Simulating Solar System Formation Theories

New Mexico Adventures in
Supercomputing Challenge
Final Report
April 6, 2005

Team 003
Albuquerque Academy

Team Members
Charlie Clauss
Alfredo Davila
Thien-Cam Nguyen
Matt Strange
Jennifer Turner

Project Mentor
Dr. Jim Mims

Contents

Executive Summary

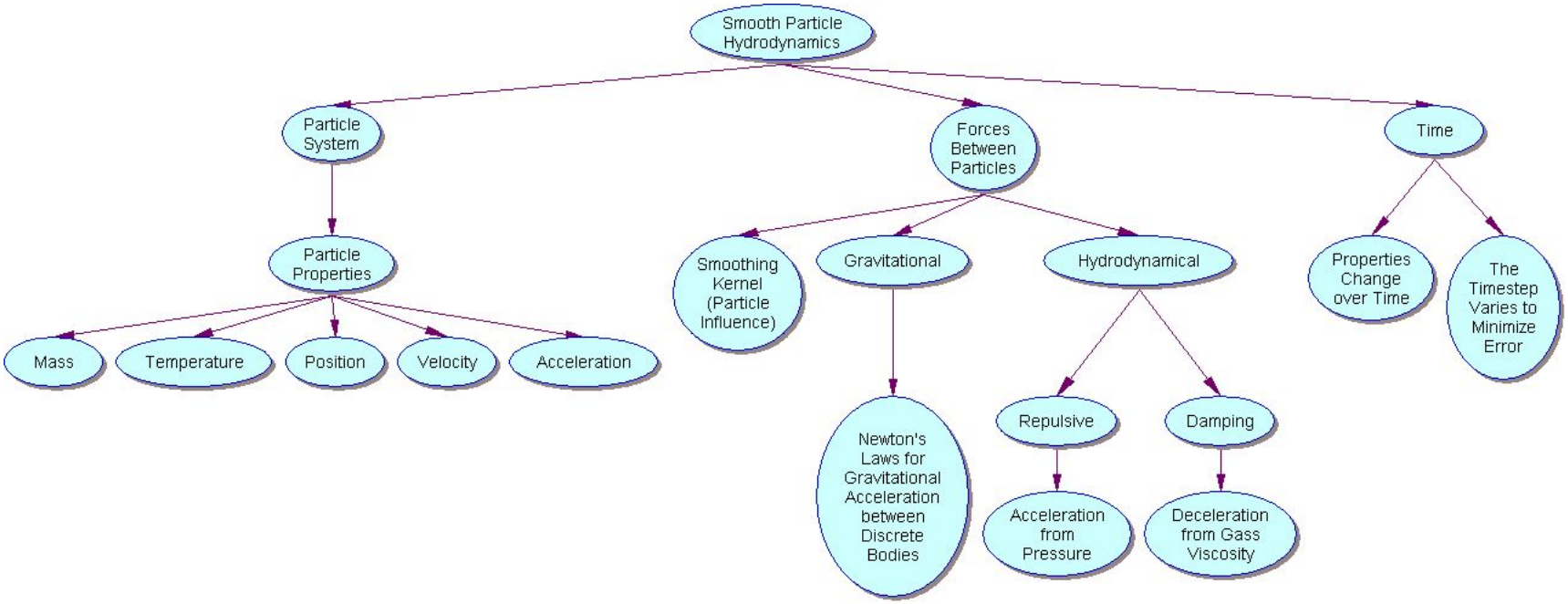
This project examined the salient features of nascent solar systems, i.e. the gravitational and hydrodynamical forces present within a gas cloud as it condenses. Specifically, this project aimed to see what initial conditions were necessary to create the primordial stage of a solar system – the accretion disc. We developed an optimized computer program that was capable of simulating a gaseous environment and dispersing shock waves through the gas cloud. We developed a number of simulations involving these shock waves and examined how the initial conditions of the shock wave affected the kind of accretion disc that was or was not consequently formed. Our results demonstrated that the position of the shock wave was not as important as the rate at which its pressure dispersed. We were able to determine a value at which this decay would produce an accretion disc that was balanced in both central mass and surrounding centripetal motion. The information we gathered from these simulations allowed us to answer some questions regarding a new theory of solar system formation which had originally sparked interest in the project.

