Simulation of Planetary Formation by Particle Accretion

New Mexico
Supercomputing Challenge
Final Report
April 4, 2006

Team 47
Las Cruces High School

Team Members
Jonathan Lee Annua
Billy Downs
Vincent Mestas
Andrei Klypin

Teacher
Gregory Marez

Contents

Executive Summary	3
Introduction	4
Description and Methodology	4
Model	4
Program Structure	5
Selected Programming Challenges	6
Results and Conclusions	7
Bibliography	10
Acknowledgments	10
Appendix A: Program Scripts	11
Appendix B: Source Code	15

Executive Summary

Our project, Simulation of Planetary Accretion, endeavored to answer how and in what distributions extra-solar planets form, in order to aid astronomers in detecting such planets in real life in a time- and money- efficient manner.

Written with the C language, our core program models the interaction of particles in three-dimensional space, tracking their individual positions, velocities, masses, and when incident, collisions. A host of scripts and other programs allowed us to process and visualize the data (see Appendix A). The main mathematical model of the project involves Newton's laws of dynamics and gravitation, and is essentially based upon a large loop calculating the variables of the particles; the brunt of the difficulty in the program consists of ensuring initializations occur properly, and that the program runs as efficiently (quickly) as possible due to the high number of repetitions involved. Furthermore, the initial values must be carefully calibrated to faithfully reproduce natural conditions. Thus, the results of our work are initially determined by concurrence with observed and predicted conditions (1).

Owing to that the implementation of the program was achieved only late in the academic year, only small simulations were run, aimed at locating initial conditions of an accretion disk capable of producing stable distributions of particles (Results, Table 1).

(1) Chambers, J.E. 2004. Planetary Accretion in the Inner Solar System. Earth and Planetary Science Letters, 223, 241-252, gives approximate timescales for the formation of planetesimals, planetary embryos, and planets.

Introduction

In recent years, as telescope and other potentially planet-detecting technologies have been improving in resolution, quantity, and cost-effectiveness, the search for extra-solar planets, particularly those similar to Earth that may harbor life, has been getting increased attention. In

order to aid in this search, we are creating a model for the formation of planets from solar accretion nebulae, clouds of dust left over from the formation of stars.

The accretion theory is the accepted model for the formation of planets and solar systems; it holds that particles, interacting chiefly under the force of their own gravity, collide, stick, and combine, first into mountain-sized planetesimals, later into Mars- and Moon- sized planetary embryos, and finally into terrestrial and Jovian (gaseous) planets. (1) Such models give astronomers important information as to where to search in the galaxy for Earth-like planets, greatly increasing the rate and decreasing the cost of detection.

Description/Methodology

The mathematical model for our program is based chiefly on Newton's laws of dynamics and gravitation; its principal function is to track the positions, velocities, masses, and attractions of a cloud of particles around a star, combining the particles when necessary. This is a simplified model of particle interactions as it does not take into account thermodynamic factors, an approach which treats the accretion disk as a type of fluid (1). Moreover, due to the nature of the methods used in the particle-particle n-body simulation, the amount of calculations increases quadratically with each new particle, limiting the number of particles that can be practically monitored.

The specific application of this program is to test the approximate outcomes of varied initial conditions such as the size of the star, velocity of angular rotation, and the total mass in the system; as such, this project aims to show the probabilities of planets forming given the known or inferred characteristics of a system.

Model

The equation $F = G (m_i m_j) / r^2$ gives the force of one particle *i* acting on particle *j,* where m represents mass, r is the distance between two particles, and G is the gravitational constant (approx. 6.673×10^{-11}.) Since F = ma, where a is acceleration, the equation reduces to $a = G (m_j) / r^2$, which gives a way to determine the acceleration of a particle caused by another. The chief function in our program (gravitate()) involves a loop which summates the acceleration of one particle from all the others, then repeats the calculation for each particle.

Once the acceleration is calculated, kinematics gives the new position of the particles; the velocity of the particle is given by $v_f = v_i + a * \Delta t$. The subsequent position is given by $x_f = x_i + v * \Delta t$. The use of two separate formulas instead of the more direct $x_f = x_i + v * \Delta t + 0.5a * \Delta t^2$ is called the leapfrog method; it allows for the tracking of velocity between timesteps and is therefore more accurate in the long run.

Program Structure

(the complete program is in Appendix B for reference)

Steps in the Program:
(Note: Declarations and function prototypes are omitted)
(Note: This description applies only to the Static branch of the program which does not

employ malloc)

1. input() is called and passed the maximum/initial number or particles.

2. input() reads the file input0.txt, recording the number of iterations, frequency between outputs, and the length of the timesteps.

