Computational Physics Assignment 4

Austin F. Oltmanns

February 20, 2018

Abstract

A 3D rendering system is developed and used to test simulation accuracy during large particle collisions. Accuracy of conservation of energy is observed throughout the collision at different system temperatures.

1 Introduction

This assignment required students to expand upon code developed in class for simulating a non-ideal gas. This particular assignment focused on setting up and using a 3D rendering system for 3 dimensional simulations. Additionally, students were tasked with designing and performing their own experiment which utilized this system. The experiment outlined in this report is designed to determine the accuraccy of conservation of energy in the simulation code developed.

The experiment consisted of colliding 2 groups of particles and observing the temperature until the reaction reached a steady state. The groups of particles were set to varying initial velocities (determining the energy/temperature of the system) which caused them to collide. The results from these experiments are included and analyzed in this report.

For simplicity, real world units have been disregarded in an effort to keep the numbers used in the simulation from becoming very large or small (which can affect the stability of the simulation. Each aspect of the simulation will be described in detail below and the code is attached as an appendix to this report.

2 3D Rendering System

2.1 Description

To effectivly visualize the simulation as it happens, it is necessary to be able to view it in 3 dimensions. For this purpose, a rendering system was created which emulates a camera outside of the periodically bound box which contains the simulation. This system takes the particles' coordinates in referance to the box which contains them and performs a coordinate transform to find those same particles coordinates in the coordinate system defined by the camera. Before projecting this image onto a plane, the particles are sorted such that the ones furthest from the camera are drawn first so they do not appear to be on top of the particles that are closer. After this step, the particles are finally projected onto a plane between the camera and the scene. The particles' shadows on this plane are essentially the image that will be viewed. Additionally, the bounding box is rendered with a red dot origin and principle axes are highlighted in color.

To perform these operations, the GNU Scientific Library's wrapper for the Basic Linear Algebra Subprograms (BLAS). This library and its C-language BLAS implementation (CBLAS) allow for matrix and vector math to take place in a C program. Additionally, it would be possible to link the program against different implementations of CBLAS which could be optimized for parralel computation. Meaning that this simulation is now scaleable and would perform well (at least in the rendering routine) on such a platform.

2.2 Coordinate Transform

The world coordinates are defined as points along 3 axes (shown in the Figs. 2-4 as the colored lines). The camera can also be though of as having 3 different axes: one which extends in the direction it points, one which points directly to its side and another which points directly up. Fig. 1 shows the two sets of

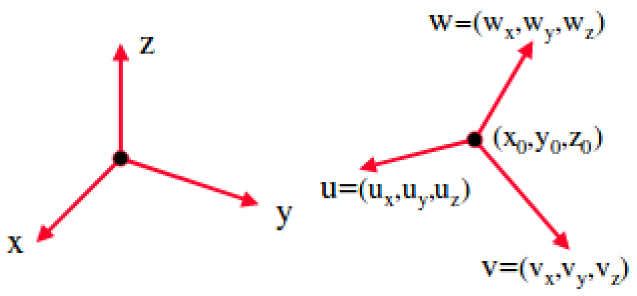

Figure 1: World and camera axes together. Camera axes have been defined with reference to the world axes.

axes. The camera can also be rotated about each of these axes independently in a pan/tilt/roll fashion. By defining the camera's axes as unit vectors from its position in the direction of the axis, a transform matrix can be generated which transforms a point given in world coordinates to the corresponding camera coordinates.

The first step is to determine the translation matrix which would translate the coordinates of a particle in world coordinates to a set of coordinates which would be defined by the camera if it had no rotation relative to the world's coordinate system. This matrix is denoted T in Eqn. 1.

$$T = \begin{bmatrix} 1 & 0 & 0 & X_0 \\ 0 & 1 & 0 & Y_0 \\ 0 & 0 & 1 & Z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

The second step is to find the rotation matrix which would transform the coordinates of a particle in world coordinates to that of the camera if it were located at the origen but rotated. This matrix is denoted R in Eqn. 2.

$$R = \begin{bmatrix} U_x & V_x & W_x & 0 \\ U_y & V_y & W_y & 0 \\ U_z & V_z & W_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

Finally, by combining these operations, the final transformation matrix (denoted M in Eqn. 3) is obtained as the product of the two prior transforms and points/particles described in world coordinates can be described in camera coordniates by applying Eqn. 4.

$$M = RT \tag{3}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = M \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{4}$$

To implement these equations, the following lines of code establish the cameras axis in world coordinates.

```
//rotate by pan and tilt(rotation around Z/W)
gsl_vector_set_basis(renderman.camU, 0);
gsl_vector_set_basis(renderman.camV, 1);
gsl_vector_set_basis(renderman.camW, 2);
gsl_blas_drot(renderman.camU, renderman.camV, cos(mycam.roll), sin(mycam.roll));
gsl_blas_drot(renderman.camU, renderman.camW, cos(mycam.pan), sin(mycam.pan));
gsl_blas_drot(renderman.camV, renderman.camW, cos(mycam.tilt), sin(mycam.tilt));
```

