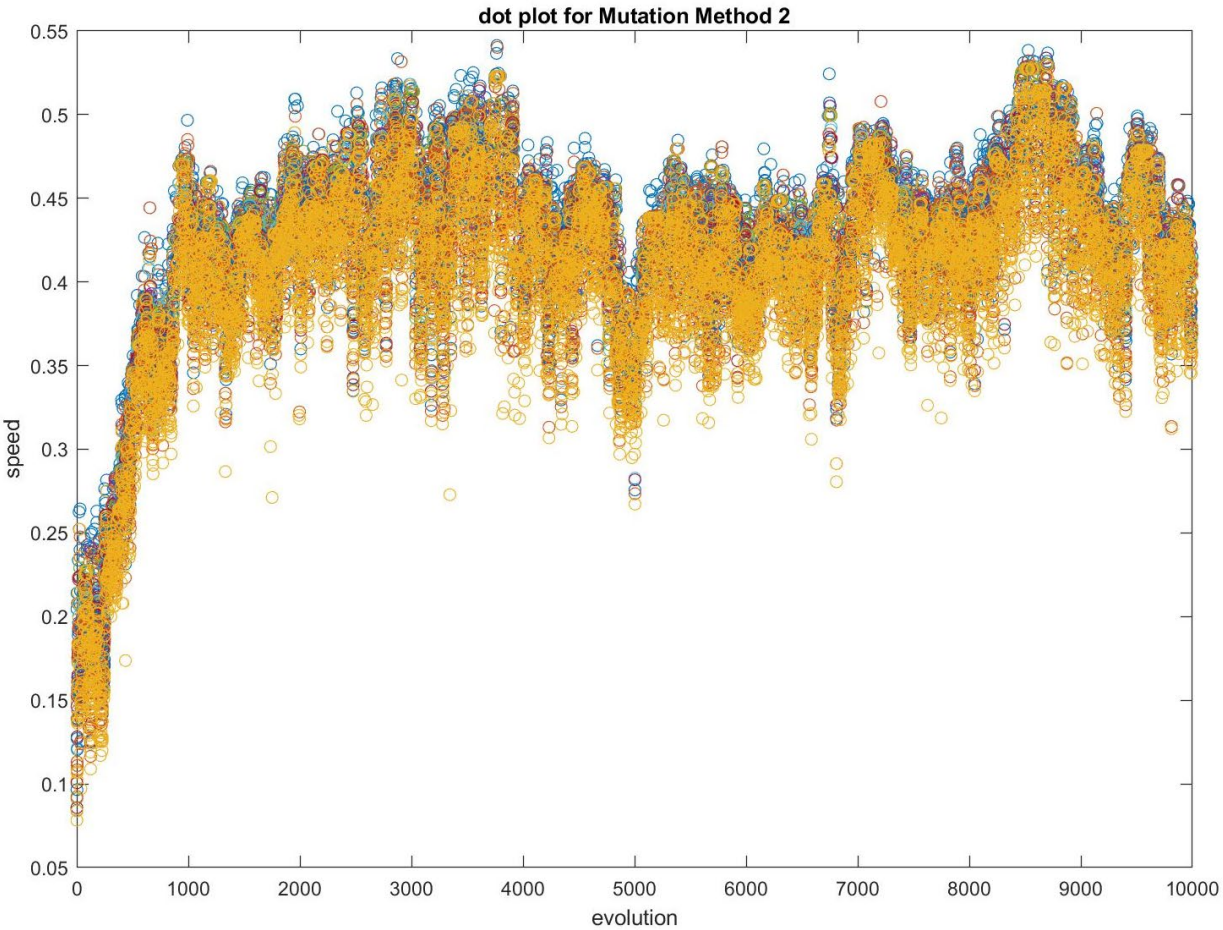


Figure 3 Dot plot Mutation1

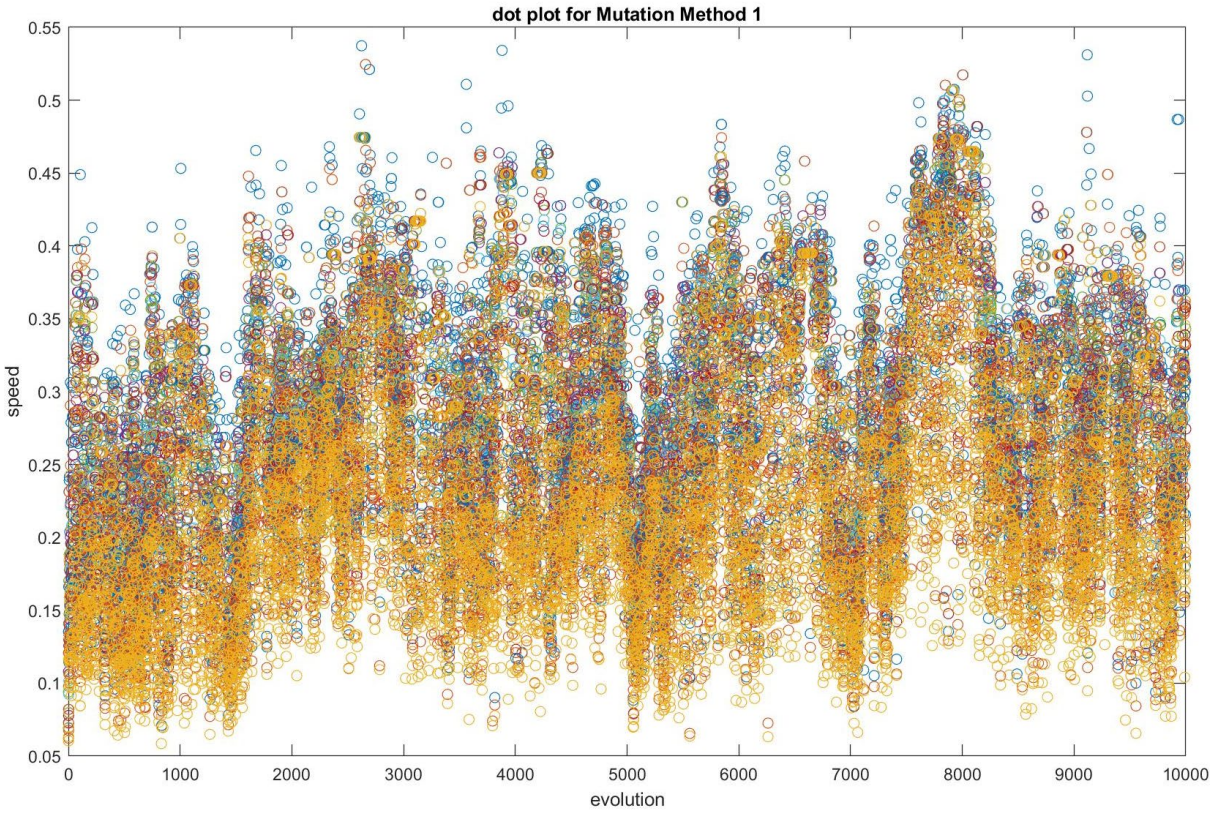


Figure 4 Dot plot Mutation2