3. input() reads the flags of the particles; a flag is the first number in each line containing information about the particle(s) to initialize and tells input() which values to expect and actions to take.

4. input() assigns the values found in input0.txt to the corresponding variables based on the flag, or creates other values using the input values as seeds/parameters.

5. output() writes the initial coordinates of the particles to ouput0.txt

6. gravitate(), the main method is called. It is set up with four nested loops (presented from the inside outwards)-

a. Loop k runs each particle that particle i is checked against--if index k is not equal to i, the acceleration of i due to k is calculated and added to a running total.

b. Loop j runs through the x, y, and z components of each particle; once loop k within it finishes calculating the acceleration, j calculates the velocity and change in position of particle i based on the acceleration, old velocity, and timestep. Acceleration is then reset and the next component is calculated.

c. Loop i runs through each particle to allow the nested loops to calculate the accelerations, velocities, and position changes in each particle.

d. Loop c runs loop i and contents, then checks if c is a multiple of the frequency. If it is, it calls output() to write the positions of the particles. Afterwards, update() is called to change the positions of the particles and combine() is called to check if any particles have collided.

7. output() first checks if the iteration is the multiple of some factor. If it is (meaning the output file has reached a certain size), it begins a new file. It then outputs the iteration number (c), and the indecies and all coordinates of the particles onto a line.

8. update() adds the change in position (delta) of each component of each particle to the old position of each component of each particle; it is basically two nested loops and an assignment statement.

9. combine() contains a two-loop structure that checks each particle against each other particle (similar to loops i and k in gravitate()), and if the radii of the two particles (determined by the mass) are larger than the distance between the two particles (i and j), the position, velocity, and mass of the first particle (i) is updated, and the shift() is called with the index of the second particle (j).

10. shift() then copies the contents of each particle after j into the particle before j, with the net result of shifting each particle above j down one spot, getting rid of it. The counter for the number of particles is then decremented so the program skips all the remnant values of particles which should no longer be in the simulation.

Selected Program Challenges

Though the mathematical model is simple in concept, several challenges arise in creating and manipulating the particles. The ideal model for the initialization of an accretion disk entails an even distribution of particles with velocities perpendicular to the position vector coming from the sun to the coordinates of the point. For simplicity, this discussion will focus on a sphere; in order to create a flattened disk, the z coordinates of all points are multiplied by some factor $0 < f$

< 1.

However, choosing based on random distance and directions does not yield an even distribution. While a random number will evenly distribute points in one dimension, if the line is rotated around an endpoint, forming a circle, the outer sections (of arbitrary length) of the line sweep out larger areas than inner sections; thus, when projected onto a circle, a randomly generated line of points with equal point distribution per unit length will have more points per unit area at the center of the circle than at the outside.

Moreover, the sine and cosine functions do not sweep out equal sectors of a circle in equal increments of angle A. For example, the sine of 30 is .5. The sine of 60 is .833, and the sine of 90 is 1. Thus, the first 30 degrees contains (projecting to a line of length 1, since we are discussing one-dimensional components of a vector) .5 the length of the line, the second 30% contains .833-.5 = .333 of the line, and the final 30 degrees contains .267 of the line; thus, if an angles at random are chosen from 0 to 90 and their vertical (sine) components are projected on a line, and if these projections correspond to the y coordinates of a point, points will be more densely spaced towards the end of the line (where a larger change in angle corresponds to a smaller change in position).

Combined and projected into a sphere (with the addition of a second random angle), the random radius-direction method yields a cloud of particles concentrated towards the center and the poles of the sphere, in pronounced cases creating a bar-like structure within a thin sphere. While the radius bias is easy to overcome both logically and computationally, the angle bias requires the use of inverse trigonometric functions, taking a comparatively long time. Dr. Klypin helped in this regard, suggesting the program create coordinates directly by their components, generating x, y, and z positions from -.5 to .5, the net result being an evenly distributed cube. As the positions were generated, the program would check them against the volume of a sphere of radius .5; if the particle fell within the sphere it would be kept, while others would be discarded. This led to a roughly 2 to 1 discard rate, but was still faster than using weighted radius and directions.

The velocities were more tricky. According to basic vector algebra, two vectors **A** and **B** are perpendicular if **A**x * **B**x + **A**y * **B**y = 0 Thus, to make the velocity vector **B** perpendicular to position **A**, **B** must have x and y coordinates, respectively, 1/**A**x and - 1/**A**y. However, this yields two problems; when either component of A is less than 1, the B component begins growing at an exponential rate, yielding values orders of magnitude higher than the median velocity. When the average radius becomes very high, though, **B** dwindles to 0, requiring absurdly large Maximum Velocity factors with which to multiply them or else run simulations with essentially unmoving particles, a very inaccurate model.