The above section of code instantiates a set of axes for the camera as unit vectors along $\hat{\imath}$, $\hat{\jmath}$ and $\hat{k}$. Then, the roll, pan and tilt of the camera is used to rotate pairs of axes around each respective axis for the rotation. After this, the next snippit of code uses this information to populate the rotation and translation matrices.

```
//form rotation and translation matrices
for (unsigned int i=0; i<3; i++)
{
  gsl_matrix_set(renderman.trans, i, 3, gsl_vector_get(mycam.pose, i));
  gsl_matrix_set(renderman.rot,   i, 0, gsl_vector_get(renderman.camU, i));
  gsl_matrix_set(renderman.rot,   i, 1, gsl_vector_get(renderman.camV, i));
  gsl_matrix_set(renderman.rot,   i, 2, gsl_vector_get(renderman.camW, i));
}
```

Then the total transformation matrix is calculated. The gsl_blas_dgemm function computes the product of two matrices (in this case: renderman.rot and renderman.trans) and sums that product with a third matrix and stores the result in that third matrix.

```
//calculate total transformation matrix
gsl_blas_dgemm(CblasNoTrans, CblasNoTrans,  1, renderman.rot, renderman.trans,
               0, renderman.coordinateTransform);
```

Next, the coordinates can be transformed by another matrix multiplication. The positions of each particle have been loaded into renderman.worldCoordHolder as an array of column vectors to form a 4xN matrix where N denotes the number of particles. Because of how the sort method is constructed, this operation is done to return the transpose of the result by passing the 'CblasTrans' flag for each matrix which is being multiplied and inverting the order of multiplication. This is a property of matrix multiplication and transposes.

```
//perform total transformation
gsl_blas_dgemm(CblasTrans, CblasTrans, 1, renderman.worldCoordHolder,
               renderman.coordinateTransform,  0, renderman.renderCoord);
```

2.3 Sorting by Distance

After the coordinates are transformed but before they are projected onto the imaging plane, they must be sorted by distance from the camera so the furthest particles are drawn first so they appear behind the particles that are closer.

This is accomplished via the C standard library function qsort. Each particle's position in camera coordinates has been recorded in the renderman.renderCoord matrix. Because the color information of each particle must persist, it is necessary to keep track of each particles color before it is sorted. Thankfully, the matrix math from above has left a matrix which has an auxillary 1 below each coordinate. At this point, it will no longer be used to perform matrix math, so it can be replaced with color information before the sort takes place.

The following code demonstrates the procedure:

```
int compare(const void *x1,const void *x2){
  if (((double *) x1)[2]< ((double *)x2)[2]) return 1;
  else return -1;
}

...

//sort rendercoord by camera distance
//assign color
for (unsigned int i=0; i<NUM_PARTICLES; i++)
{
  gsl_matrix_set(renderman.renderCoord, i, 3, i%3 +2);
}
//sort by depth
qsort(gsl_matrix_ptr(renderman.renderCoord,0,0), NUM_PARTICLES,
                     4*sizeof(double), &compare);
```

2.4 Projection onto a Plane

Now that the points can be described in camera coordinates, they must be projected onto a 2D screen. This part is somewhat simpler than the coordinate transform because it simply uses similar triangles for the calculation. A point in camera coordinates will be shown on the plane as having its distance from the origin of the plane (defined as the point orthogonal to the plane and through the camera) scaled by the distance from the plane to the camera divided by the distance of the particle to the camera. The distance from the plane to the camera is the focal length of the camera (mycam.zoom in the below code). This method is known as a perspective projection. Once the points have been projected onto this plane, it is trivial to scale this plane to the rendering area (a window on a computer monitor). The following snippit of code demonstrates this procedure:

```
//display particles
for (unsigned int i=0; i<NUM_PARTICLES; i++)
{
  int xdraw = mycam.zoom * gsl_matrix_get(renderman.renderCoord, i, 0)
              / gsl_matrix_get(renderman.renderCoord, i, 2) + (double)xdim/2.0;
  int ydraw = mycam.zoom * gsl_matrix_get(renderman.renderCoord, i, 1)
              / gsl_matrix_get(renderman.renderCoord, i, 2) + (double)ydim/2.0;
  myfilledcircle((int) gsl_matrix_get(renderman.renderCoord, i, 3), xdraw, ydraw, 3);
}
```

2.5 Bounding Box

Using the same technique that was used to project the points onto the screen, corner points from the experiments periodically bound box are transformed to points in the rendering window and lines are drawn between them, displaying the outline of the bounding box. For brevity, its implementation is not shown here. Instead, Fig. 2 shows the rendered image of particles initialized and ready to collide.

3 Conservation Experiment

3.1 Description

In order to determine how well energy is conserved, two packets of 120 particles each were sent at each other at 8 different speeds corresponding to 8 different initial system temperatures. If energy is conserved well in the simulation, the temperature should remain constant throughout the reaction. Otherwise, the temperature will change throughout the reaction.