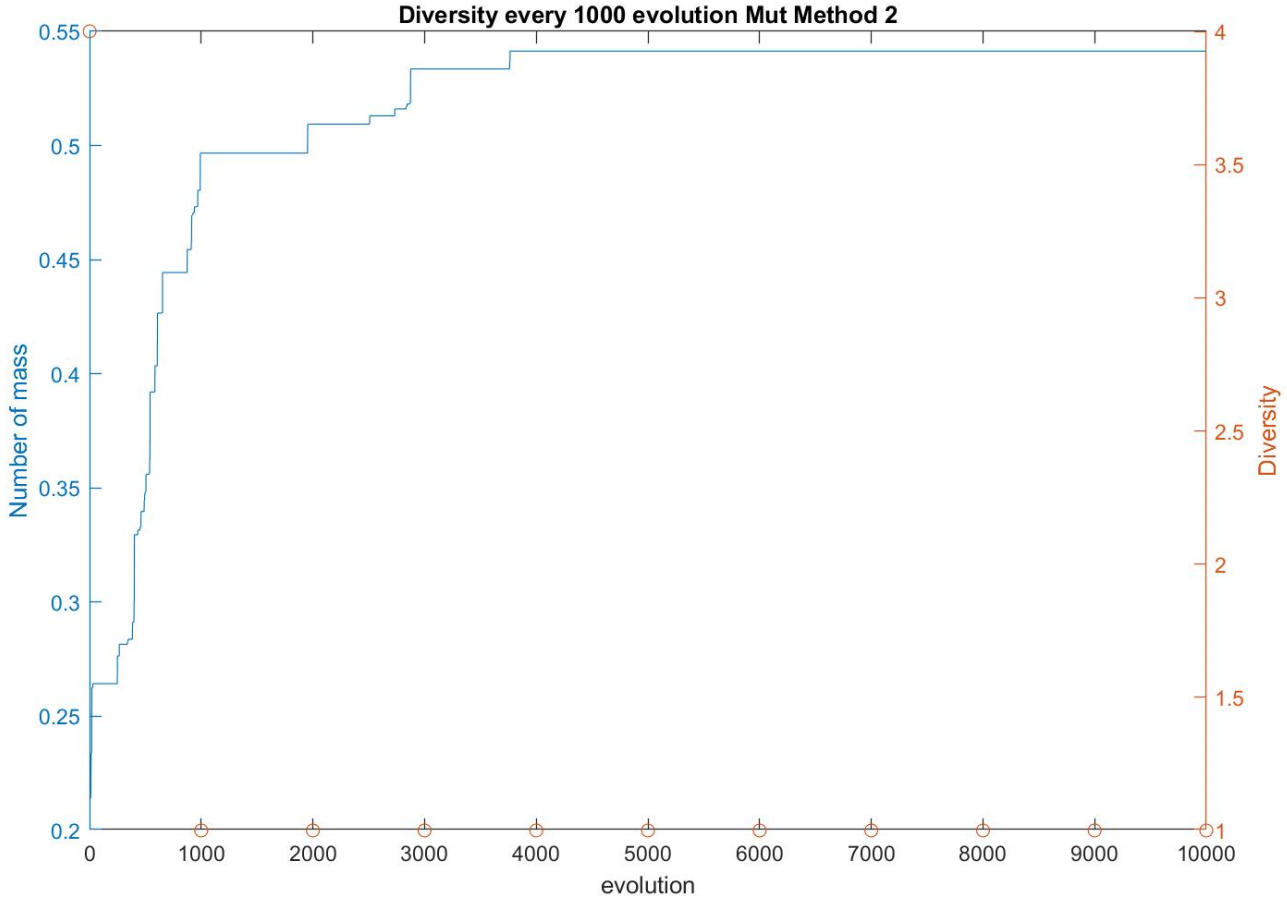


Figure 5 Diversity plot m1

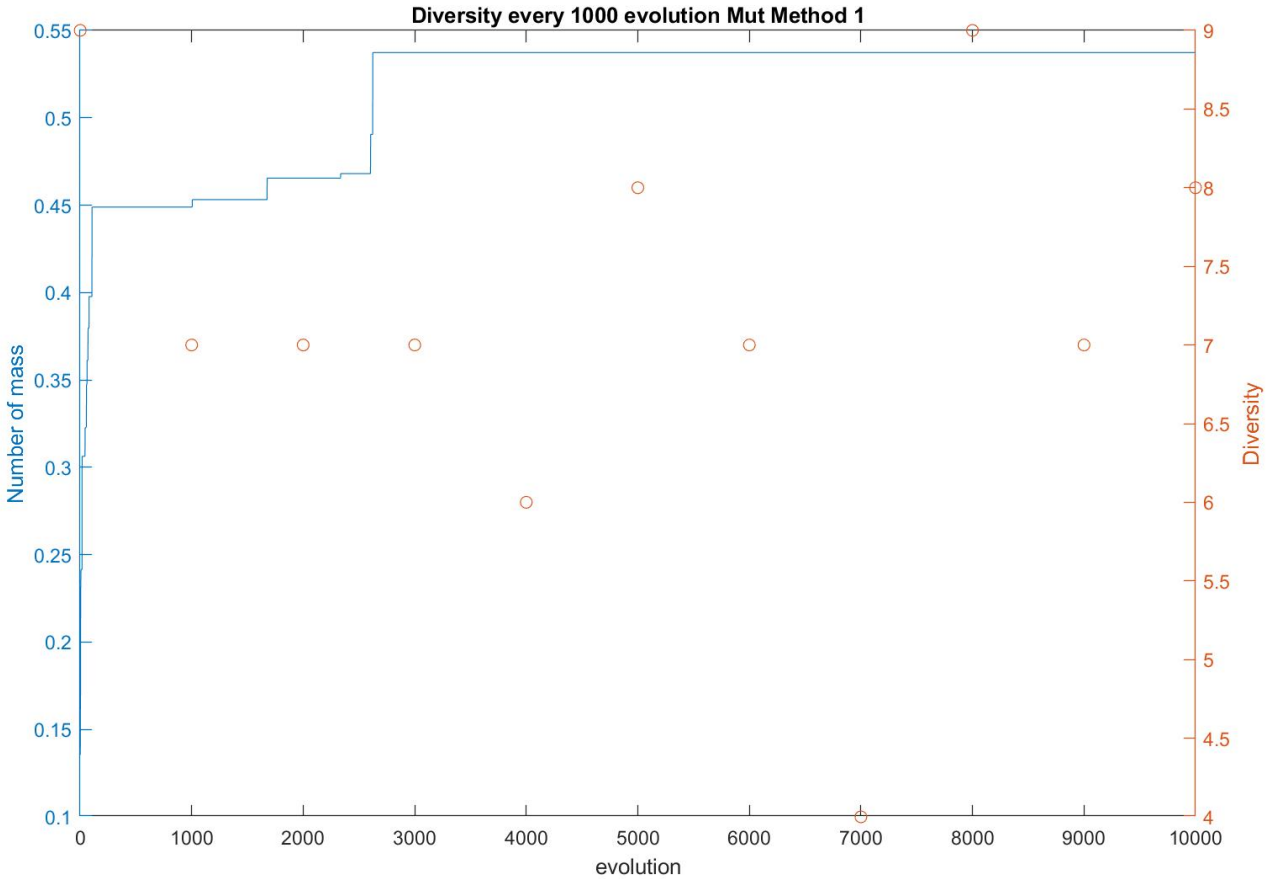
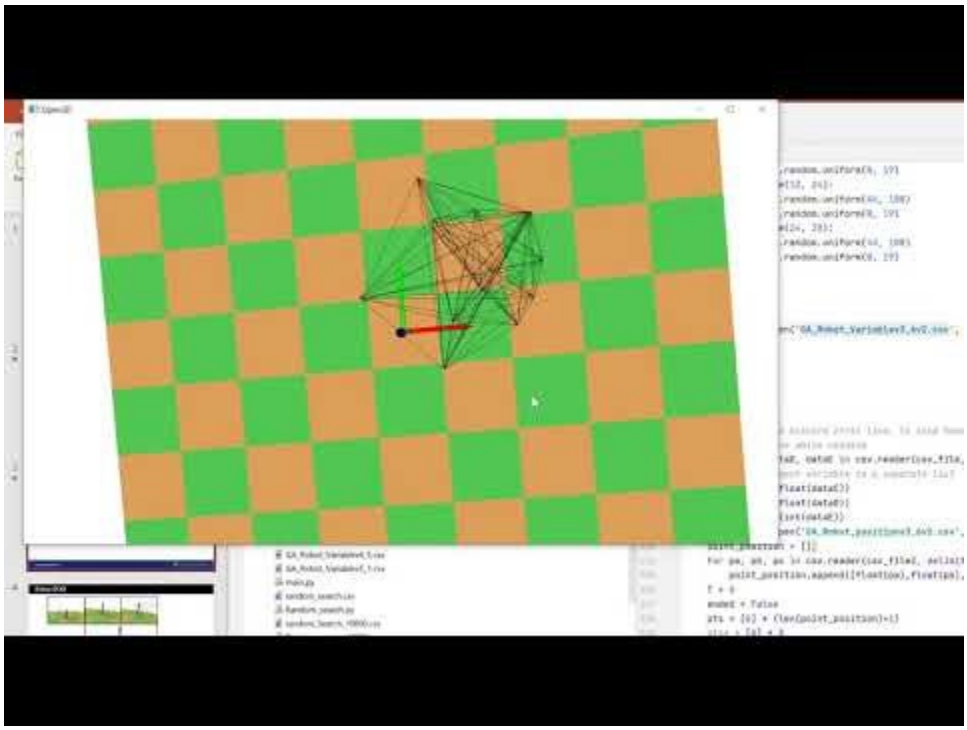