After trying numerous math and computation-intensive methods to normalize the velocities as they were generated, the team settled on imposing a minimum position component of 1 (not a significant source of error for simulations spanning 11 orders of magnitude) and dividing each component of the velocity by the root of the sum of the squares of both components. This gave values of **B**x and **B**y proportional to their unadjusted values (necessary to keep the velocity perpendicular to the position), and expressed their magnitudes in terms of their relation to the other component. Afterwards, the adjusted velocity components were multiplied by (maxPosition - abs(componentPosition)) / (maxPosition), which causes the velocity to become larger the closer is to the center of the system (this is seen in planetary orbits; Mercury is fastest and Pluto is slowest). Finally, this value (between 0 and a little over 1) was multiplied by the maximum position and adjusted for direction. The adjustment of the velocity allowed for

easy and predictable control over the rotational velocity of the disk, allowing it to be simulated and affected directly.

Long before we had made the combination algorithm, we stumbled upon the major difficulty in executing our program: the nature of the computations required an exponential amount of calculations, meaning running twice as many particles increased the time required by roughly fourfold. In order to save as much time as possible, the combine() algorithm employed two fairly simple shortcuts. First, the ratio of the two particles being tested (mass[i] / mass[j]), which was necessary to determine the new coordinates, was pre-calculated and stored to save the time of dividing out a constant factor several times per combination operation.

The other trick was to relate mass to the radius of a particle by a constant (if two particles' physical radii are found to be larger than their separation, they have collided and are assumed to have combined). We assumed the particles were spherical and made of iron (the most abundant element in planetary cores), substituted mass times density for volume, and solved for the radius of a sphere with given mass; in the end, we managed to pull out a constant related to the cubed root of density and 4/3π, approximating it to .032. Thereafter, radius was assumed to be .032 * mass^ (1/3), a much more practical method than passing a cubed root operation countless times, considering we did not track density in our program.

Results and Conclusions

The following images represent exploratory/verification executions of our program and are designed to give an introduction to the representation of our results and a visualization of the principles of our program.

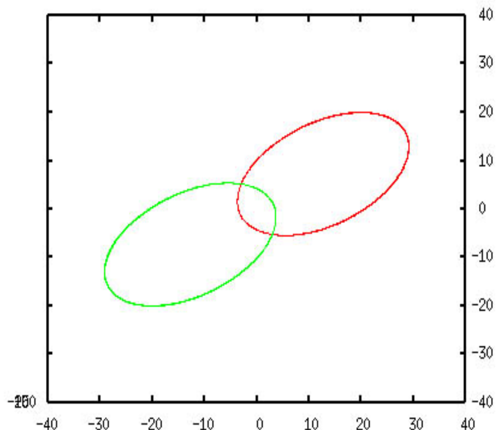
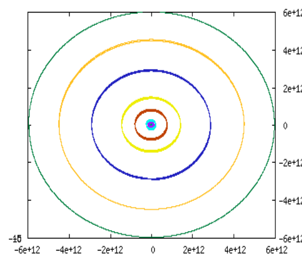


Fig 1 This is the most basic configuration of particles, with its chief application being to test the accuracy of modified algorithms; it includes two particles in the xy plane, one at (20,20) and the other at (-20,-20), both with masses 10^{10}kg. They are both given velocities of .05 in opposing directions, and have periods of rotation (at which point they return to their original positions) of approximately 1170 iterations with a timestep of 1.

Fig 2 This image represents the nine planets of the solar system orbiting the sun for 250 earth years; the configuration is not true-to-life (Pluto's orbit is actually highly elliptical), but holds the planets in ideally stable orbits based on their astronomical data. The purpose of this simulation is to test the accuracy and the efficiency (amount of calculations per unit time) of the program, as ten particles necessitates numerous computations per



particle even for a small number of iterations.

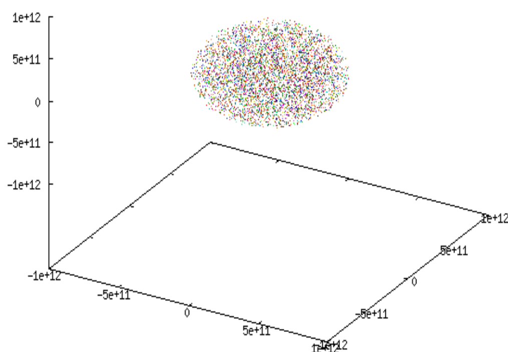


Fig 3 This is a cloud of 1000 randomly generated particles in a sphere. The approximate even distribution of these particles is crucial to the integrity of the calculations (see Methodology for a more thorough discussion); in uneven distributions, biases of particles towards the center of the poles of the sphere are clearly visible.