3.2 Results

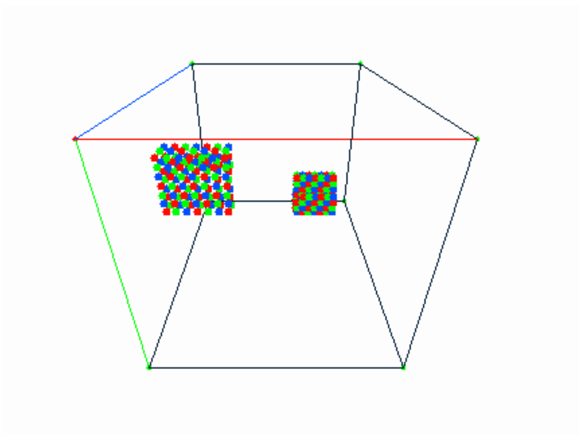

Figure 2: Particles initilaized before being collided.

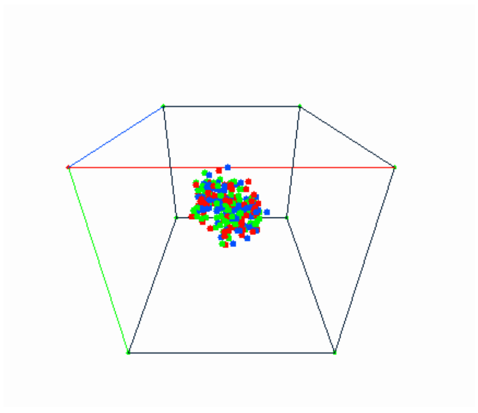

Figure 3: Particles colliding.

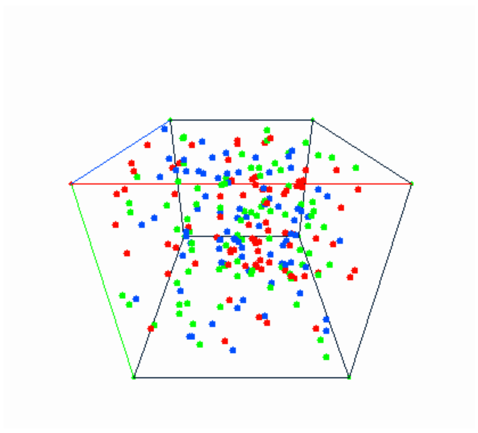

Figure 4: Particles some time after the collision.