Figure 6 Diversity plot m2

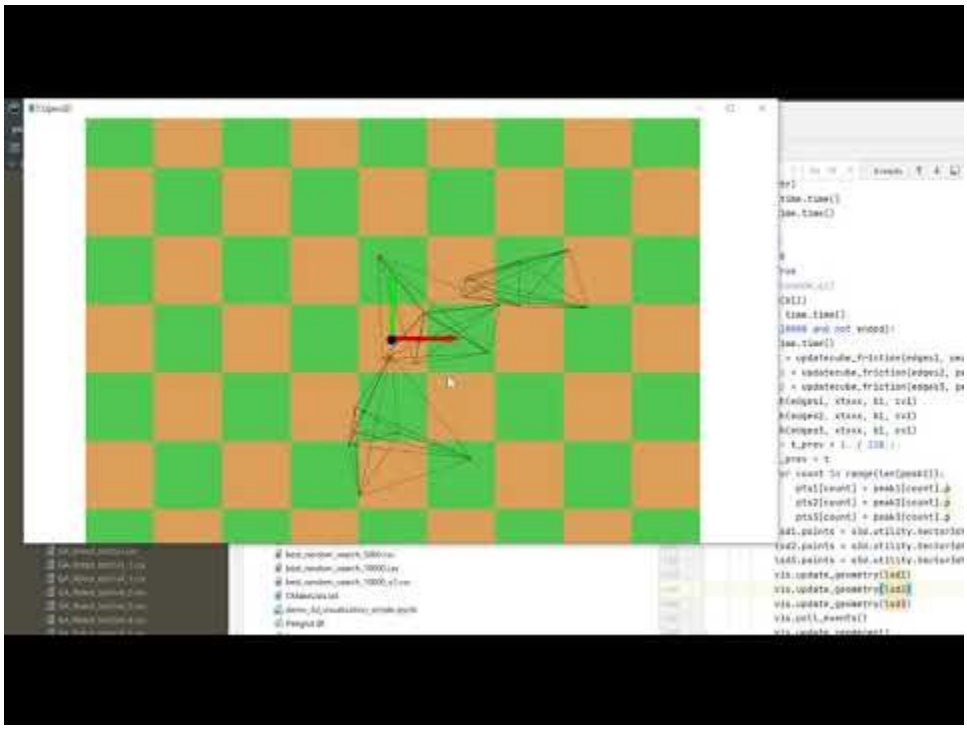
Video:

Robot with most amount of mass:



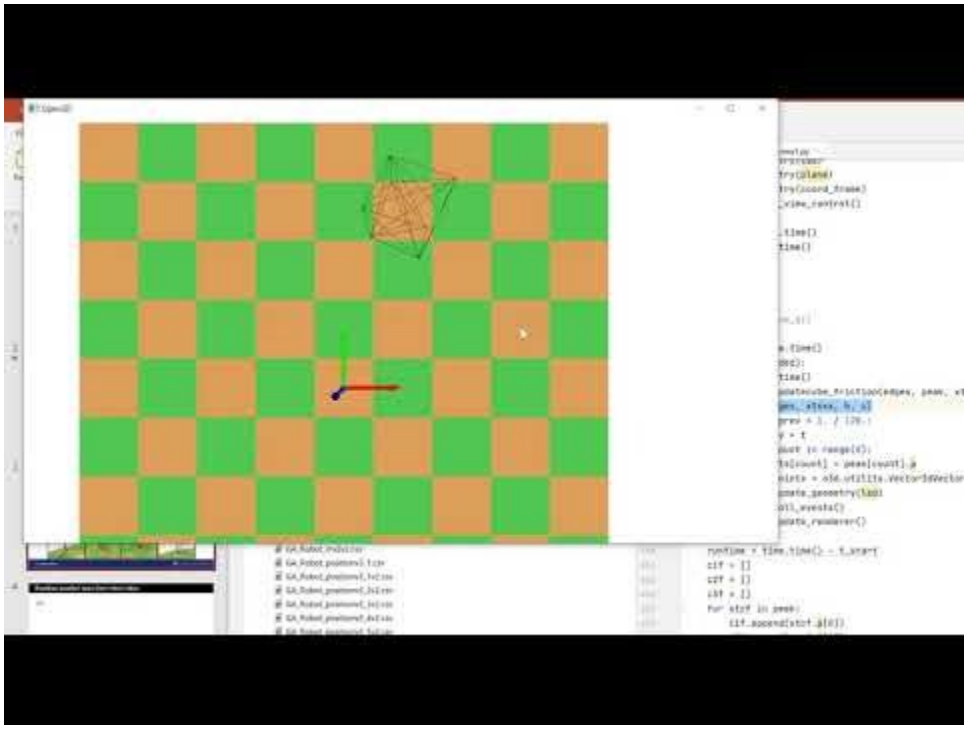
https://youtu.be/wAu5qTco3ac

Multi Robot:



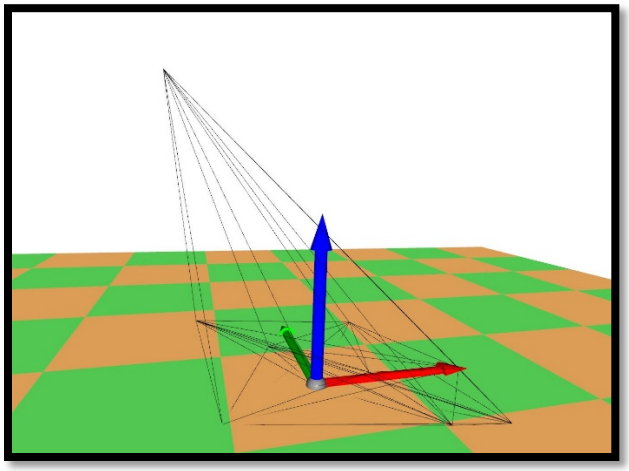
https://youtu.be/aRgRv4vLuU0

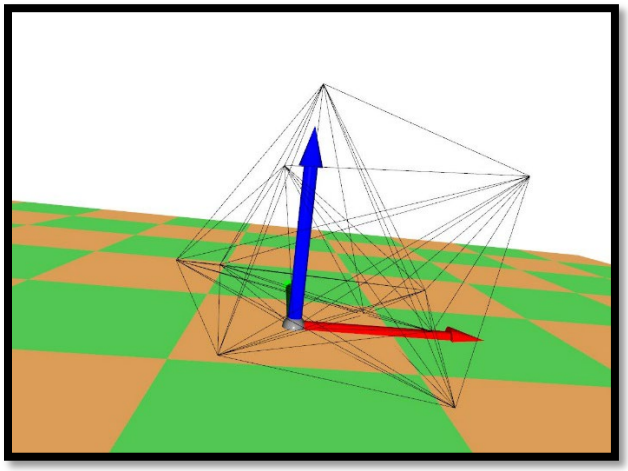
Cube Robot with reduced springs:

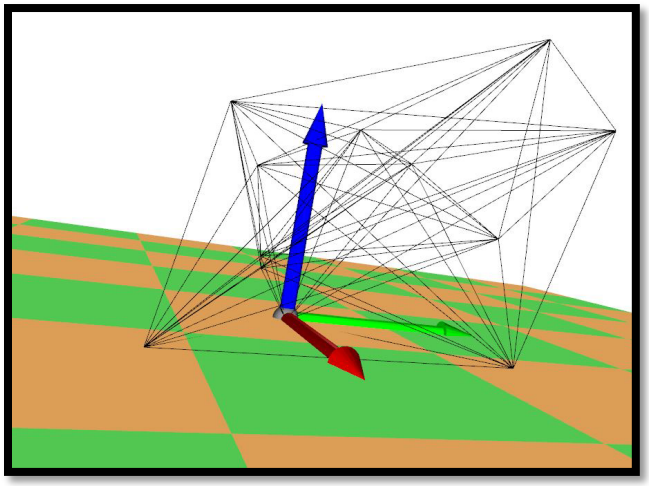


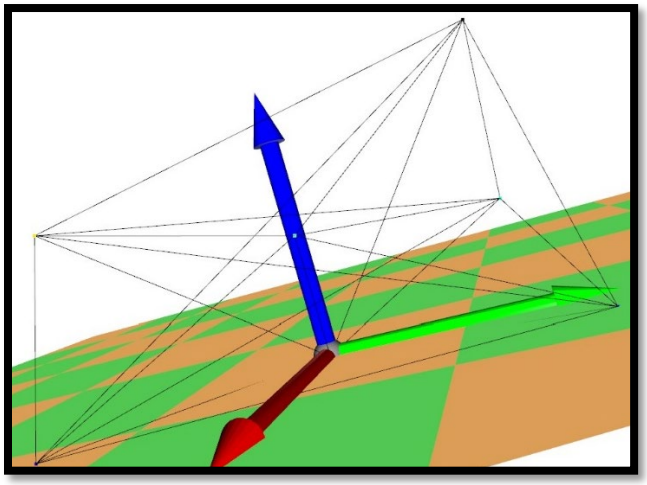
https://youtu.be/8PKGoL-NpGU

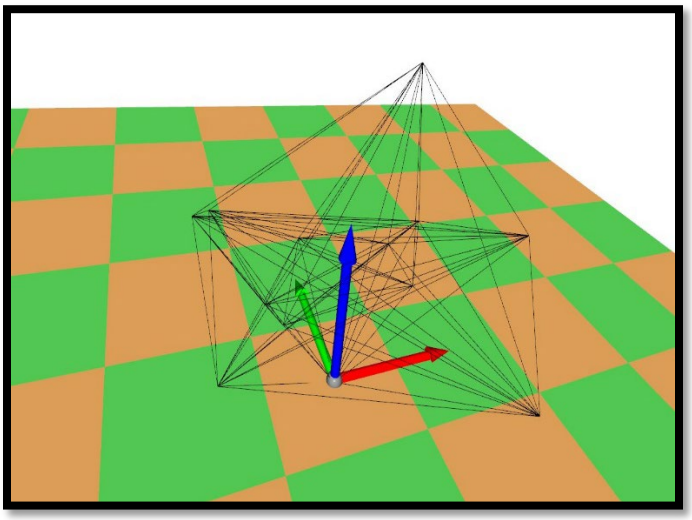
Robot Zoo

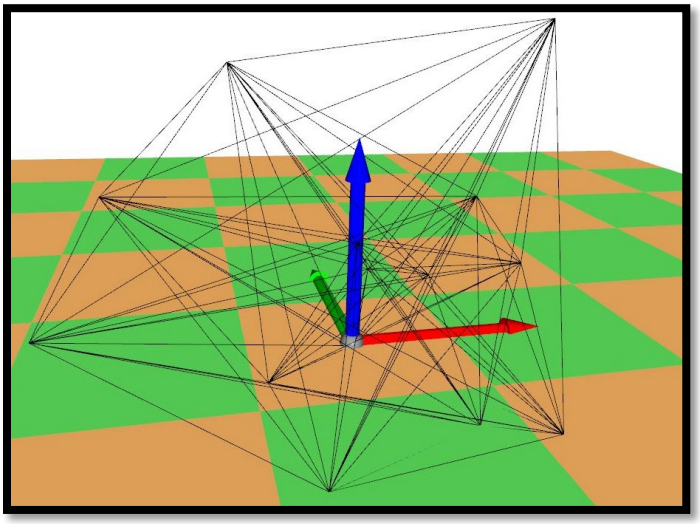


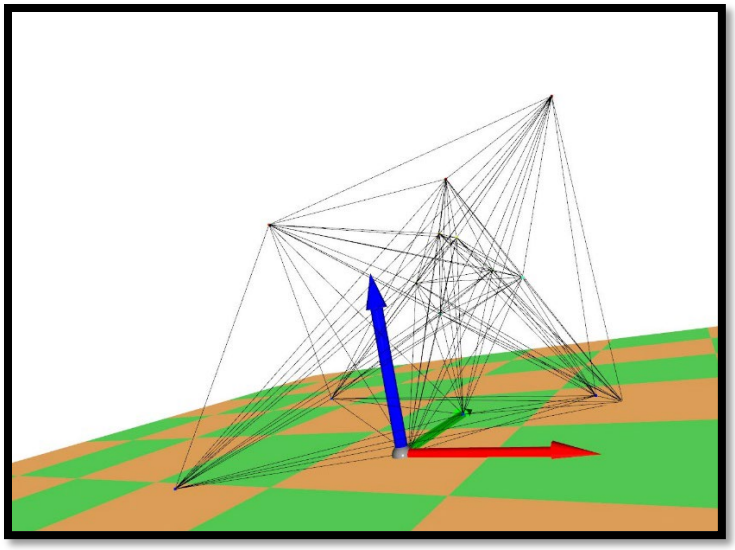


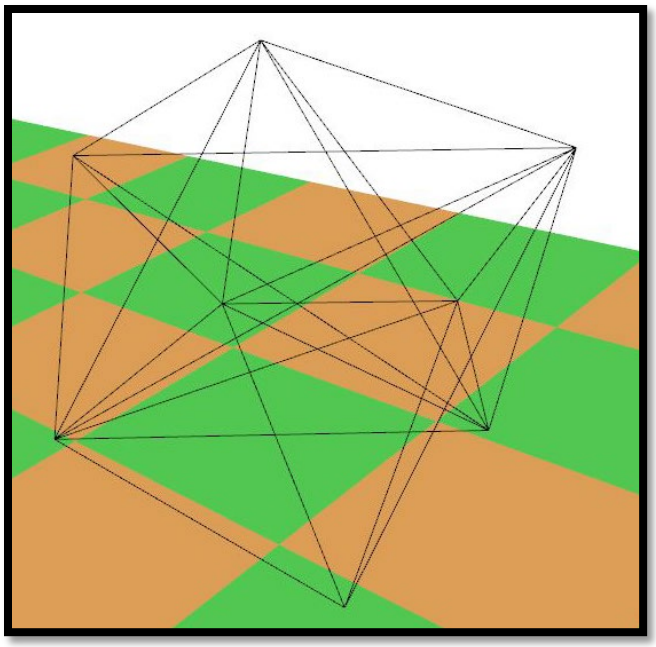


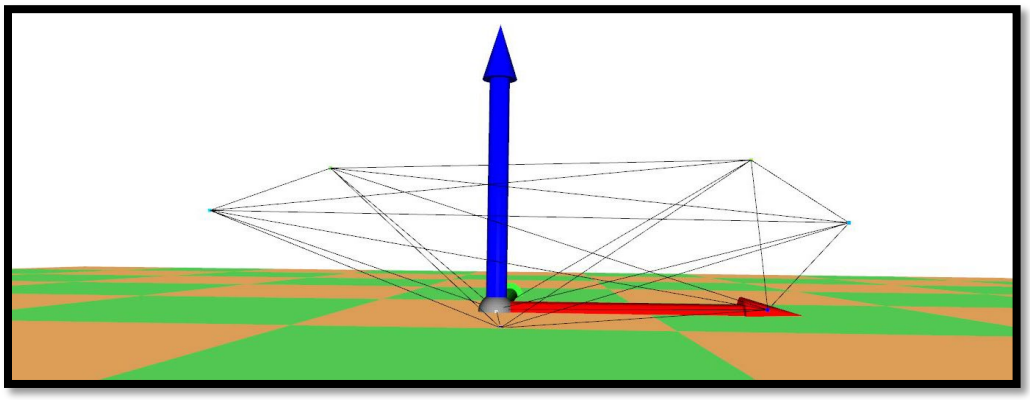












Appendix: Code

```
#include <fstream>
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <vector>
#include <stdarg.h>
#include <numeric>
#include <cstdlib>
```

```
#include <ctime>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <algorithm>
#include <random>
#include <array>
#include <vector>
#define _USE_MATH_DEFINES
#include <cmath>
#include<chrono>
using namespace std;
using namespace std::chrono;
random_device rd;
mt19937 e(rd());
static const double Gravity = 9.81;
static const double dt = 0.0001;
static const double u_s = 0.61;
static const double u_k = 0.47;
static const double damping = 0.9999;
struct mass {
    double m;
    double p[3];
    double v[3];
    double a[3];
};

struct spring {
    double k;
    double L0;
    double L0x;
    int m1;
    int m2;
};

struct sets {
    int pn; /*number of points*/
    int sn; /*number of springs*/
    vector<double> p0; /*x position*/
    vector<double> p1; /*y position*/
    vector<double> p2; /*z position*/
    vector<double> b;
```

```
    vector<double> c;
    vector<int> sck;
//    double p[7][3];
//    double b[28];
//    double c[28];
//    int sck[28];
    double speed;
};

bool compare(spring a, spring b) {
//for descending order replace with a.distancce >b.distancce
    if (a.L0 < b.L0)
        return 1;
    else
        return 0;
}

bool comparevar(sets a, sets b) {
//for descending order replace with a.distancce >b.distancce
    if (a.speed > b.speed)
        return 1;
    else
        return 0;
}

void copy_var(sets origin[], sets tops[]){
    for(int i = 0; i<10; i++){
        int sn = 0;
        int pn = 0;
        sn = origin[i].sn;
        pn = origin[i].pn-1;
        tops[i].sn = sn;
        tops[i].pn = pn+1;
        tops[i].p0.resize(pn);
        tops[i].p1.resize(pn);
        tops[i].p2.resize(pn);
        tops[i].b.resize(sn);
        tops[i].c.resize(sn);
        tops[i].sck.resize(sn);
        for(int k = 0; k<pn;k++){
            tops[i].p0[k] = origin[i].p0[k];
            tops[i].p1[k] = origin[i].p1[k];
```

```
            tops[i].p2[k] = origin[i].p2[k];
        }
        for (int j =0; j<sn; j++){
            tops[i].b[j] = origin[i].b[j];
            tops[i].c[j] = origin[i].c[j];
            tops[i].sck[j] = origin[i].sck[j];
        }
        tops[i].speed = 0;
    }
}

/*Randomly generate all element needed for robot*/
void random_gv3(sets var[]) {
    uniform_real_distribution<double> first(70, 100);
    uniform_real_distribution<double> cxt(0, 19);
    uniform_int_distribution<int> spc1(5, 20);
    uniform_int_distribution<int> pointnumber(6, 15);
    uniform_real_distribution<double> posxy(-0.1,0.1);
    uniform_real_distribution<double> posz(0,0.1);
    for(int count =0; count<20;count++){
        int pn = 0;
        int sn = 0;
        pn = pointnumber(e)+1;
        sn = (pn*(pn-1))/2;
        var[count].pn = pn;
        var[count].sn = sn;
        var[count].p0.resize(pn-1);
        var[count].p1.resize(pn-1);
        var[count].p2.resize(pn-1);
        var[count].b.resize(sn);
        var[count].c.resize(sn);
        var[count].sck.resize(sn);
        for(int i =0; i<pn-1;i++){
            var[count].p0[i] = posxy(e);
            var[count].p1[i] = posxy(e);
            var[count].p2[i] = posz(e);
        }
        for (int i = 0; i < sn; i++) {
            var[count].b[i] = first(e);
            var[count].c[i] = cxt(e);
            var[count].sck[i] = spc1(e) * 1000;
        }
```

```
        var[count].speed = 0;
    }
}

void mass_initv3(vector<mass> &point, double weight, double sp[], sets var) {
    for (int i = 0; i < var.pn; i++) {
        point[i].m = weight;
        point[i].v[0] = 0;
        point[i].v[1] = 0;
        point[i].v[2] = 0;
        point[i].a[0] = 0;
        point[i].a[1] = 0;
        point[i].a[2] = 0;
    }
    point[0].p[0] = sp[0];
    point[0].p[1] = sp[1];
    point[0].p[2] = sp[2];
    for(int i = 1; i<var.pn;i++){
        point[i].p[0] = sp[0]+var.p0[i-1];
        point[i].p[1] = sp[1]+var.p1[i-1];
        point[i].p[2] = sp[2]+var.p2[i-1];
    }
}
void spring_initv3(vector<spring> &edge, vector<mass> &point, vector<int> &spc, int sn, int pn) {
    for (int count = 0; count < sn; count++) {
        edge[count].k = spc[count];
    }
    int k = 0;
    for (int i = 0; i < pn; i++) {
        for (int j = i + 1; j < pn; j++) {
            double distance = 0;
            edge[k].m1 = i;
            edge[k].m2 = j;
            distance = sqrt(pow((point[i].p[0] - point[j].p[0]), 2) + pow((point[i].p[1] - point[j].p[1]), 2) +
pow((point[i].p[2] - point[j].p[2]), 2));
            edge[k].L0 = distance;
            edge[k].L0x = distance;
            k++;
        }
    }
}

void breathv3(vector<spring> &edge, double T, vector<double> &b, vector<double> &c, int sn) {
```

```
    for (int i = 0; i < sn; i++) {
        edge[i].L0 = edge[i].L0x + sin(23 * T + c[i]) / b[i];
    }
}

double update_cube_firctionv3(vector<spring> &edge, vector<mass> &point, double cT, int pn, int sn) {
    vector<double> fx;
    vector<double> fy;
    vector<double> fz;
    vector<bool> groundflag;
    fx.resize(pn);
    fy.resize(pn);
    fz.resize(pn);
    groundflag.resize(pn);
    for (int count = 0; count < pn; count++) {
        fx[count] = 0;
        fy[count] = 0;
        fz[count] = 0;
        groundflag[count] = false;
    }
    for (int i = 0; i < pn; i++) {
        for (int j = 0; j < sn; j++) {
            if (edge[j].m1 == i || edge[j].m2 == i) {
                int self = 0;
                int opp = 0;
                double X = 0;
                double Y = 0;
                double Z = 0;
                double L = 0;
                double F = 0;
                if (edge[j].m1 == i) {