Due to late implementations of the proper initialization of velocities, necessary for simulations attempting to replicate an accretion disk, we were only able to run very rough simulations over very short periods, aimed at selecting data paths which to follow in later simulations. The values in these simulations are highly arbitrary; the mass of the central star (in most cases 10^{30} kg) is near that of the sun. The maximum radius of the particles ranges from Venetian to asteroid belt-distances, with the understanding particles, subject to the gravitational pull of those on the outside, will experience an outward force to counter the centripetal net force needed to orbit the sun, causing them to move outwards (we encountered this in several of the simulations). The velocity of the particles was the most variable part of the simulation, ranging two orders of magnitude due to uncertainty in the behaviors of the particles. The masses sit between planetesimals (mountain-sized) and planetary embryos (moon- or Mars-sized), and were chosen in order to affect a more pronounced interaction between the particles and the sun.

The reason for the arbitrary nature of these values is twofold--in the immediate sense, larger mass values allow for the simulation mimic a more mature stage in the solar system, allowing for extrapolations both forwards and backwards in the simulation. In a more distant view, these values represent a baseline by which to compare other sets of parameters; further simulations of the program will be performed based on which sets of parameters appear to generate stable distributions of particles.

All of these simulations were run with 749 particles and a sun particle (manually initialized), making for a total of 750 particles. A total of 100000 iterations were performed at 60 seconds each, representing approximately 69 days. The numbers remaining (the last two columns) are the number of non-sun particles left after those iterations.

Table 1

Simulation	Mass of Sun	Radius	Velocity	Mass	# after 35yrs	# after 69 yrs
1	1e30	1e11	1e3	1e20	208	21
2	1e30	1e11	1e4	1e19	707	707
3	1e30	5e11	1e3	1e19	731	689
4	5e29	1e11	1e3	1e19	410	76
5	1e30	1e11	1e3	1e19	184	21
6	1e30	5e11	5e2	1e19	725	683
7	1e30	1e11	5e2	1e20	190	14

The scope of this simulation, spanning days (Chambers (1) indicates a period of .1 to 1 million years for planetesimals to form into planetary embryos, with planets only appearing 10-100 million years after), is imperfect, especially given the possible stochastic variation in a random simulation involving only 750 members. However, based on the data, simulations 2, 3, and 6 merit further investigation, with over 90% of the members surviving the full period (note: on simulation 2 many of the radii are increasing, indicating velocity was too high to form a stable system). The simulations containing either greater radius or greater velocity appear to be more stable (despite a difference in velocity of a factor of 2, the results of 3 and 6 are for all practical purposes the same), indicating the accretion disk likely encompassed a larger radius and faster rotation.

The next step, after running several more simulations with characteristics similar to 2, 3, and 6, is to check whether or not the particles eventually combine with each other (heretofore they only interacted with the sun), meaning the conditions not only support the stable existence of particles but also foster their interactions.

Selected Bibliography

(1) Chambers, J.E. 2004. Planetary Accretion in the Inner Solar System. Earth and Planetary Science Letters, 223, 241-252.

(2) Planetary Systems: Formation, Evolution, and Detection. Weidenshilling, Stuart J. Science, Jan 20, 1995 v267 n5196 p395(2).

(3) http://www.amara.com/papers/nbody.html

Acknowledgments

We would like to extend a special thank you to Dr. Anatoly Klypin of New Mexico State University who provided us with useful tips on how to get our program off the ground, and how to do it well. Credit for the leapfrog method, the sphere distribution solution, and several other important contributions. He's done more to make our program faster than the ICC compiler.

Appendix A

Guide to Planetary Accretion Project Files

Files:

clean
doit
inpgrp.txt
input0.txt
main.c

packit
plotit
saveit
workit
writeit.c

During the course of running the project, the following files are created:

graphitit
grp
output***.txt
Particles
writeit

Project depends on:
nedit
gnuplot 4.0.0 (3.7 works with limited functionality)
icc (gcc can be used also)

File Documentation

clean:

usage: ./clean
description: Removes output files between executions.

doit:

usage: ./doit
description: Cleans the package, compiles main.c, runs the project, creates a grapher script, and graphs the output, then removes the output script. Use this to execute the project. If gcc is being used, open doit and change "icc" to "gcc".

inpgrp.txt:

description: This is the file containing the parameters for the graphing scripts. Not having this file will cause a segmentation fault.
format: double pointsize, double xRange, double yRange, double zRange, int numParticles

input0.txt:

description: This file initializes the particle and environment variables for Particles. Not having this file will cause a segmentation fault.

format: The first three values are:
int numIterations, int pointFrequency, double timeStep

Ye olde methodes by wheech particles are definede are of the followeing fifve:

The methods are referred to as "flags"--each format below contains the flag and its attributes; all values are double unless otherwise noted, directions are in degrees:

Formats have been formatted for ease of reading; when creating the file, values may be delimited by any combination of spaces, tabs, and carriage returns, but no other characters.