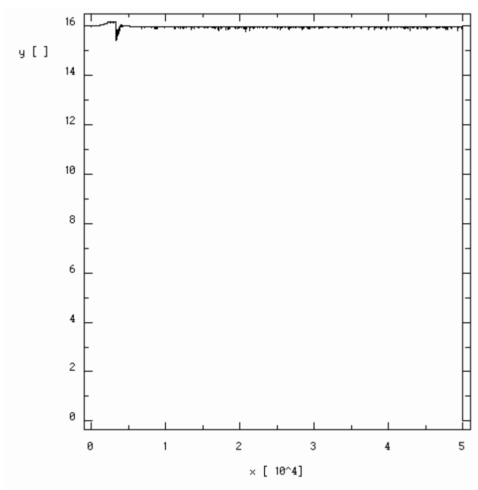

Figure 5: Temperature vs Time in seconds for T_0=16

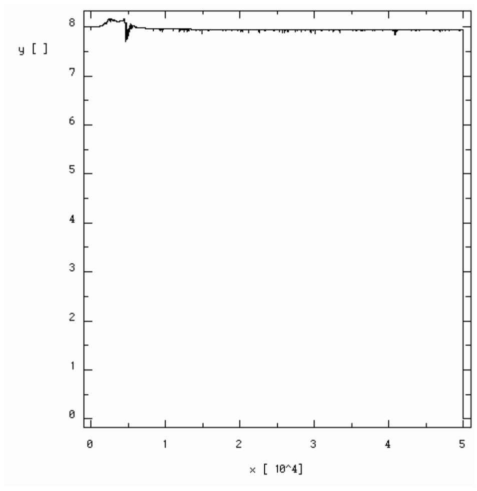

Figure 6: Temperature vs Time in seconds for T_0=8

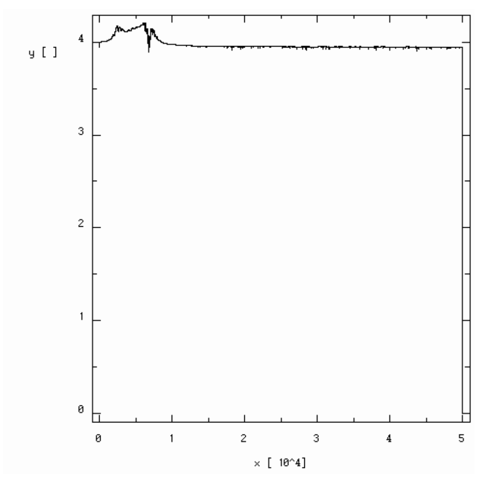

Figure 7: Temperature vs Time in seconds for T_0=4

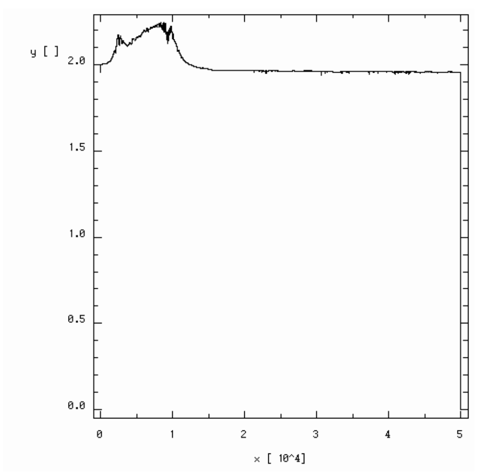

Figure 8: Temperature vs Time in seconds for T_0=2

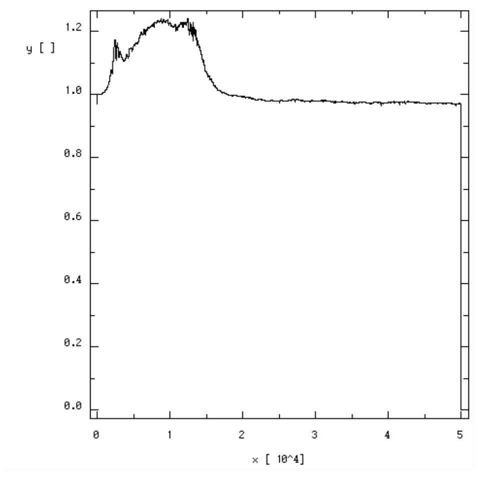

Figure 9: Temperature vs Time in seconds for T_0=1

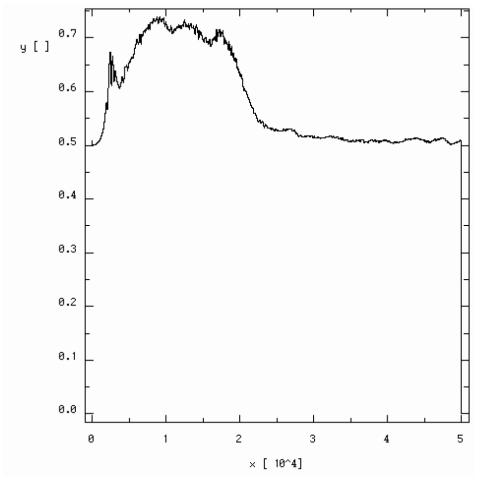

Figure 10: Temperature vs Time in seconds for T_0=0.5

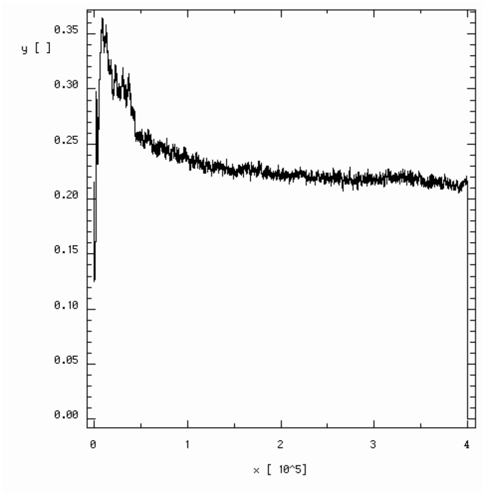

Figure 12: Temperature vs Time in 10's of seconds for T_0=0.125

3.3 Analysis

It can be seen clearly that energy is not conserved well, especially in lower temperature situations. However, in higher temperature situations energy is conserved better. In the lowest temperature test (seen in Fig. 11), the system does not appear to be returning to the initial conditions. Even if it is, it is taking far to long to be useful if other events had to take place in the situation. On the other end of the spectrum, the high temperature simulations return to the initial temperature as the system settles.

4 Conclusion

Obviously there are approximations which take place in a simulation. The question a simulator must ask is how much approximation is acceptable and how much error can be allowed. The data from these experiments show that this module should most likely not be used for low temperature collisions of groups of gas particles. However, at higher temperatures the resuslts could be acceptable depending on application.

Additionally, the 3D rendering system outlined in this report is very modular and could be extended easily to do things like having the camera follow a path or track certain elements throughout the simulation.

The code used to complete this assignment is attached as an appendix to this document.

5 References

http://www.math.tau.ac.il/ dcor/Graphics/cg-slides/geom3d.pdf
http://www.cse.psu.edu/ rtc12/CSE486/lecture12.pdf