                    self = edge[j].m1;
                    opp = edge[j].m2;
                } else if (edge[j].m2 == i) {
                    self = edge[j].m2;
                    opp = edge[j].m1;
                }
                X = pow((point[opp].p[0] - point[self].p[0]), 2);
                Y = pow((point[opp].p[1] - point[self].p[1]), 2);
                Z = pow((point[opp].p[2] - point[self].p[2]), 2);
                L = sqrt(X + Y + Z);
                F = edge[j].k * (L - edge[j].L0);
```

```
                fx[i] = fx[i] - F * ((point[self].p[0] - point[opp].p[0]) / L);
                fy[i] = fy[i] - F * ((point[self].p[1] - point[opp].p[1]) / L);
                fz[i] = fz[i] - F * ((point[self].p[2] - point[opp].p[2]) / L);
            }
        }
        fz[i] = fz[i] - point[i].m * Gravity;
        if (point[i].p[2] <= 0) {
            double fh = sqrt(pow(fx[i], 2) + pow(fy[i], 2));
            if (fh < abs(fz[i] * u_s)) {
                groundflag[i] = true;
            } else {
                fx[i] = fx[i] - (fx[i] / fh) * abs(fz[i] * u_k);
                fy[i] = fy[i] - (fy[i] / fh) * abs(fz[i] * u_k);
            }
            fz[i] = fz[i] + (0 - point[i].p[2]) * 100000;
        }
    }
    for (int count = 0; count < pn; count++) {
        point[count].a[0] = fx[count] / point[count].m;
        point[count].a[1] = fy[count] / point[count].m;
        point[count].a[2] = fz[count] / point[count].m;
        point[count].v[0] = point[count].v[0] + point[count].a[0] * dt;
        point[count].v[1] = point[count].v[1] + point[count].a[1] * dt;
        point[count].v[2] = point[count].v[2] + point[count].a[2] * dt;
        point[count].v[0] = damping * point[count].v[0];
        point[count].v[1] = damping * point[count].v[1];
        point[count].v[2] = damping * point[count].v[2];
        if (groundflag[count] == true) {
            point[count].v[0] = 0;
            point[count].v[1] = 0;
            groundflag[count] = false;
        }
        point[count].p[0] = point[count].p[0] + point[count].v[0] * dt;
        point[count].p[1] = point[count].p[1] + point[count].v[1] * dt;
        point[count].p[2] = point[count].p[2] + point[count].v[2] * dt;
    }
    return cT + dt;
}

/*Mutation version 1*/
void mutv3(sets &Var) {
    int indexx1 = 0;
```

```
    int indexx2 = 0;
    int sindexx = 0;
    uniform_int_distribution<int> massindex(0, Var.pn-2);
    uniform_int_distribution<int> springindex(0, Var.sn-1);
    uniform_real_distribution<double> first(70, 100);
    uniform_real_distribution<double> cxt(0, 19);
    uniform_int_distribution<int> spc1(5, 20);
    uniform_real_distribution<double> posxy(-0.1,0.1);
    uniform_real_distribution<double> posz(0,0.2);
    indexx1 = massindex(e);
    indexx2 = massindex(e);
    sindexx = springindex(e);
    while (indexx1 == indexx2) {
        indexx2 = massindex(e);
    }
    Var.p0[indexx1] = posxy(e);
    Var.p1[indexx1] = posxy(e);
    Var.p2[indexx1] = posz(e);
    Var.p0[indexx2] = posxy(e);
    Var.p1[indexx2] = posxy(e);
    Var.p2[indexx2] = posz(e);
    Var.b[sindexx] = first(e);
    Var.c[sindexx] = cxt(e);
    Var.sck[sindexx] = spc1(e) * 1000;
}
/*Spring mutation*/
void mutv3(sets &Var) {
//    cout<<"enter mut\n";
    int indexx1 = 0;
    int indexx2 = 0;
    int sindexx = 0;
    int newsize = 0;
    int newindex = 0;
    int sn = Var.sn;

    uniform_int_distribution<int> springindex(7,27);
    uniform_int_distribution<int> size(7,28);
    uniform_real_distribution<double> first(70, 100);
    uniform_real_distribution<double> cxt(0, 19);
    uniform_int_distribution<int> spc1(5, 20);
    newsize = size(e);
    while(newsize == sn){
```

```
        newsize = size(e);
    }
    if(newsize<sn){
//        cout<<"newsize<sn\n";
        Var.sn = newsize;
//        cout<<"new size is: "<<newsize<<"\n";
//        cout<<Var.spring.size()<<endl;
//        cout<<"step1\n";
        for(int icx = newsize; icx<sn;icx++){
            Var.passed[Var.spring[icx]] = false;
        }
//        cout<<"step2\n";
        Var.spring.resize(newsize);
//        cout<<"step3\n";
    }else{
//        cout<<"newsize>sn\n";
        Var.sn = newsize;
        Var.spring.resize(newsize);
//        cout<<"new size is: "<<newsize<<"\n";
//        cout<<Var.spring.size()<<endl;
        for(int icx = sn; icx<newsize;icx++){
            newindex = springindex(e);
            while (Var.passed[newindex] == true){
                newindex = springindex(e);
            }
            Var.spring[icx] = newindex;
            Var.passed[newindex] = true;
            Var.b[newindex] = first(e);
            Var.c[newindex] = cxt(e);
            Var.sck[newindex] = spc1(e) * 1000;
        }
    }
//    cout<<"finish resizing\n";
    if(Var.spring.size()==28 || Var.spring.size()==7){
//        cout<<"newsize equal 28\n";
        return;
    }
//    cout<<"newsize does not equal 28\n";
    uniform_int_distribution<int> index(7,newsize-1);
    indexx1 = index(e);
    indexx2 = springindex(e);
//    cout<<newsize<<"\n";
```

```
//    cout<<Var.spring.size()<<endl;
//    cout<<indexx1<<" indexx1\n";
    while(Var.spring[indexx1] == indexx2 || Var.passed[indexx2] == true){
        indexx2 = springindex(e);
    }
    Var.passed[Var.spring[indexx1]] = false;
//    cout<<"false ok\n";
    Var.passed[indexx2] = true;
//    cout<<"true ok\n";
    Var.spring[indexx1] = indexx2;
//    cout<<"left mut\n";
}
/*mutation Version 2*/
void mutv3(sets &Var) {
    int indexx1 = 0;
    int indexx2 = 0;
    int sindexx = 0;
    int pn = 0;
    int sn = 0;
    int temppn = 0;
    int tempsn = 0;
    uniform_int_distribution<int> massindex(0, Var.pn - 2);
    uniform_int_distribution<int> springindex(0, Var.sn - 1);
    uniform_real_distribution<double> first(70, 100);
    uniform_real_distribution<double> cxt(0, 19);
    uniform_int_distribution<int> spc1(5, 20);
    uniform_real_distribution<double> posxy(-0.1, 0.1);
    uniform_real_distribution<double> posz(0, 0.1);
    uniform_int_distribution<int> pointnumber(6, 15);
    indexx1 = massindex(e);