Corresponds to MP:
1 xPosition, yPosition, zPosition,
xVelocity, yVelocity, zVelocity,
mass

Corresponds to MPD:
2 xPosition, yPosition, zPosition,
velocityMagnitude, vDirectionXY, vDirectionXZ,
magnitude

Corresponds to MPDA:
3 positionMagnitude, pDirectionXY, pDirectionXZ,
velocityMagnitude, vDirectionXY, vDirectionXZ
mass

Random initiation:
6 numParticles, maxRadius, minRadius, maxVelocity, maxMass, minMass

at present minRadius has no effect

main.c:

description: The main program. See source (Appednix B) for additional comments. When changing the number or particles in a simulation, both MAXPARTICLES in main.c and the final value in inpgrp.txt must be equal. Additionally, be sure the number of particles specified in input0.txt agrees with these values. Not doing so could cause an incorrect number of particles to be displayed or a run-time error. (Note: doit outputs the compiled program as "Particles")

packit:

usage: ./packit <target directory>
description: Purges output files and executables from the package, tars the package, and saves the tar to the target directory.

plotit:

usage: ./plotit
description: complies writeit, creates the graphing script grp, then plots the data. use this instead of doit to view an existing output file.

saveit:

usage: ./saveit <name> <target directory>
description: Saves the input and output files to a tar archive and copies it to the specified location. Use this if you do not want to lose your data upon running packit.

workit:

usage: ./workit
description: Compiles the graph-script maker and opens a nedit workspace with main.c, output0.txt, input0.txt, and inpgrp.txt. This should be the first script run upon successful un-tar-ing of the package. If gcc is being used, open workit and change "icc" to "gcc".

writeit.c:

description: The source for the graph-script maker program. Creates the file run by doit to make gnuplot plot the correct number of points (this was formerly done by hand but required a prohibitive about of time when working with more than a few particles). Takes input from inpgrp.txt and creates the file grp.

Files created by project:

graphitit:

usage: gnuplot graphitit
description: This file is created and removed by every call of grp, and does not need to be accessed unless there is an error in executing gnuplot.

grp:

usage: ./grp
description: This file is automatically created by writeit and is removed by doit upon successful execution. If the user wants to replot a set of data, ./writeit ; ./grp should be executed.

output0.txt:

description: Contains the iteration number, point numbers, and point coordinates (x,y,z format) outputted by Particles. This file is read by gnuplot, so do not modify it. ouput0.txt is deleted and recreated every time doit executes, so it should be renamed if the user wants to save the data for later plotting or analysis.

Particles

usage: ./Particles
description: The executable of main.c. Requires input0.txt to read data. Particles is recompiled every time doit is called and deleted by packit at the end of every session. If the user wishes to not recompile the program in order to graph, use the following: ./Particles ; ./workit; ./grp

writeit

usage: ./writeit
description: The program that reads inpgrp.txt to create the graphing script grp. It is compiled by workit at the beginning of each worksession, this executable is deleted by packit at the end of every session.

Appendix B

Source Code for main.c

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#define GRAVITY 6.673e-11
#define SOFTENER 1e-10
#define MAXPARTICLES 750
#define pi 3.14159265358979323

//random number between 0 and 1
```

```
double nRand();

//take input from keyboard or file
void input(int quant);
void inpMP(int i);
void inpMPD(int i);
void inpMPDA(int i);

//working with arrays
void shift(int index);

//manipulating the particles
void gravitate();
void combine();
void update();
void output();


//Make Particle

//Position and velocity coordinates
void mp(int pNum,
   double xp,   double yp,   double zp,
   double xv,   double yv,   double zv,
   double pMass);

//Position coordinates, velocity vector
void mpd(int pNum,
   double xp,   double yp,   double zp,
   double mv,   double dxyv,   double dxzv,
   double pMass);

//Position and velocity vectors
void mpda(int pNum,
   double mp,   double dxyp,   double dxzp,
   double mv,   double dxyv,   double dxzv,
   double pMass);