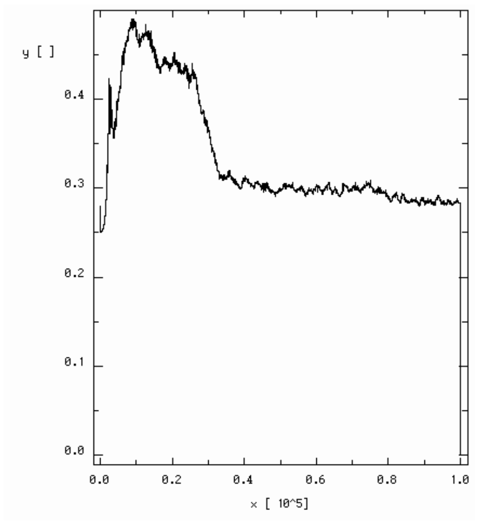

Figure 11: Temperature vs Time in seconds for T_0=0.25

6 Appendix

```
#include <math.h>
#include <mygraph.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <stdio.h>
#include <gsl/gsl_blas.h>
#include <stdlib.h>


#define NUM_PARTICLES 240
#define DIM 3
#define SAMPLELENGTH 400000

double LENGTH=50; //side length of box

double masses[NUM_PARTICLES];
double x[NUM_PARTICLES][DIM];
double v[NUM_PARTICLES][DIM];
double tplot[SAMPLELENGTH] = {0};

double dt =.001;
char buffer[500] = {0};
int firstrun =1;
double setTempVal = 1.5;
double setDensityVal = .05;

double nonIdealPressure;

int points = SAMPLELENGTH;

struct camera_S
{
  gsl_vector * pose;
  double pan;
  double tilt;
  double roll;
  double zoom;
} mycam;

struct renderman_S
{
  gsl_vector * camU;
  gsl_vector * camV;
  gsl_vector * camW;
  gsl_matrix * rot;
  gsl_matrix * trans;
  gsl_matrix * renderCoord;
  gsl_matrix * worldCoordHolder;
  gsl_matrix * coordinateTransform;
  gsl_matrix * boxpoints;
  gsl_matrix * renderBoxpoints;
} renderman;

void calculateForce(double pos[NUM_PARTICLES][DIM], double vel[NUM_PARTICLES][DIM],
                    double force[NUM_PARTICLES][DIM])
```

```
{
  nonIdealPressure =0;
  memset(force,0,NUM_PARTICLES*DIM*sizeof(double));
  for (int n=0; n<NUM_PARTICLES; n++)
  {
    for (int m=n+1; m<NUM_PARTICLES; m++)
    {
      double dr[DIM],dR=0;
      for (int d=0; d<DIM; d++)
      {
            dr[d]=x[m][d]-(x[n][d]-LENGTH);
            double ddr;
            ddr=x[m][d]-(x[n][d]);
            if (fabs(ddr)<fabs(dr[d])) dr[d]=ddr;
            ddr=x[m][d]-(x[n][d]+LENGTH);
            if (fabs(ddr)<fabs(dr[d])) dr[d]=ddr;
            dR+=dr[d]*dr[d];
      }
      double dR6=dR*dR*dR;
      double dR12=dR6*dR6;
      double Fabs=12/dR12-6/dR6;
      nonIdealPressure += Fabs;
      Fabs/=dR;
      for (int d=0;d<DIM; d++){
            force[n][d]-=Fabs*dr[d];
            force[m][d]+=Fabs*dr[d];
      }
    }
  }

  for (int d=0; d<DIM; d++) nonIdealPressure/=LENGTH;

  return;
}

void iterate(double pos[NUM_PARTICLES][DIM],
             double vel[NUM_PARTICLES][DIM], double dt)
{
  double force[NUM_PARTICLES][DIM];
  calculateForce(pos, vel, force);
  if (firstrun == 1)
  {
      for (int n=0; n<NUM_PARTICLES;n++)
    {
      for (int d=0; d<DIM; d++)
        {
      vel[n][d] += .5 * (force[n][d]) / masses[n] * dt;
      }
    }
    firstrun =0;
  }
  else
  {
    for (int n=0; n<NUM_PARTICLES;n++)
    {
        for (int d=0; d<DIM; d++)
        {
          vel[n][d] += (force[n][d]) / masses[n] * dt;
      }
    }
  }
  for (int n=0; n<NUM_PARTICLES;n++)
  {
    for (int d=0; d<DIM; d++)
    {
      pos[n][d] += vel[n][d] * dt;
      while (pos[n][d] < 0) pos[n][d] += LENGTH;
      while (pos[n][d] >= LENGTH) pos[n][d] -= LENGTH;
    }
  }
}

double findTemp()
{
    double temp =0;
    for (int n=0; n<NUM_PARTICLES;n++)
    {
      for (int d=0; d<DIM; d++)
          {
        temp += masses[n] * v[n][d]*v[n][d];
      }
    }
    return temp/NUM_PARTICLES/DIM;
}

double findDensity()
{
    double density =0;
    for (int n=0; n<NUM_PARTICLES;n++)
    {
       density += masses[n];
    }
    density = density / (double)(LENGTH*LENGTH);
    return density;
}

double findPressure(double temp, double density)
{
    return temp*density + nonIdealPressure;
}

void init()
{
  int M=pow(NUM_PARTICLES,1./DIM)+1;
  for (int n=0; n<NUM_PARTICLES; n++)
  {
    masses[n]=1;
    for (int d=0; d<DIM; d++)
    {
      int nn=n;
      for (int dd=0; dd<d; dd++) nn/=M;

      x[n][d]=(nn%M)*LENGTH/(double)M;
      if (d==1)
      {
            if (x[n][0]<LENGTH/2.0)
        {
          v[n][d]=1;
        }
            else
        {
          v[n][d]=-1;
        }
      }
      else
      {
      v[n][d]=0;
      }
    }
  }
  v[0][0] =1;
