Genetic Algorithm Linear Dipoles

This is a partial listing of the Java code for the Genetic Algorithm linear dipoles program.

It includes all the code from the **Initialize** and **Evolution** sections, but does not include the variables' descriptions or the display section. The specification of the variables, for example the type of each variable and reasonable dimensions for the arrays, should be clear from the context.

Initialization section

```
// Set up the initial variables

generation=0;
nstrings=40;
nangles=10;
bestfitness=0.0;

pmut=0.2;
pcross=1.0;
dipole_length=1.0;
gap=2.0;

showstrings=true;
elitism=false;

for (istring=0; istring<nstrings; istring++)
{
   for (iangle=0; iangle<nangles; iangle++)
   {
      string[istring][iangle]=(int)(360.0*Math.random());
   }
}
```

Evolution section

```
/* Now we do the GA bit. First find the fitness of every string. To do this we find the
position of the ends of each dipole, then add together all the interactions. We only
calculate interactions between neighbouring dipoles and ignore interactions between
more distant dipoles. */

newpop=0;
bestfitness=-500.0;
average_fitness=0.0;
```

```
for (istring=0; istring<nstrings; istring++)
{
   energy[istring]=0.0;
   for (iangle=0; iangle<nangles-1; iangle++)
   {
      if(iangle!=0)
      {
         theta=string[istring][iangle-1]*((2.0*3.1415926)/360.0);
         xleft=(dipole_length/2.0)*Math.sin(theta);
         yleft=(dipole_length/2.0)*Math.cos(theta);
      }
      if(iangle!=(nangles-1))
      {
         theta=string[istring][iangle+1]*((2.0*3.1415926)/360.0);
         xright=(dipole_length/2.0)*Math.sin(theta);
         yright=(dipole_length/2.0)*Math.cos(theta);
      }
      theta=string[istring][iangle]*((2.0*3.1415926)/360.0);
      xcentre=(dipole_length/2.0)*Math.sin(theta);
      ycentre=(dipole_length/2.0)*Math.cos(theta);
      if (iangle!=0)
      {
         dx=xcentre-(xleft-gap);
         dy=ycentre-yleft;
         distance=Math.sqrt(dx*dx+dy*dy);
         energy[istring]=energy[istring]+1.0/distance;

         dx=-xcentre-(-xleft-gap);
         dy=ycentre-yleft;
         distance=Math.sqrt(dx*dx+dy*dy);
         energy[istring]=energy[istring]+1.0/distance;

         dx=xcentre-(xleft-gap);
         dy=-ycentre-yleft;
         distance=Math.sqrt(dx*dx+dy*dy);
         energy[istring]=energy[istring]-1.0/distance;

         dx=-xcentre-(xleft-gap);
         dy=-ycentre-yleft;
         distance=Math.sqrt(dx*dx+dy*dy);
         energy[istring]=energy[istring]-1.0/distance;
      }
      if (iangle!=nangles)
      {

// First the repulsive interactions

// top-top
         dx=xcentre-(xright+gap);
```

```
            dy=ycentre-yright;
            distance=Math.sqrt(dx*dx+dy*dy);
            energy[istring]=energy[istring]+1.0/distance;

// bottom-bottom
            dx=-xcentre-(gap-xright);
            dy=ycentre-yright;
            distance=Math.sqrt(dx*dx+dy*dy);
            energy[istring]=energy[istring]+1.0/distance;

// … and then the attractions
// top-bottom
            dx=xcentre-(gap-xright);
            dy=-ycentre-yright;
            distance=Math.sqrt(dx*dx+dy*dy);
            energy[istring]=energy[istring]-1.0/distance;

// bottom-top
            dx=-xcentre-(gap+xright);
            dy=(-ycentre-yright);
            distance=Math.sqrt(dx*dx+dy*dy);
            energy[istring]=energy[istring]-1.0/distance;
        }
    }
}

// Now the energies have been found, determine a fitness for each string

for (istring=0; istring<nstrings; istring++)
{
    fitness[istring]=1.0/(7.5+energy[istring]);
}

/* Truncate the fitness and the energy for display. This truncation, by making a small
change to the fitness of each string, slightly affects the calculation, but the effect is
tiny. It could easily be avoided by storing the truncated fitness in a different array. */

for (istring=0; istring<nstrings; istring++)
{
    energy[istring]=(double)((int)(10000*energy[istring]))/10000.0;
    fitness[istring]=(double)((int)(10000*fitness[istring]))/10000.0;
    average_fitness=average_fitness+fitness[istring];
    if (bestfitness<fitness[istring])
    {
        bestfitness=fitness[istring];
        beststring=istring;
    }
}
average_fitness=average_fitness/nstrings;
```

```
if(showstrings)  // Set up the strings to be displayed
{
   s0="   0      " + string[0][0] + " " + string[0][1] + " " + string[0][2] + " " +
string[0][3] + " " + string [0][4] + " " + string[0][5] + " " + string[0][6] + " " +
string[0][7] + " " + string[0][8] + " " + string[0][9]+ "        {" + energy[0]+ " , " +
fitness[0] + "}";
   s1="   1      " + string[1][0] + " " + string[1][1] + " " + string[1][2] + " " +
string[1][3] + " " + string [1][4] + " " + string[1][5] + " " + string[1][6] + " " +
string[1][7] + " " + string[1][8] + " " + string[1][9]+ "        {" + energy[1]+ " , " +
fitness[1] + "}";
   s2="   2      " + string[2][0] + " " + string[2][1] + " " + string[2][2] + " " +
string[2][3] + " " + string [2][4] + " " + string[2][5] + " " + string[2][6] + " " +
string[2][7] + " " + string[2][8] + " " + string[2][9]+ "        {" + energy[2]+ " , " +
fitness[2] + "}";
   s3="   3      " + string[3][0] + " " + string[3][1] + " " + string[3][2] + " " +
string[3][3] + " " + string [3][4] + " " + string[3][5] + " " + string[3][6] + " " +
string[3][7] + " " + string[3][8] + " " + string[3][9]+ "        {" + energy[3]+ " , " +
fitness[3] + "}";
   s4="   4      " + string[4][0] + " " + string[4][1] + " " + string[4][2] + " " +
string[4][3] + " " + string [4][4] + " " + string[4][5] + " " + string[4][6] + " " +
string[4][7] + " " + string[4][8] + " " + string[4][9]+ "        {" + energy[4]+ " , " +
fitness[4] + "}";
   s5="   5      " + string[5][0] + " " + string[5][1] + " " + string[5][2] + " " +
string[5][3] + " " + string [5][4] + " " + string[5][5] + " " + string[5][6] + " " +
string[5][7] + " " + string[5][8] + " " + string[5][9]+ "        {" + energy[5]+ " , " +
fitness[5] + "}";
   s6="   6      " + string[6][0] + " " + string[6][1] + " " + string[6][2] + " " +
string[6][3] + " " + string [6][4] + " " + string[6][5] + " " + string[6][6] + " " +
string[6][7] + " " + string[6][8] + " " + string[6][9]+ "        {" + energy[6]+ " , " +
fitness[6] + "}";
   s7="   7      " + string[7][0] + " " + string[6][1] + " " + string[7][2] + " " +
string[7][3] + " " + string [7][4] + " " + string[7][5] + " " + string[7][6] + " " +
string[7][7] + " " + string[7][8] + " " + string[7][9]+ "        {" + energy[7]+ " , " +
fitness[7] + "}";
   s8="   8      " + string[8][0] + " " + string[8][1] + " " + string[8][2] + " " +
string[8][3] + " " + string [8][4] + " " + string[8][5] + " " + string[8][6] + " " +
string[8][7] + " " + string[8][8] + " " + string[8][9]+ "        {" + energy[8]+ " , " +
fitness[8] + "}";
   s9="   9      " + string[9][0] + " " + string[9][1] + " " + string[9][2] + " " +
string[9][3] + " " + string [9][4] + " " + string[9][5] + " " + string[9][6] + " " +
string[9][7] + " " + string[9][8] + " " + string[9][9]+ "        {" + energy[9]+ " , " +
fitness[9] + "}";
}
else
{
   s0=s1=s2=s3=s4=s5=s6=s7=s8=s9="";
}
```