    FILE *out, *in;
    double position[3][MAXPARTICLES], velocity[3][MAXPARTICLES];
    double mass[MAXPARTICLES];
    double delta[3][MAXPARTICLES]; //stores the change in position until
all particles have been computed
  double timeStep;
  int numParticles = MAXPARTICLES, freq;
  unsigned long int iterations;
  int i, j, k;
  unsigned long int c;
  int segment;
  char filename[13];
```

```
int main (void)
{
 //initialization of files and random seed
  srand((unsigned)time(NULL));
  out = fopen("output0.txt","w");
  if ((out = fopen( "output0.txt","a")) == NULL)
   {
    printf("Can't open FILE\n");
   }

  in = fopen("input0.txt","r");

  input(MAXPARTICLES);

  segment = (int)(freq * ( 1.2e5 / numParticles));
  printf("%d\n", segment);
  output();

  gravitate();

  return(0);

}
 //returns a double from (almost) 0 to 1
 double nRand()
  {
   unsigned int randN;
   double retRand;
    randN = rand() % 65000 + 1;
    retRand = randN / 65001.0;
    return retRand;
  }

 /*
  Note on Nomenclature: Variable names ending with p or P refer to
position EXCEPT for numP, which refers to the Number of Particles to
make under flag 5; v or V refer to velocity; and M to mass. Variable
names of form dx** refer to the direction either on the xy or xz axes
of either position or velocity. mv and mp refer to the magnitudes of
velocity and position.

  Note on coordinate systems: By default, I named the two-axis system
xy. That is, when looking at a two-dimensional plot in gnuplot, one
sees the x and y axis. However, adding a third z dimension causes an
apparent error in nomenclature, as from the perspective of gnuplot,
the z axis represents the height (that is, looking from above at a
three-dimensional image will yield a projection along the x-y axis.)
In traditional coordinate systems, x and z form the "horizontal
plane", while this program uses x and y for said plane. Thus, in order
to have a two-dimensional image, z calculations must be excluded
because z is the last axis calculated.
        z
```

```
        !
        !       Program's coordinate axes
        !___x
       /
      /
     y

       y
       !
       !        Traditional coordinate axes
       !___x
      /
     /
    z

  */


 void input(int quant)  //reads particle data and initialization type
from a file
 {
  double iterDummy;  /*although iterations is an int, reading it as a
double allows exponential notation (1e6)*/
  fscanf(in, "%lf%d%lf", &iterDummy, &freq, &timeStep);
  iterations = (unsigned long int)iterDummy;


  double maxP, maxV, minP, minV, maxM, minM;  //these values are used on
flag 6
   for (i = 0; i < quant ; i++)   //keep reading values while
maxParticles isn't reached.
  {
   int flag;  /*Tells which protocol to use. This must be the first
value in every particle declaration from the input file.*/

   fscanf(in, "%d", &flag);
   if (flag == 1)  //flags 1 - 3 manually initialize particles
    {
    inpMP(i);
    }
   if (flag == 2)
    {
    inpMPD(i);
    }
   if (flag == 3)
    {
    inpMPDA(i);
    }

   if (flag == 6) //randomly initialize x particles with the specified
minimum and maximum ranges for position, velocity, and mass.
```

```
     //all velocities are counter-clockwise and perpendicular to the
position vectors
     //Minimum position and velocity are not used.
    {
     int numP, j=0;
     fscanf(in, "%d%lf%lf%lf%lf%lf",&numP, &maxP, &minP, &maxV, &maxM,
&minM);

     double xSide, ySide, zSide, xVel, yVel, zVel, mass, xAdj, yAdj;
     while(j < numP)
      {
          //create test values...
     xSide = nRand() - .5;
     ySide = nRand() - .5;
     zSide = nRand() - .5;



        //...if the particle is inside the sphere...
     if (sqrt(xSide * xSide + ySide * ySide + zSide * zSide) <= .5 &&
sqrt(xSide * xSide + ySide * ySide + zSide * zSide) >= minP / (2 * maxP))
         //...make the particle...
     {
     xSide = (xSide * 2 * (maxP - 1)) + 1;

     ySide = (ySide * 2 * (maxP - 1)) + 1;

     zSide = (zSide * .2 * (maxP - 1)) + 1;

      if (xSide >= 1 || xSide <= -1)
      {
       xVel = (1/ xSide);
      }
      else
      {
       if(xSide > 0)
       {
         xVel = 1;
       }
       else
       {
         xVel =-1;
       }
      }
      if (ySide >= 1 || ySide <= -1)
      {
       yVel = (1/ ySide);
      }
      else
      {
       if(ySide > 0)
       {
         yVel = 1;
       }
```