  firstrun =1;
}

void initCollision()
{
  int M=pow(NUM_PARTICLES,1./DIM)+1;

  for (int n=0; n<NUM_PARTICLES/2; n++)
  {
    masses[n]=1;
    for (int d=0; d<DIM; d++)
    {
      int nn=n;
      for (int dd=0; dd<d; dd++) nn/=M;

      x[n][d]=(nn%M)*LENGTH/(double)M/4.0 + LENGTH/8.0;
      v[n][d] = LENGTH / 30.0;
    }
  }

  for (int n=NUM_PARTICLES/2; n<NUM_PARTICLES; n++)
  {
    masses[n]=1;
    for (int d=0; d<DIM; d++)
    {
      int nn=n;
      for (int dd=0; dd<d; dd++) nn/=M;

      x[n][d]=(nn%M)*LENGTH/(double)M/4.0 + LENGTH/2.0 + LENGTH/8.0;
      v[n][d] = -LENGTH / 30.0;

    }
  }
  firstrun =1;
}

void initDraw()
{
  mycam.pose = gsl_vector_alloc(3);
  gsl_vector_set(mycam.pose, 0, -.5*LENGTH);
  gsl_vector_set(mycam.pose, 1, .3*LENGTH);
  gsl_vector_set(mycam.pose, 2, .6 * LENGTH);
```

```
  mycam.pan = 0;
  mycam.tilt = .4;
  mycam.zoom = 200;
  mycam.roll =0;

  renderman.camU = gsl_vector_alloc(3);
  renderman.camV = gsl_vector_alloc(3);
  renderman.camW = gsl_vector_alloc(3);
  renderman.rot = gsl_matrix_alloc(4,4);
  renderman.trans = gsl_matrix_alloc(4,4);
  renderman.boxpoints = gsl_matrix_alloc(4,8);
  renderman.renderBoxpoints = gsl_matrix_alloc(4,8);

  gsl_matrix_set(renderman.rot, 3,3,1);
  for (unsigned int i=0; i<8; i++)
  {
    gsl_matrix_set(renderman.boxpoints, 3, i, 1);
    gsl_matrix_set(renderman.renderBoxpoints, 3, i, 1);
  }
  for (unsigned int i=0; i<4; i++)
        gsl_matrix_set(renderman.trans, i,i,1);

  renderman.renderCoord = gsl_matrix_alloc(NUM_PARTICLES, 4);
  renderman.worldCoordHolder = gsl_matrix_alloc(4, NUM_PARTICLES);
  renderman.coordinateTransform = gsl_matrix_alloc(4,4);
  gsl_matrix_set_all(renderman.worldCoordHolder, 1);
}

void setTemp()
{
   double currTemp = findTemp();
   double factor = sqrt(setTempVal/currTemp);

  for (int n=0; n<NUM_PARTICLES;n++)
  {
    for (int d=0; d<DIM; d++)
    {
       v[n][d] *= factor;
    }
  }
}

void setDensity()
{
  double factor = 1.0 /LENGTH;
  double currDensity = findDensity();

  LENGTH = pow(NUM_PARTICLES/setDensityVal, 1.0/DIM);
  factor *= LENGTH;
  for (int n=0; n<NUM_PARTICLES;n++)
  {
    for (int d=0; d<DIM; d++)
    {
       x[n][d] *= factor; //assumes life is in the 1st quadrant
    }
  }
}
```

```
int compare(const void *x1,const void *x2){
  if (((double *) x1)[2]< ((double *)x2)[2]) return 1;
  else return -1;
}

void draw3d(int xdim, int ydim)
{
  //rotate by pan and tilt(rotation around Z/W)
  gsl_vector_set_basis(renderman.camU, 0);
  gsl_vector_set_basis(renderman.camV, 1);
  gsl_vector_set_basis(renderman.camW, 2);
  gsl_blas_drot(renderman.camU, renderman.camV, cos(mycam.roll), sin(mycam.roll));
  gsl_blas_drot(renderman.camU, renderman.camW, cos(mycam.pan), sin(mycam.pan));
  gsl_blas_drot(renderman.camV, renderman.camW, cos(mycam.tilt), sin(mycam.tilt));

  //form rotation and translation matrices
  for (unsigned int i=0; i<3; i++)
  {
    gsl_matrix_set(renderman.trans, i, 3, gsl_vector_get(mycam.pose, i));
    gsl_matrix_set(renderman.rot,   i, 0, gsl_vector_get(renderman.camU, i));
    gsl_matrix_set(renderman.rot,   i, 1, gsl_vector_get(renderman.camV, i));
    gsl_matrix_set(renderman.rot,   i, 2, gsl_vector_get(renderman.camW, i));
  }

  //get box vertices ready
  gsl_matrix_set(renderman.boxpoints, 0, 1, LENGTH);
  gsl_matrix_set(renderman.boxpoints, 1, 2, LENGTH);
  gsl_matrix_set(renderman.boxpoints, 2, 3, LENGTH);
  gsl_matrix_set(renderman.boxpoints, 0, 4, LENGTH);
  gsl_matrix_set(renderman.boxpoints, 1, 4, LENGTH);
  gsl_matrix_set(renderman.boxpoints, 1, 5, LENGTH);
  gsl_matrix_set(renderman.boxpoints, 2, 5, LENGTH);
  gsl_matrix_set(renderman.boxpoints, 2, 6, LENGTH);
  gsl_matrix_set(renderman.boxpoints, 0, 6, LENGTH);
  gsl_matrix_set(renderman.boxpoints, 0, 7, LENGTH);
  gsl_matrix_set(renderman.boxpoints, 1, 7, LENGTH);
