Simple Flocking

This is a partial listing of the Java code for the simple flocking 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

```
/* np is the number of particles in the flock, which can range from 2 to 96 */

np=2;
npproxy=np;

/* Select random Cartesian coordinates for the particles in a box 100 units on a side
and choose random velocities for each particle. We set these for all particles,
including any that are not yet displayed. */

for (i=0; i<96; i++)
{
   x[i]=100.0*Math.random();
   y[i]=100.0*Math.random();
   z[i]=100.0*Math.random();

   vx[i]=(0.5-Math.random());
   vy[i]=(0.5-Math.random());
   vz[i]=(0.5-Math.random());
}

/* trace_length is the length of the tail on each particle. The length must be at least
one, since a length of 0 is interpreted as an unending trace, which will slow the
simulation. */

trace_length=12;

/* nclose is the number of neighbours of the particle with which it can intereact. */

nclose=0;

/* The degree of interaction between particles is proportional to the value of
correlation. */

correlation=4.0;
```

Evolution section

```
/* We start with a fix to prevent the program hanging. This can occur if the slider is
used to change the value of np while the calculations in this section are being
performed. Consequently, the slider actually changes the value of nppropxy rather np.
We reset np if needed at the start of this section. */

if (np != npproxy) np=npproxy;

/* Allow each particle to move. If it goes beyond the edge of the box, bring it back,
and reverse the appropriate component of velocity. */

for (i=0; i<np; i++)
{
   x[i]=x[i]+vx[i];
   if (x[i]>100.0)
   {
      x[i]=100.0;
      vx[i]=-vx[i];
   }
   if (x[i]<0.0)
   {
      x[i]=0.0;
      vx[i]=-vx[i];
   }

   y[i]=y[i]+vy[i];
   if (y[i]>100.0)
   {
      y[i]=100.0;
      vy[i]=-vy[i];
   }
   if (y[i]<0.0)
   {
      y[i]=0.0;
      vy[i]=-vy[i];
   }

   z[i]=z[i]+vz[i];
   if (z[i]>100.0)
   {
      z[i]=100.0;
      vz[i]=-vz[i];
   }
   if (z[i]<0.0)
   {
      z[i]=0.0;
      vz[i]=-vz[i];
   }
}
```

/* Flocking behaviour implies that the particles have some knowledge of each other. We bring this about by identifying for each particle its nearest neighbours and adjusting the particle velocity so that it is more like that of its neighbours. */

/* First, we calculate the distance between all the particles. Since the distance itself is not needed in the calculation, we just calculate the squared distance, to save time. */

```
for (i=0; i<np-1; i++)
{
   for (j=i; j<np; j++)
   {
      d2[i][j]=Math.pow(x[i]-x[j],2.0)+Math.pow(y[i]-y[j],2)+Math.pow(z[i]-z[j],2);
      d2[j][i]=d2[i][j];
   }
}
```

/* Next we find the nclose nearest neighbours. There are various ways this could be done; the method used here is crude but satisfactory. */

```
for (i=0; i<np; i++)
{
   for (k=0; k<nclose; k++)
   {
      double smallestd2=1.0e08;
      int best_neighbour=0;
      for (j=0; j<np; j++)
      {
         if (smallestd2>d2[i][j])
         {
            if (i!=j)
            {
               smallestd2=d2[i][j];
               best_neighbour=j;
            }
         }
      }
      neighbour[i][k]=best_neighbour;
      d2[i][j]=1.0e08;
   }
}
```

/* Now we know which particles are neighbours, blend the velocities. If the velocities are simply averaged, the particles will, over time, virtually come to a standstill, since the average velocity of a large number of particles in any Cartesian direction will be close to zero. Consequently, we add a small random contribution to each velocity.

In this simple model all neighbours are treated identically, so the size of the interaction is not related to how close two particles are. In a more sophisticated

```
approach, the interaction between two particles could be related to their distance
apart. */

double noise=nclose*0.1;
double lowerv=correlation*nclose/100.0;
for (i=0; i<np; i++)
{
   newvx[i]=vx[i]*(1.0-lowerv)+noise*(Math.random()-0.5);
   newvy[i]=vy[i]*(1.0-lowerv)+noise*(Math.random()-0.5);
   newvz[i]=vz[i]*(1.0-lowerv)+noise*(Math.random()-0.5);
   for (k=0; k<nclose; k++)
   {
      j=neighbour[i][k];
      newvx[i]=newvx[i]+correlation*vx[j]/100.0;
      newvy[i]=newvy[i]+correlation*vy[j]/100.0;
      newvz[i]=newvz[i]+correlation*vz[j]/100.0;
   }
}

totalv=0.0;
for (i=0; i<np; i++)
{
   totalv=totalv+Math.abs(newvx[i])+Math.abs(newvy[i])+Math.abs(newvz[i]);
}

/* Copy the new velocities over the old ones and scale them. */

double scale=np/totalv;
for (i=0; i<np; i++)
{
   vx[i]=newvx[i]*scale;
   vy[i]=newvy[i]*scale;
   vz[i]=newvz[i]*scale;
}
```