    indexx2 = massindex(e);
    sindexx = springindex(e);
    while (indexx1 == indexx2) {
        indexx2 = massindex(e);
    }
    Var.p0[indexx1] = posxy(e);
    Var.p1[indexx1] = posxy(e);
    Var.p2[indexx1] = posz(e);
    Var.p0[indexx2] = posxy(e);
    Var.p1[indexx2] = posxy(e);
    Var.p2[indexx2] = posz(e);
    Var.b[sindexx] = first(e);
```

```
    Var.c[sindexx] = cxt(e);
    Var.sck[sindexx] = spc1(e) * 1000;
    pn = pointnumber(e)+1;
    while(pn==Var.pn){
        pn = pointnumber(e)+1;
    }
    sn = (pn*(pn-1))/2;
    Var.p0.resize(pn-1);
    Var.p1.resize(pn-1);
    Var.p2.resize(pn-1);
    Var.b.resize(sn);
    Var.c.resize(sn);
    Var.sck.resize(sn);
    if(Var.pn<pn){
        temppn = Var.pn-1;
        tempsn = Var.sn;
        for(int i =temppn;i<pn-1;i++){
            Var.p0[i] = posxy(e);
            Var.p1[i] = posxy(e);
            Var.p2[i] = posz(e);
        }
        for (int j = tempsn; j < sn; j++) {
            Var.b[j] = first(e);
            Var.c[j] = cxt(e);
            Var.sck[j] = spc1(e) * 1000;
        }
        Var.pn = pn;
        Var.sn = sn;
    }else{
        Var.pn = pn;
        Var.sn = sn;
    }
}
/*Crossover version 1*/
void crossoverv3(sets tops[], sets children[]){
    uniform_int_distribution<int> rate(0, 20);
    uniform_int_distribution<int> parents(0, 9);
    int cr = 0;
    int mr = 0;
    int cp1 =0;
    int cp2 =0;
    int father = 0;
```

```
    int mother = 0;
    int fpn;
    int mpn;
    int cpn;
    int lpn;
    int lspn;
    for(int i =0; i<20; i++){
        cr = rate(e);
        if(cr == 0 || cr ==2 || cr ==4 || cr ==6 || cr == 8 || cr == 10|| cr == 12 || cr == 14 || cr == 16 || cr
==18){
            father = parents(e);
            mother = parents(e);
            while(father==mother){
                mother = parents(e);
            }
            fpn = tops[father].pn;
            mpn = tops[mother].pn;
            if(fpn>=mpn) {
                cpn = mpn;
                lpn = fpn;
                lspn = tops[father].sn;
                uniform_int_distribution<int> crossposition(1, cpn-2);
                children[i].sn = lspn;
                children[i].pn = lpn;
                children[i].p0.resize(lpn);
                children[i].p1.resize(lpn);
                children[i].p2.resize(lpn);
                children[i].b.resize(lspn);
                children[i].c.resize(lspn);
                children[i].sck.resize(lspn);
                cp1 = crossposition(e);
                for (int ctx = 0; ctx < cp1; ctx++) {
                    children[i].p0[ctx] = tops[mother].p0[ctx];
                    children[i].p1[ctx] = tops[mother].p1[ctx];
                    children[i].p2[ctx] = tops[mother].p2[ctx];
                }
                for(int ctx = cp1; ctx<lpn-1;ctx++){
                    children[i].p0[ctx] = tops[father].p0[ctx];
                    children[i].p1[ctx] = tops[father].p1[ctx];
                    children[i].p2[ctx] = tops[father].p2[ctx];
                }
                for(int spn = 0; spn<lspn;spn++){
                    children[i].b[spn] = tops[father].b[spn];
```

```
            children[i].c[spn] = tops[father].c[spn];
            children[i].sck[spn] = tops[father].sck[spn];
        }
    }else {
        cpn = fpn;
        lpn = mpn;
        lspn = tops[mother].sn;
        uniform_int_distribution<int> crossposition(1, cpn-2);
        children[i].sn = lspn;
        children[i].pn = lpn;
        children[i].p0.resize(lpn);
        children[i].p1.resize(lpn);
        children[i].p2.resize(lpn);
        children[i].b.resize(lspn);
        children[i].c.resize(lspn);
        children[i].sck.resize(lspn);
        cp1 = crossposition(e);
        for (int ctx = 0; ctx < cp1; ctx++) {
            children[i].p0[ctx] = tops[father].p0[ctx];
            children[i].p1[ctx] = tops[father].p1[ctx];
            children[i].p2[ctx] = tops[father].p2[ctx];
        }
        for (int ctx = cp1; ctx < lpn-1; ctx++) {
            children[i].p0[ctx] = tops[mother].p0[ctx];
            children[i].p1[ctx] = tops[mother].p1[ctx];
            children[i].p2[ctx] = tops[mother].p2[ctx];
        }
        for (int spn = 0; spn < lspn; spn++) {
            children[i].b[spn] = tops[mother].b[spn];
            children[i].c[spn] = tops[mother].c[spn];
            children[i].sck[spn] = tops[mother].sck[spn];
        }
    }
}else{
    father = parents(e);
    children[i].pn = tops[father].pn;
    children[i].sn = tops[father].sn;
    children[i].p0.resize(tops[father].pn-1);
    children[i].p1.resize(tops[father].pn-1);
    children[i].p2.resize(tops[father].pn-1);
    children[i].b.resize(tops[father].sn);
    children[i].c.resize(tops[father].sn);
```

```
            children[i].sck.resize(tops[father].sn);
            for(int ct = 0; ct<tops[father].pn-1;ct++){
                children[i].p0[ct] = tops[father].p0[ct];
                children[i].p1[ct] = tops[father].p1[ct];
                children[i].p2[ct] = tops[father].p2[ct];
            }
            for(int ctx = 0; ctx < tops[father].sn; ctx++){
                children[i].b[ctx] = tops[father].b[ctx];
                children[i].c[ctx] = tops[father].c[ctx];
                children[i].sck[ctx] = tops[father].sck[ctx];
            }
        }
        children[i].speed = 0;
        mr = rate(e);
        if(mr == 1 || mr ==3 || mr ==5 || mr ==7 || mr == 9 || mr == 11|| mr == 13 || mr == 15 || mr == 17 || mr
==19){
            mutv3(children[i]);
        }
    }
}


/*Spring Crossover*/
void crossoverv3(sets tops[], sets children[]){
    uniform_int_distribution<int> rate(0, 20);
    uniform_int_distribution<int> parents(0, 9);
    uniform_int_distribution<int> springindex(7,27);
    int cr = 0;
    int mr = 0;
    int cp1 =0;
    int fs = 0;
    int ms = 0;
    int ss = 0;
    int fsn;
    int lsn;
    int an = 0;
    int father = 0;
    int mother = 0;
    for(int i =0; i<20; i++){
        cr = rate(e);
        if(cr == 0 || cr ==2 || cr ==4 || cr ==6 || cr == 8 || cr == 10|| cr == 12 || cr == 14 || cr == 16 || cr