  gsl_matrix_set(renderman.boxpoints, 2, 7, LENGTH);

  printf("\nrotation matrix: \n");
  for (unsigned int i=0; i<4; i++)
  printf("%f, %f, %f, %f,\n", gsl_matrix_get(renderman.rot, i,0),
                              gsl_matrix_get(renderman.rot, i,1),
                              gsl_matrix_get(renderman.rot, i,2),
                              gsl_matrix_get(renderman.rot, i,3));

  printf("\ntranslation matrix: \n");
  for (unsigned int i=0; i<4; i++)
  printf("%f, %f, %f, %f,\n", gsl_matrix_get(renderman.trans, i,0),
                              gsl_matrix_get(renderman.trans, i,1),
                              gsl_matrix_get(renderman.trans, i,2),
                              gsl_matrix_get(renderman.trans, i,3));


  //cast pose of each particle to our matrix type via nasty copy
  for (unsigned int i=0; i<NUM_PARTICLES; i++)
  {
    for (unsigned int d=0; d<DIM;d++)
    {
      gsl_matrix_set(renderman.worldCoordHolder, d, i, x[i][d]);
    }
  }

  //calculate total transformation matrix
  gsl_blas_dgemm(CblasNoTrans, CblasNoTrans,  1, renderman.rot, renderman.trans,
                 0, renderman.coordinateTransform);

  printf("\ntransform matrix: \n");
  for (unsigned int i=0; i<4; i++)
  printf("%f, %f, %f, %f,\n", gsl_matrix_get(renderman.coordinateTransform, i,0),
                              gsl_matrix_get(renderman.coordinateTransform, i,1),
                              gsl_matrix_get(renderman.coordinateTransform, i,2),
                              gsl_matrix_get(renderman.coordinateTransform, i,3));

  //perform total transformation
  gsl_blas_dgemm(CblasTrans, CblasTrans, 1, renderman.worldCoordHolder,
                 renderman.coordinateTransform,  0, renderman.renderCoord);
  //transform box points
  gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 1, renderman.coordinateTransform,
                 renderman.boxpoints, 0, renderman.renderBoxpoints);


  //then project particles onto image plane via
  //ximage = f*xcam/zcam
  //yimage = f*ycam/zcam
  //where f is focal length
  //pick f (and camera location) so entire bounded box is displayed comfortably
  //draw box vertices
  int screenvertices[2][8];
  for (unsigned int i=0; i<8; i++)
  {
    int xdraw = mycam.zoom * gsl_matrix_get(renderman.renderBoxpoints, 0, i)
                / gsl_matrix_get(renderman.renderBoxpoints, 2, i) + (double)xdim/2.0;
    int ydraw = mycam.zoom * gsl_matrix_get(renderman.renderBoxpoints, 1, i)
                / gsl_matrix_get(renderman.renderBoxpoints, 2, i) + (double)ydim/2.0;
    myfilledcircle(2 + ((i == 0) ? 0 : 1), xdraw, ydraw, 2);
    screenvertices[0][i] = xdraw;
    screenvertices[1][i] = ydraw;
  }
  //draw bounding lines
  mydrawline(2, screenvertices[0][0], screenvertices[1][0],
                screenvertices[0][1], screenvertices[1][1]); //red
  mydrawline(3, screenvertices[0][0], screenvertices[1][0],
                screenvertices[0][2], screenvertices[1][2]); //green
  mydrawline(4, screenvertices[0][0], screenvertices[1][0],
                screenvertices[0][3], screenvertices[1][3]); //blue

  mydrawline(6, screenvertices[0][4], screenvertices[1][4],
                screenvertices[0][1], screenvertices[1][1]);
  mydrawline(6, screenvertices[0][4], screenvertices[1][4],
                screenvertices[0][2], screenvertices[1][2]);
  mydrawline(6, screenvertices[0][4], screenvertices[1][4],
                screenvertices[0][7], screenvertices[1][7]);

  mydrawline(6, screenvertices[0][5], screenvertices[1][5],
                screenvertices[0][7], screenvertices[1][7]);
  mydrawline(6, screenvertices[0][5], screenvertices[1][5],
                screenvertices[0][2], screenvertices[1][2]);
  mydrawline(6, screenvertices[0][5], screenvertices[1][5],
                screenvertices[0][3], screenvertices[1][3]);