```
        else
        {
          yVel =-1;
        }
       }


      zVel = 0;
       /*this normalizes the xVel and yVel variables, expressing them as the
ratio of their proportions to each other; gives an answer from about 0 to 1
(for some reason it goes over 1 sometimes) The second line multiplies the
velocities by their position from the center and the maximum velocity*/
      xAdj = xVel / sqrt(xVel * xVel + yVel * yVel);
      xAdj *= ((maxP - abs(xSide)) / maxP) * maxV;

      yAdj = yVel / sqrt(xVel * xVel + yVel * yVel);
      yAdj *= ((maxP - abs(ySide)) / maxP) * maxV;

      /*Adjusts directions so velocities are clockwise.*/
      if((int)(fabs(xSide)/xSide) + .1 == (int)(fabs(ySide)/ySide) + .1)
//find the sign of the number
      {
       yAdj *= -1;
      }
      else
      {
       xAdj *= -1;
      }


      mass = (nRand() * (maxM - minM)) + minM;

      mp(i + j,
         xSide, ySide, zSide,
         xAdj, yAdj, zVel,
         mass);
      j++;         //...and update the counter
      }

       }
      i += j;

     }
   }
}

 void inpMP(int i)
  {
   char d;
   double xp, yp, zp, xv, yv, zv, mass;
   fscanf(in, "%lf%lf%lf%lf%lf%lf%lf", &xp, &yp, &zp, &xv, &yv, &zv, &mass);
   mp(i,
        xp, yp, zp,
        xv, yv, zv,
```

```
        mass);

   }
 void inpMPD(int i)
   {
    double xp, yp, zp, mv, dxyv, dxzv, mass;
    fscanf(in, "%lf%lf%lf%lf%lf%lf%lf", &xp, &yp, &zp, &mv, &dxyv,  &dxzv,
&mass);
    mpd(i,
        xp, yp, zp,
        mv, dxyv, dxzv,
        mass);

   }

 void inpMPDA(int i)
   {
    double mp, dxyp, dxzp, mv, dxyv, dxzv, mass;
    fscanf(in, "%lf%lf%lf%lf%lf%lf%lf", &mp, &dxyp, &dxzp, &mv, &dxyv, &dxzv,
&mass);
          mpda(i,
      mp, dxyp, dxzp,
      mv, dxyv, dxzv,
      mass);
   }

 //Make Particle based on coordinates
 void mp(int pNum,
     double xp,   double yp,   double zp,
     double xv,   double yv,   double zv,
     double pMass)
   {
    position[0][pNum] = xp;
    position[1][pNum] = yp;
    position[2][pNum] = zp;
    velocity[0][pNum] = xv;
    velocity[1][pNum] = yv;
    velocity[2][pNum] = zv;
    mass[pNum] = pMass;

   }
 //Make Particle treating the velocity as a vector
 void mpd(int pNum,
     double xp,   double yp,   double zp,
     double mv,   double dxyv,   double dxzv,
     double pMass)
   {
    dxyv *= pi/180.0;
    dxzv *= pi/180.0;
    position[0][pNum] = xp;
    position[1][pNum] = yp;
    position[2][pNum] = zp;
    velocity[0][pNum] = mv * cos(dxyv) * cos (dxzv);
    velocity[1][pNum] = mv * cos(dxzv) * sin (dxyv);
```

```
   velocity[2][pNum] = mv * sin(dxzv);
   mass[pNum] = pMass;
  }

 //Make Particle treating both position and velocity as vector
quantities
 void mpda(int pNum,
    double mp,   double dxyp,   double dxzp,
    double mv,   double dxyv,   double dxzv,
    double pMass)
  {
   dxyp *= pi/180;
   dxzp *= pi/180;
   dxyv *= pi/180;
   dxzv *= pi/180;
   position[0][pNum] = mp * cos(dxyp) * cos (dxzp);
   position[1][pNum] = mp * cos(dxzp) * sin (dxyp);
   position[2][pNum] = mp * sin(dxzp);
   velocity[0][pNum] = mv * cos(dxyv) * cos (dxzv);
   velocity[1][pNum] = mv * cos(dxzv) * sin (dxyv);
   velocity[2][pNum] = mv * sin(dxzv);
   mass[pNum] = pMass;
  }


 /*
  THIS IS NEEDED BY THE COMBINE ALGORITHM
  Shifts the contents of all arrays back one at the given point and
decrements numParticles to represent the combination of two particles
 */
 void shift(int index)
 {
  int l, m;
  for (l = index; l < numParticles - 1; l++)
   {
    mass[l] = mass[l+1];
    for(m = 0; m < 3; m++)
     {
      position[m][l] = position[m][l+1];
      velocity[m][l] = velocity[m][l+1];
     }