==18){
            father = parents(e);
            mother = parents(e);
```

```
            while(father == mother || (tops[father].sn == 7 && tops[mother].sn == 7)){
                mother = parents(e);
            }
            fs = tops[father].sn;
            ms = tops[mother].sn;
            for(int ct = 0; ct<28;ct++){
                children[i].passed[ct] = false;
            }
//            cout<<ms<<"ms before compare\n";
//            cout<<fs<<"fs before compare\n";
            if(fs == 7){
//                cout<<ms<<"ms\n";
                fsn = fs+tops[mother].sn/2;
                children[i].sn = fsn;
                children[i].spring.resize(fsn);
                for(int ctx = 0; ctx<7;ctx++){
                    children[i].spring[ctx]  = ctx;
                    children[i].passed[ctx] = true;
                    children[i].b[ctx] = tops[father].b[ctx];
                    children[i].c[ctx] = tops[father].c[ctx];
                    children[i].sck[ctx] = tops[father].sck[ctx];
                }
                for(int ctx = 7; ctx<fsn;ctx++){
                    children[i].spring[ctx]  = tops[mother].spring[ctx];
                    children[i].passed[tops[mother].spring[ctx]] = true;
                    children[i].b[tops[mother].spring[ctx]] = tops[mother].b[tops[mother].spring[ctx]];
                    children[i].c[tops[mother].spring[ctx]] = tops[mother].c[tops[mother].spring[ctx]];
                    children[i].sck[tops[mother].spring[ctx]] = tops[mother].sck[tops[mother].spring[ctx]];
                }
//                cout<<"left ms\n";
            }else if(ms == 7 ){
//                cout<<fs<<"fs\n";
                fsn = fs+tops[father].sn/2;
                children[i].sn = fsn;
                children[i].spring.resize(fsn);
                for(int ctx = 0; ctx<7;ctx++){
                    children[i].spring[ctx]  = ctx;
                    children[i].passed[ctx] = true;
                    children[i].b[ctx] = tops[mother].b[ctx];
                    children[i].c[ctx] = tops[mother].c[ctx];
                    children[i].sck[ctx] = tops[mother].sck[ctx];
                }
```

```
                for(int ctx = 7; ctx<fsn;ctx++){
                    children[i].spring[ctx]  = tops[father].spring[ctx];
                    children[i].passed[tops[father].spring[ctx]] = true;
                    children[i].b[tops[father].spring[ctx]] = tops[father].b[tops[father].spring[ctx]];
                    children[i].c[tops[father].spring[ctx]] = tops[father].c[tops[father].spring[ctx]];
                    children[i].sck[tops[father].spring[ctx]] = tops[father].sck[tops[father].spring[ctx]];
                }
//                cout<<"left fs\n";
            }else if(ms>=fs){
//                cout<<"enter ms>fs\n";
                ss = fs;
                fsn = (ms+fs)/2;
                uniform_int_distribution<int> crosspos(7,ss);
                cp1 = crosspos(e);
                children[i].spring.resize(fsn);
                children[i].sn = fsn;
                for(int ctx = 0; ctx<7;ctx++){
                    children[i].spring[ctx]  = ctx;
                    children[i].passed[ctx] = true;
                    children[i].b[ctx] = tops[father].b[ctx];
                    children[i].c[ctx] = tops[father].c[ctx];
                    children[i].sck[ctx] = tops[father].sck[ctx];
                }
                for(int ctx = 7; ctx <cp1;ctx++){
                    children[i].spring[ctx]  = tops[father].spring[ctx];
                    children[i].passed[tops[father].spring[ctx]] = true;
                    children[i].b[tops[father].spring[ctx]] = tops[father].b[tops[father].spring[ctx]];
                    children[i].c[tops[father].spring[ctx]] = tops[father].c[tops[father].spring[ctx]];
                    children[i].sck[tops[father].spring[ctx]] = tops[father].sck[tops[father].spring[ctx]];
                }
                for(int ctx = cp1; ctx<fsn;ctx++){
                    an = tops[mother].spring[ctx];
                    while(children[i].passed[an] == true){
                        an = springindex(e);
                    }
                    children[i].spring[ctx] = an;
                    children[i].passed[an] = true;
                    children[i].b[tops[mother].spring[ctx]] = tops[mother].b[tops[mother].spring[ctx]];
                    children[i].c[tops[mother].spring[ctx]] = tops[mother].c[tops[mother].spring[ctx]];
                    children[i].sck[tops[mother].spring[ctx]] = tops[mother].sck[tops[mother].spring[ctx]];
                }
//                cout<<"left ms>fs\n";
```

```
            }else if(fs>ms){
//                cout<<"enter fs>ms\n";
                ss = ms;
                fsn = (ms+fs)/2;
                uniform_int_distribution<int> crosspos(7,ss);
                cp1 = crosspos(e);
                children[i].spring.resize(fsn);
                children[i].sn = fsn;
                for(int ctx = 0; ctx<7;ctx++){
                    children[i].spring[ctx]  = ctx;
                    children[i].passed[ctx] = true;
                    children[i].b[ctx] = tops[mother].b[ctx];
                    children[i].c[ctx] = tops[mother].c[ctx];
                    children[i].sck[ctx] = tops[mother].sck[ctx];
                }
                for(int ctx = 7; ctx <cp1;ctx++){
                    children[i].spring[ctx]  = tops[mother].spring[ctx];
                    children[i].passed[tops[mother].spring[ctx]] = true;
                    children[i].b[tops[mother].spring[ctx]] = tops[mother].b[tops[mother].spring[ctx]];
                    children[i].c[tops[mother].spring[ctx]] = tops[mother].c[tops[mother].spring[ctx]];
                    children[i].sck[tops[mother].spring[ctx]] = tops[mother].sck[tops[mother].spring[ctx]];
                }
                for(int ctx = cp1; ctx<fsn;ctx++){
                    an = tops[father].spring[ctx];
                    while(children[i].passed[an] == true){
                        an = springindex(e);
                    }