  mydrawline(6, screenvertices[0][6], screenvertices[1][6],
                screenvertices[0][7], screenvertices[1][7]);
  mydrawline(6, screenvertices[0][6], screenvertices[1][6],
                screenvertices[0][3], screenvertices[1][3]);
  mydrawline(6, screenvertices[0][6], screenvertices[1][6],
                screenvertices[0][1], screenvertices[1][1]);

  //sort rendercoord by camera distance
  //assign color
  for (unsigned int i=0; i<NUM_PARTICLES; i++)
  {
    gsl_matrix_set(renderman.renderCoord, i, 3, i%3 +2);
  }
  //sort by depth
  qsort(gsl_matrix_ptr(renderman.renderCoord,0,0), NUM_PARTICLES,
        4*sizeof(double), &compare);

  //display particles
  for (unsigned int i=0; i<NUM_PARTICLES; i++)
  {
    int xdraw = mycam.zoom * gsl_matrix_get(renderman.renderCoord, i, 0)
              / gsl_matrix_get(renderman.renderCoord, i, 2) + (double)xdim/2.0;
    int ydraw = mycam.zoom * gsl_matrix_get(renderman.renderCoord, i, 1)
              / gsl_matrix_get(renderman.renderCoord, i, 2) + (double)ydim/2.0;
    myfilledcircle((int) gsl_matrix_get(renderman.renderCoord, i, 3), xdraw, ydraw, 3);

    printf("\nworld coord: \n");
    printf("%f, %f, %f\n", x[i][0],x[i][1],x[i][2]);
    printf("\nworld coord (in holder): \n");
    printf("%f, %f, %f, %f\n", gsl_matrix_get(renderman.worldCoordHolder, 0, i),
                               gsl_matrix_get(renderman.worldCoordHolder, 1, i),
                               gsl_matrix_get(renderman.worldCoordHolder, 2, i),
                               gsl_matrix_get(renderman.worldCoordHolder, 3, i));
    printf("\ncamera coord: \n");
    printf("%f, %f, %f, %f\n", gsl_matrix_get(renderman.renderCoord, i, 0),
                               gsl_matrix_get(renderman.renderCoord, i, 1),
                               gsl_matrix_get(renderman.renderCoord, i, 2),
                               gsl_matrix_get(renderman.renderCoord, i, 3));
    printf("\ndraw coord: \n");
    printf("%d, %d\n", xdraw, ydraw);
  }
}

int main(){
  struct timespec ts={0,100};
  int cont=0;
  int sstep=0;
  int done=0;
  int repeat=100;

  double mytemp =0;
  unsigned int tempindex =0;
```

```
initDraw();

init();

AddFreedraw("Particles",&draw3d);
StartMenu("Newton",1);
DefineDouble("dt",&dt);
DefineDouble("zoom", &(mycam.zoom));
DefineDouble("pan", &(mycam.pan));
DefineDouble("tilt", &(mycam.tilt));
DefineDouble("roll", &(mycam.roll));

DefineDouble("camX", gsl_vector_ptr(mycam.pose, 0));
DefineDouble("camY", gsl_vector_ptr(mycam.pose, 1));
DefineDouble("camZ", gsl_vector_ptr(mycam.pose, 2));
//DefineDouble("k",&k);
DefineGraphN_R("Temp_vs_Time",&tplot[0],&points,NULL);

StartMenu("init_menu",0);
for (int n=0; n<NUM_PARTICLES; n++)
{
  //DefineDouble("x0", &x0[n][0]);
  //DefineDouble("y0", &x0[n][1]);
  //DefineDouble("vel x0", &v0[n][0]);
  //DefineDouble("vel y0", &v0[n][1]);
  //DefineDouble("charge",&charges[n]);
  DefineDouble("mass"  ,&masses[n]);
}
DefineFunction("Do_init",&init);
EndMenu();
DefineFunction("Collision_Init", &initCollision);
DefineDouble("Curr_Temp", &mytemp);
DefineDouble("Set_Temp_val", &setTempVal);
DefineFunction("do_Set_Temp", &setTemp);
DefineGraph(curve2d_, "Time_vs_Temp");

DefineDouble("Set_Density_val", &setDensityVal);
DefineFunction("do_Set_Density", &setDensity);

DefineGraph(freedraw_,"graph");
DefineInt("num_steps",&repeat);
DefineBool("step",&sstep);
DefineLong("NS_slow",&ts.tv_nsec);
DefineBool("cont",&cont);
DefineBool("done",&done);
EndMenu();
while (!done){
  Events(1);
  DrawGraphs();
  if (cont||sstep){
    sstep=0;
    for (int i=0; i<repeat; i++)
    {
      iterate(x,v,dt);
      mytemp = findTemp();
      tplot[tempindex] =mytemp;
      if (++tempindex >= SAMPLELENGTH-1) {tempindex =0; cont =0;}
```

```
            //measure();
        }
        if (cont) nanosleep(&ts,NULL);
      }
      else sleep(1);
  }
}
```