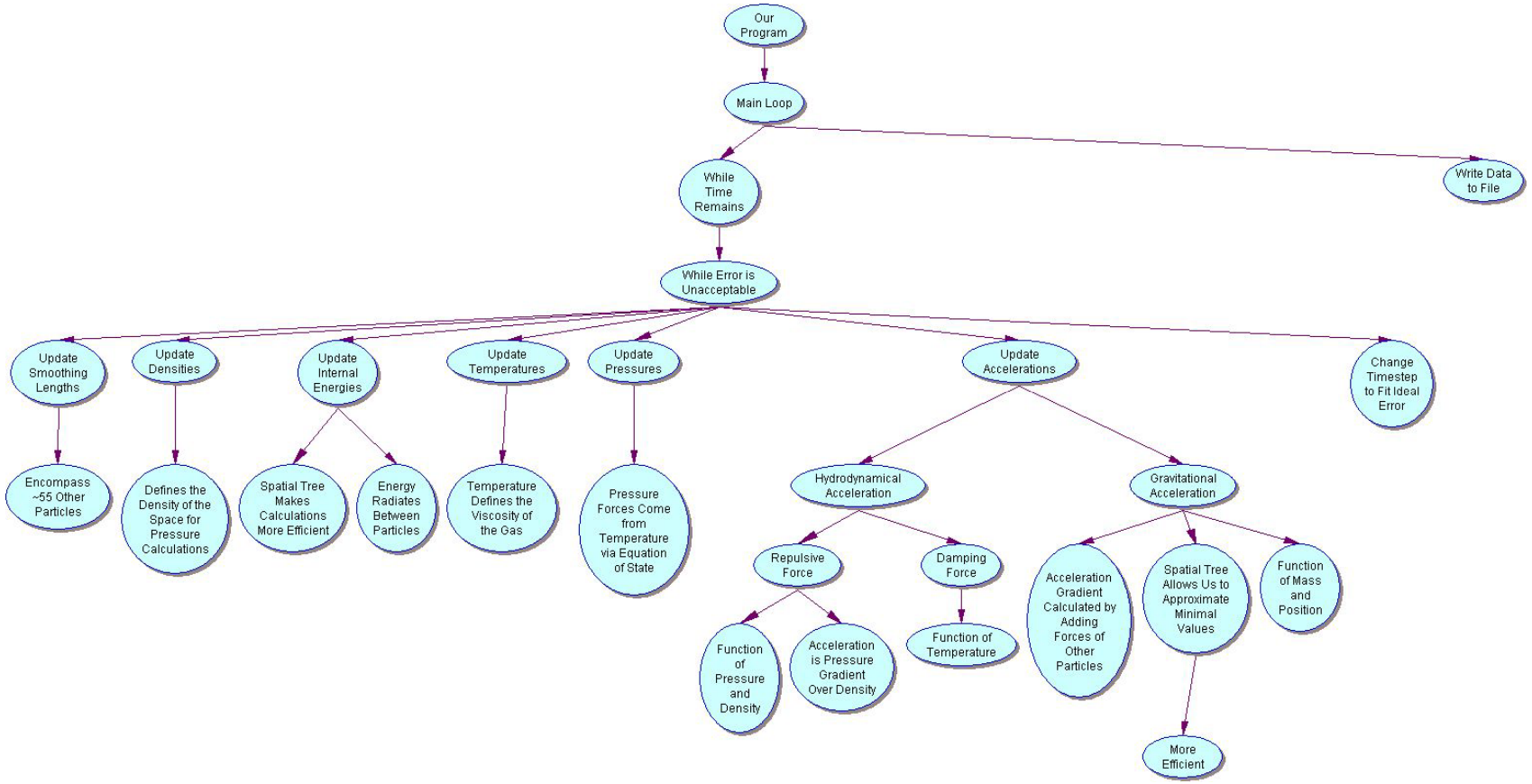
Introduction

Many methods exist to study the formation of solar systems, whether through energy analysis, chemical analysis or computational simulation. Such tests challenge the inherent assumptions of the theory and often lead to its being altered in some manner. Our project aims to test a new theory for the formation of our own solar system. The theory, presented in a press release from Arizona State University, suggests that a violent environment could have created our solar system. Traditionally, formative theories have held that gradual gravitational forces would condense nebula gases into protostars and protoplanets. However, there exists chemical evidence that other forces may have been present (e.g. enriched elements on planets as a result of nearby supernovae). Thus, taking the effects of UV radiation and strong shock waves from massive stars into account, a different kind of simulation can be tested. Our team became interested in the ramifications of such a formative theory and decided to create a computational simulation to study it; something we believe has not been attempted previously. We simplified to the point of just examining the effects of the radiation and shock waves, measured internally via *internal (non-potential) energy* and *pressure*, respectively. After researching computational methods and implementing an acceptable simulator, we took a cloud of gas in equilibrium, simulated a thermal stimulation and a subsequent shock wave (pressure stimulation) from an external source and studied the results. Thus, we could test how various permutations of the theory might or might not condense into the kind of protostar / protoplanetary disc system indicative of future solar systems. Going beyond this point requires different analyses (e.g. evaporation and thermonuclear forces) not related to our initial goal of studying the formative theory.

Methods

The following sections detail the physical and numerical methods our project employs. Two flowcharts are presented first to introduce both general SPH and our code.


Smooth Particle Hydrodynamics
Particle System
Forces Between Particles
Time
Particle Properties
Smoothing Kernel (Particle Influence)
Gravitational
Hydrodynamical
Properties Change over Time
The Timestep Varies to Minimize Error
Mass
Temperature
Position
Velocity
Acceleration