/* Select the strings for the next generation using a binary tournament. If elitism applies, start by placing the best string directly into the next population. */

```
if (elitism)
{
   for (iangle=0; iangle<nangles; iangle++)
   {
      newstring[newpop][iangle]=string[beststring][iangle];
   }
newpop++;
}

/* Now run the binary tournament */

do
{
   int string1, string2;
   string1=(int)((nstrings+1)*Math.random()); // pick a random first string
   do
   {
      string2=(int)((nstrings+1)*Math.random()); // pick a random 2nd string
   }
   while (string1==string2);  // check the two strings picked are not identical

// Find the better of the two strings and copy it into the new population

   int better_string=string2;
   if(fitness[string1]>fitness[string2]) better_string=string1;

   for (iangle=0; iangle<nangles; iangle++)
   {
      newstring[newpop][iangle]=string[better_string][iangle];
   }
   newpop++;
}
while (newpop<nstrings-1);  // Keep going until the new population is the right size

// 1-point Crossover

for (istring=0; istring<nstrings; istring++)
{

/* Only do crossover if pcross is greater than a randomly-chosen number */

   if (Math.random()<pcross)
   {
      int string1, string2;
      do
      {
         string1=(int)((nstrings)*Math.random()); // Select a random string
      }
      while(string1>nstrings || (elitism && string1==0));
```

```
      do
      {
         string2=(int)((nstrings)*Math.random()); // and a second random string
      }
      while (string2>nstrings || string1==string2|| (elitism && string2 == 0));

// Choose a point at which to cut the strings

      cut=(int)(1+(nangles-2)*Math.random());

// Swap the segments before the cut point

      for (int i=0; i<cut; i++)
      {
         int a=newstring[string1][i];
         newstring[string1][i]=newstring[string2][i];
         newstring[string2][i]=a;
      }
   }
}

/* Mutate the strings with a probability pmut. If elitism is being applied, do not allow
the first string to be changed. */

int firststring=0;
if (elitism) firststring=1;
for (istring=firststring; istring<nstrings; istring++)
{
   if (Math.random()<pmut)
   {
      int newval=(int)(360.0*Math.random());
      int newangle=(int)(nangles*Math.random());
      newstring[istring][newangle]=newval;
   }
}

// Finally copy the new population over the old one

for (istring=0; istring<nstrings; istring++)
{
   for (iangle=0; iangle<nangles; iangle++)
   {
      string[istring][iangle]=newstring[istring][iangle];
   }
}

generation++;
```