                    children[i].spring[ctx] = an;
                    children[i].passed[an] = true;
                    children[i].b[tops[father].spring[ctx]] = tops[father].b[tops[father].spring[ctx]];
                    children[i].c[tops[father].spring[ctx]] = tops[father].c[tops[father].spring[ctx]];
                    children[i].sck[tops[father].spring[ctx]] = tops[father].sck[tops[father].spring[ctx]];
                }
//                cout<<"left fs>ms\n";
            }

        }else{
            father = parents(e);
            lsn = tops[father].sn;
            children[i].sn = lsn;
            children[i].spring.resize(lsn);
            for(int ps = 0; ps<28;ps++){
```

```
                children[i].passed[ps] = false;
            }
            for(int ct = 0; ct<lsn;ct++){
                children[i].spring[ct] = tops[father].spring[ct];
                children[i].passed[tops[father].spring[ct]] = true;
            }
            for(int ctx = 0; ctx < 28; ctx++){
                children[i].b[ctx] = tops[father].b[ctx];
                children[i].c[ctx] = tops[father].c[ctx];
                children[i].sck[ctx] = tops[father].sck[ctx];
            }
        }
        children[i].speed = 0;
        mr = rate(e);
        if(mr == 1 || mr ==3 || mr ==5 || mr ==7 || mr == 9 || mr == 11|| mr == 13 || mr == 15 || mr == 17 || mr
==19){
            mutv3(children[i]);
        }
    }
}


int main() {
    vector<mass> point;
    vector<spring> edge;
    sets varx[20];
    sets tophalf[10];
    double sp[3];
    double bestspeed;
    sp[0] = 0.0;
    sp[1] = 0.0;
    sp[2] = 0.0;
    double ctm[3];
    ctm[0] = 0;
    ctm[1] = 0;
    ctm[2] = 0;
    ofstream fout2("GA_Robot_test1v3_6v2.csv", ios::out);
    ofstream fout4("GA_robot_for_diversity_sp.csv",ios::out);
    ofstream fout5("GA_robot_for_diversity_np.csv",ios::out);
    random_gv3(varx);
    high_resolution_clock::time_point t1 = high_resolution_clock::now();
    for (int pop = 0; pop < 20; pop++) {
        int T = 0;
        double xtxxx = 0;
```

```
        point.resize(varx[pop].pn);
        edge.resize(varx[pop].sn);
        mass_initv3(point, 1, sp, varx[pop]);
        spring_initv3(edge, point, varx[pop].sck, varx[pop].sn, varx[pop].pn);
        ctm[0] = 0;
        ctm[1] = 0;
        ctm[2] = 0;
        for (int xtc = 0; xtc < varx[pop].pn; xtc++) {
            ctm[0] += point[xtc].p[0];
            ctm[1] += point[xtc].p[1];
            ctm[2] += point[xtc].p[2];
        }
        ctm[0] = ctm[0] / varx[pop].pn;
        ctm[1] = ctm[1] / varx[pop].pn;
        ctm[2] = ctm[2] / varx[pop].pn;
        while (T < 1000) {
            xtxxx = update_cube_firctionv3(edge, point, xtxxx, varx[pop].pn, varx[pop].sn);
            breathv3(edge, xtxxx, varx[pop].b, varx[pop].c, varx[pop].sn);
            T = T + 1;
        }
        double ctmf[3];
        ctmf[0] = 0;
        ctmf[1] = 0;
        ctmf[2] = 0;
        for (int xtc = 0; xtc < varx[pop].pn; xtc++) {
            ctmf[0] += point[xtc].p[0];
            ctmf[1] += point[xtc].p[1];
            ctmf[2] += point[xtc].p[2];
        }
        ctmf[0] = ctmf[0] / varx[pop].pn;
        ctmf[1] = ctmf[1] / varx[pop].pn;
        ctmf[2] = ctmf[2] / varx[pop].pn;
        double dist = 0;
        dist = sqrt(pow(ctmf[0] - ctm[0], 2) + pow(ctmf[1] - ctm[1], 2));
//        cout<<dist<<","<<xtxxx<<"\n";
//        cout<<ctmf[0]<<","<<ctmf[1]<<","<<ctmf[2]<<"\n";
        varx[pop].speed = dist / xtxxx;
    }
    int n = sizeof(varx) / sizeof(sets);
    sort(varx, varx + n, comparevar);
    bestspeed = varx[0].speed;
    for (int i = 0; i < 10000; i++) {
```

```
copy_var(varx, tophalf);
crossoverv3(tophalf, varx);
for (int pop = 0; pop < 20; pop++) {
    int T = 0;
    double xtxxx = 0;
    point.resize(varx[pop].pn);
    edge.resize(varx[pop].sn);
    mass_initv3(point, 1, sp, varx[pop]);
    spring_initv3(edge, point, varx[pop].sck, varx[pop].sn, varx[pop].pn);
    ctm[0] = 0;
    ctm[1] = 0;
    ctm[2] = 0;
    for (int xtc = 0; xtc < varx[pop].pn; xtc++) {
        ctm[0] += point[xtc].p[0];
        ctm[1] += point[xtc].p[1];
        ctm[2] += point[xtc].p[2];
    }
    ctm[0] = ctm[0] / varx[pop].pn;
    ctm[1] = ctm[1] / varx[pop].pn;
    ctm[2] = ctm[2] / varx[pop].pn;
    while (T < 1000) {
        xtxxx = update_cube_firctionv3(edge, point, xtxxx, varx[pop].pn, varx[pop].sn);
        breathv3(edge, xtxxx, varx[pop].b, varx[pop].c, varx[pop].sn);
        T = T + 1;
    }
    double ctmf[3];
    ctmf[0] = 0;
    ctmf[1] = 0;
    ctmf[2] = 0;
    for (int xtc = 0; xtc < varx[pop].pn; xtc++) {
        ctmf[0] += point[xtc].p[0];
        ctmf[1] += point[xtc].p[1];
        ctmf[2] += point[xtc].p[2];
    }
    ctmf[0] = ctmf[0] / varx[pop].pn;
    ctmf[1] = ctmf[1] / varx[pop].pn;
    ctmf[2] = ctmf[2] / varx[pop].pn;
    double dist = 0;
    dist = sqrt(pow(ctmf[0] - ctm[0], 2) + pow(ctmf[1] - ctm[1], 2));
    varx[pop].speed = dist / xtxxx;
}
int n = sizeof(varx) / sizeof(sets);
```

```
        sort(varx, varx + n, comparevar);
        if (varx[0].speed > bestspeed) {
            bestspeed = varx[0].speed;
            cout << "found bestspeed: " << bestspeed << " at evo: " << i << "\n";
            ofstream fout1("GA_Robot_Variablev3_6v2.csv", ios::out);
            ofstream fout3("GA_Robot_positionv3_6v2.csv", ios::out);
            for (int fo = 0; fo < varx[0].sn; fo++) {
                fout1 << varx[0].b[fo] << "," << varx[0].c[fo] << "," << varx[0].sck[fo] << "\n";
            }
            for(int fox = 0; fox<varx[0].pn-1;fox++){
                fout3<< varx[0].p0[fox]<< "," <<varx[0].p1[fox]<< "," <<varx[0].p2[fox]<<"\n";
            }
            fout1.close();
            fout3.close();
        }
        fout2<<bestspeed<<"\n";
        for(int ccx =0; ccx<20; ccx++){
            fout4<<varx[ccx].speed<<",";
            fout5<<varx[ccx].pn<<",";
        }
        fout4<<"\n";
        fout5<<"\n";
        if(i%100==0){
            cout<<i<<"\n";
        }
    }
    fout2.close();
    fout4.close();
    fout5.close();
}
```