Newton's Laws for Gravitational Acceleration between Discrete Bodies
Repulsive
Damping
Acceleration from Pressure
Deceleration from Gass Viscosity


Our Program
Main Loop
While Time Remains
Write Data to File
While Error is Unacceptable
Update Smoothing Lengths
Update Densities
Update Internal Energies
Update Temperatures
Update Pressures
Update Accelerations
Change Timestep to Fit Ideal Error
Encompass ~55 Other Particles
Defines the Density of the Space for Pressure Calculations
Spatial Tree Makes Calculations More Efficient
Energy Radiates Between Particles
Temperature Defines the Viscosity of the Gas
Pressure Forces Come from Temperature via Equation of State
Hydrodynamical Acceleration
Gravitational Acceleration
Repulsive Force
Damping Force
Function of Pressure and Density
Acceleration is Pressure Gradient Over Density
Function of Temperature
Acceleration Gradient Calculated by Adding Forces of Other Particles
Spatial Tree Allows Us to Approximate Minimal Values
Function of Mass and Position
More Efficient

Merits of Our Simulation Setup

Computational studies of solar system formation are old enough to have been researched thoroughly. When we first began research, it was apparent that such studies are usually done with particle simulations, where a set of particles approximates the gas cloud and the particles react with one another. Two methods have evolved: grid-based methods (where forces are interpolated to a grid and then back to particles via Fourier transforms) and smooth particle hydrodynamic (SPH) methods, which use a smoothing function to interpolate forces in and out of the particle system. Although we had some experience with the grid-based methods (a gravitational Particle-Mesh study last year), SPH was attractive because it produced fewer singularities in calculations and, to be general, acted more like a gas cloud than a predefined set of particles.

SPH is largely standardized; the methods we used primarily consisted of optimizations from ~1986 and datasets from ~1994-6, ratified within a 2001 dissertation on SPH's application to another formative theory (the 'Capture Theory'). Many SPH texts are devoted solely to its applications in astrophysics; however, these remained out of our reach due to their obscurity and price. We found the before-mentioned source and decided to use it since the majority of its citations were available in public libraries or upon the internet.