   }
   numParticles--;

 }
 /*
  Calculates the force of gravity on each particle from all other
particles, repeats for all particles, moves the particles, then checks
to see if any have combined, and if so updates the arrays and
numParticles (decreases the total number in the system.) Then outputs
the the results if the current iteration divides evenly into the
frequency of output. Repeats _iteration_ times.
```

```
 */
void gravitate()
{

 double acceleration = 0.0;
 for(c = 0; c < iterations ; c++)
  {
 for(i = 0; i < numParticles; i++) //Affected particle
 {
  for(j = 0; j < 3; j++) //Directional component 2 = xy, 3=xyz
  {
   for(k = 0; k < numParticles; k++) //Acting particle
   {
    if(k != i)
    {
    acceleration += (mass[k] / pow(((position[0][k]-position[0][i]) *
(position[0][k]-position[0][i]) + (position[1][k]-position[1][i]) *
(position[1][k]-position[1][i]) + (position[2][k] - position[2][i]) *
(position[2][k] - position[2][i]) + SOFTENER * SOFTENER), 3.0/2.0)) *
(position[j][k]-position[j][i]);
    }
   }
    velocity [j][i] += GRAVITY * acceleration * timeStep;
    delta[j][i] = velocity[j][i] * timeStep;
    acceleration = 0.0;

  }


 }

 if(c %  freq == 0)
  {
  if( c % 2500 == 0)
   {
   printf("%2.4lf%%\n", (c / (double)iterations) * 100);
   }
  output();
  //tracker();
  }
  update();
  combine();

  }
}
/*
 Checks to see if any particles have collision coordinates, then
 combines them, adjusting their positions, velocities, and masses
 Called after update() in gravitate()
*/
void combine()
{
 double ratio;
```

```
  for(i = 0; i < numParticles - 1; i++)
   {
    for(j = i+1; j < numParticles; j++)
     {
     if((.032 * (pow(mass[i], .3333333) + pow(mass[j], .3333333))) >=
sqrt((position[0][j]-position[0][i]) * (position[0][j]-position[0][i]) +
(position[1][j]-position[1][i]) * (position[1][j]-position[1][i]) +
(position[2][j] - position[2][i]) * (position[2][j] - position[2][i])) && j
!= i)
      {
             ratio = mass[i]/mass[j];
       position[0][i]= (ratio * position [0][j] + position[0][i]) / (1 +
ratio);

       position[1][i]= (ratio * position [1][j] + position[1][i]) / (1 +
ratio);
       position[2][i]= (ratio * position [2][j] + position[2][i]) / (1 +
ratio);
       velocity[0][i]= (mass[i] * velocity[0][i] + mass[j] *
velocity[0][j])/(mass[i] + mass[j]);
       velocity[1][i]= (mass[i] * velocity[1][i] + mass[j] *
velocity[1][j])/(mass[i] + mass[j]);
       velocity[2][i]= (mass[i] * velocity[2][i] + mass[j] *
velocity[2][j])/(mass[i] + mass[j]);
       //printf("Position[%d]= %5.5f %5.5f %5.5f\n", i, position[0][i],
position[1][i], position[2][i]);
       //printf("Velocity[%d]= %5.5f %5.5f %5.5f\n", i, velocity[0][i],
velocity[1][i], velocity[2][i]);
       printf("Combined %d (%f) with %d (%f) at iteration %d\n", i, mass[i],
j, mass[j], c);
       mass[i] += mass[j];
       shift(j);

      }
     }
   }
 }
 /*
   creates new positions for the particles at the end of every
timestep
 */
 void update()
 {

  for(i = 0; i< 3; i++) //components
  {
   for(j=0; j < numParticles ; j++)
   {
    position[i][j] += delta[i][j];
   }
  }
 }

 /*
```

```
  writes every particle's position on a line in a file
*/
void output()
{

 //make a new file if the old one gets too big
 if(c % segment == 0)
  {
   fclose(out);
   sprintf(filename, "output%d.txt", c / segment);
   printf("%s\n" , filename);
   out = fopen( filename,"w");
  }

  fprintf(out, "%d\t", c);
 for(i = 0 ; i < numParticles ; i++)
 {
  fprintf(out, "%d\t", i);
   for(j = 0 ; j < 3 ; j++) //components
   {
    fprintf(out, "%.1f\t", position[j][i]);

   }

 }
  fprintf(out, "\n"); //begins a new line
}
```