How SPH Works

For those unfamiliar with SPH, below is a general overview of how the system works. The nebula is approximated by a set of particles (in our case, a uniform 3-space grid) with masses (in our simulation, these were set to be equal at 1e8 kg), kinetic energies (initially set to zero in anticipation of the oncoming radiation waves) and temperatures (set to 100 K in anticipation of oncoming radiation waves). A number of other physical variables must be deduced from these initial values (see *Physics Equations*). At each step of the simulation, certain values have been integrated forward in time (namely accelerations and internal energy values). From this, the general changes in the system are calculated at each particle from each other particle (computationally, this is a problem; please see *Optimizations* for our solutions). When calculating the effect of one particle on another, the calculation is not done directly, but with a Smoothing Kernel, which avoids numerical singularities by distributing the calculation in a manner akin to the classic Gaussian Bell Curve.

SPH Choices

Many different options exist when creating an SPH simulation. We chose to use variable smoothing lengths, a technique that maximizes the probabilistic uniformity of particle interactions. Since the Smoothing Kernel is influenced only by distance and smoothing length, a dynamic smoothing length would accompany the dynamic distance changes (i.e. acceleration and velocity) that was inherent in our simulation. Our kernel is defined as a cubic spline (Monaghan and Lattanzio spline), rather than a Gaussian, since Gaussian curves have no defined endpoints. To avoid gravitational singularities, we used a setup derived from the kernel (i.e. it integrates the gravitational force out of a sphere, smoothing with the kernel, which is similarly dependent upon *r* [Hernquist and Katz method]).

Physics Equations

We used standard physics equations that always take the kernel into account when a distance is involved. These equations allow us to map other physical values upon the particles when certain initial values are known (see *How SPH Works*).

W is the kernel, and h is the smoothing length (particle's effect upon others). R is a distance vector.

Density is calculated from the combined presence of other particles; for all j in range, p += m(j) * W(r(ij), h(j)).

Gravitational force is calculated using gravitational smoothing that is approximately 1/10 a particle's smoothing length. For all j in range, F += [m(j)*r(ij)] / [(r(ij)^2 + gh(ij)^2) * r(ij))], where gh is the gravitational smoothing factor (notice that it affects the denominator, where singularities can arise most easily).

As mentioned above, the simulation is moved forward by a timestep. Velocities combine as (ts * accel(t)). Some of the issues regarding the timestep are discussed in the Research Notes.

The pressure force from the shock wave has been defined in a manner somewhat different from the others; notably, it is controlled via compile-time constants. Having worked with such adjustable variables in genetic algorithms, it seemed fitting to provide some adjustable variables for this simulation as well. Initially, we had hoped to implement a pattern-recognition system, which, when tied into a genetic algorithm, would examine the simulations itself. However, our time constraints forced us to analyze the results ourselves and adjust code internals similarly.

The shock wave moves faster than the speed of sound to keep the simulation feasible on a desktop system. We give the wave a speed value, a maximal pressure value,

a dampening value (i.e. the size of the wave) and a time-based decay value, which is subtracted from the maximum pressure every second. This setup allows us to do mathematical tests involving the above values. For details, see the *pressure.cpp* file.

Time constraints kept this one section of code unoptimized. It is, currently, the one section of the program that suffers from the O(N^2) flaw, as we were unable to find appropriate symmetrization data for pressures.

SPH Optimizations

To avoid the computationally atrocious O(N^2) (i.e. every particle against every other particle), our code both uses a spatially divided tree which makes it simple to create a limit to the number of calculations that will be performed. Using popular Multipole Acceptability Criteria (MACs), we were able to determine whether certain branches of the spatial tree were worth pursuing at all or whether they should be approximated since their contributions would be negligible. A similar scheme was applied to hydrodynamic calculations, which involve large numbers of "neighbor searches" to spread energies across the system.

Wherever possible, we try to avoid the computationally expensive square-root value in calculating distance, compensating by squaring the entire equation.

A very specialized quicksort algorithm is used in the smoothing length process to find the distance to the 55^{th} particle away (how dynamic smoothing lengths are assigned).

Discussion

Our code approximated a shock wave in its final version. The shock wave had a number of variables that could be set; position and decay speed, for example. This gave us a test bay in which to study shock waves.

We had already decided upon looking for a system that:

1. appeared stable (i.e. particles were not being active lost),
2. had a center body (i.e. a protostar), and also
3. had particles with forces orthogonal to the inward force (i.e. centripetal motion).

After moving the origin of the shock wave a number of times, it was decided that position was a relative factor, and that it would either:

1. delay / speed up the condensation process depending upon its distance or
2. translate the position of the protostar depending on the angle between wave and particle system.

Thus, the shock wave's origin was fixed in a corner 50 timesteps away from the nearest particle, with the assumption that this would give us the best view of pressure forces acting in multiple dimensions.

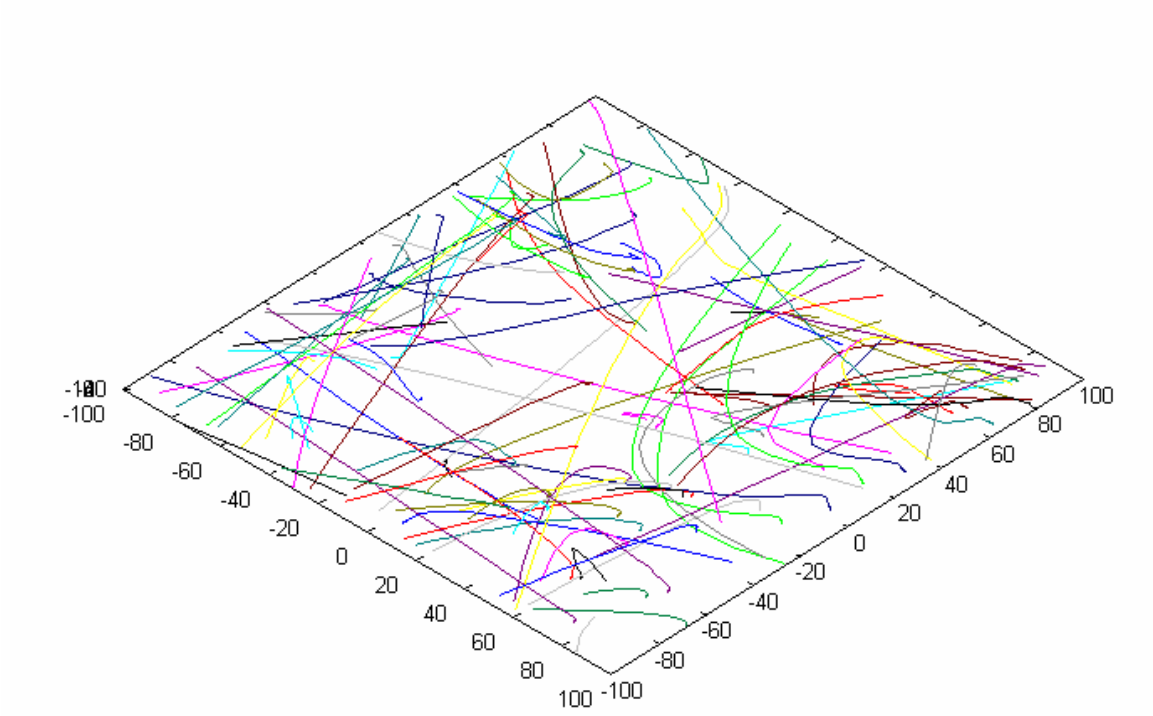
It became obvious that the most important tradeoff in our simulation would be between the central forces of the massive protostar-type center body and the diverse forces of the simulated shock wave. Another variable was added to accommodate this reasoning, the decay variable, which simulated the speed of pressure dispersal through the gas. We had assumed that a large wave-type force that was followed by the forces of a newly formed protostar-type body would create the fundamentals of an accretion disc. This setup worked well – particles gained acceleration coefficients in many directions,

but most were pulled back by the newly-formed massive body that had formed in the wake of the emerging shock wave. In other words, the shock wave brought about the appropriate directional entropy to make the system disc-like (previous systems inevitably collapsed into a single point rather than a single point with surrounding bodies.

Studies of the magnitude of the decay found that decay of 0.5% of the maximum possible pressure each second had the best balance of massive center particles and surrounding particles with centripetal motion. However, this simulation had been scaled to work on a desktop computer. The number of particles was kept at 1000 and masses were scaled down to avoid excessive gravitational forces. Thus, the numbers are mostly indicative of how decay relates to other scaled values.

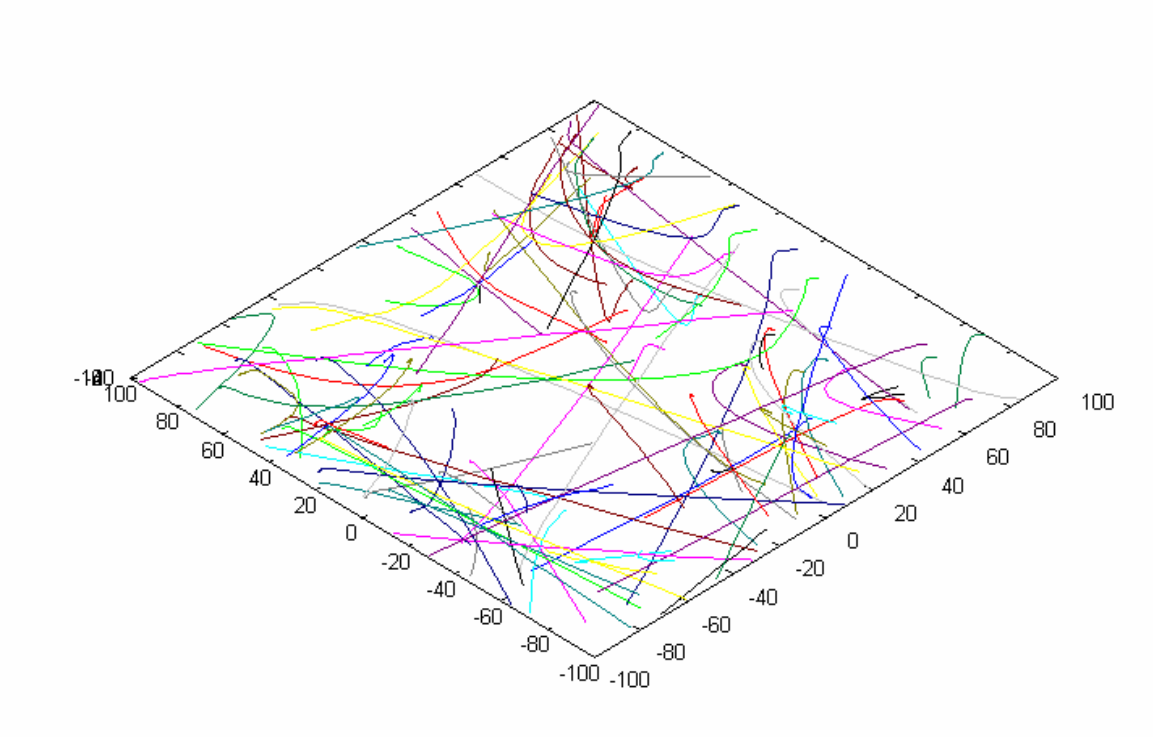
Images

These images are gnuplot models of the paths of each particle in the system. The shot is taken from the corner (-100,-100,-100), with the Z axis tilted as far down as possible.



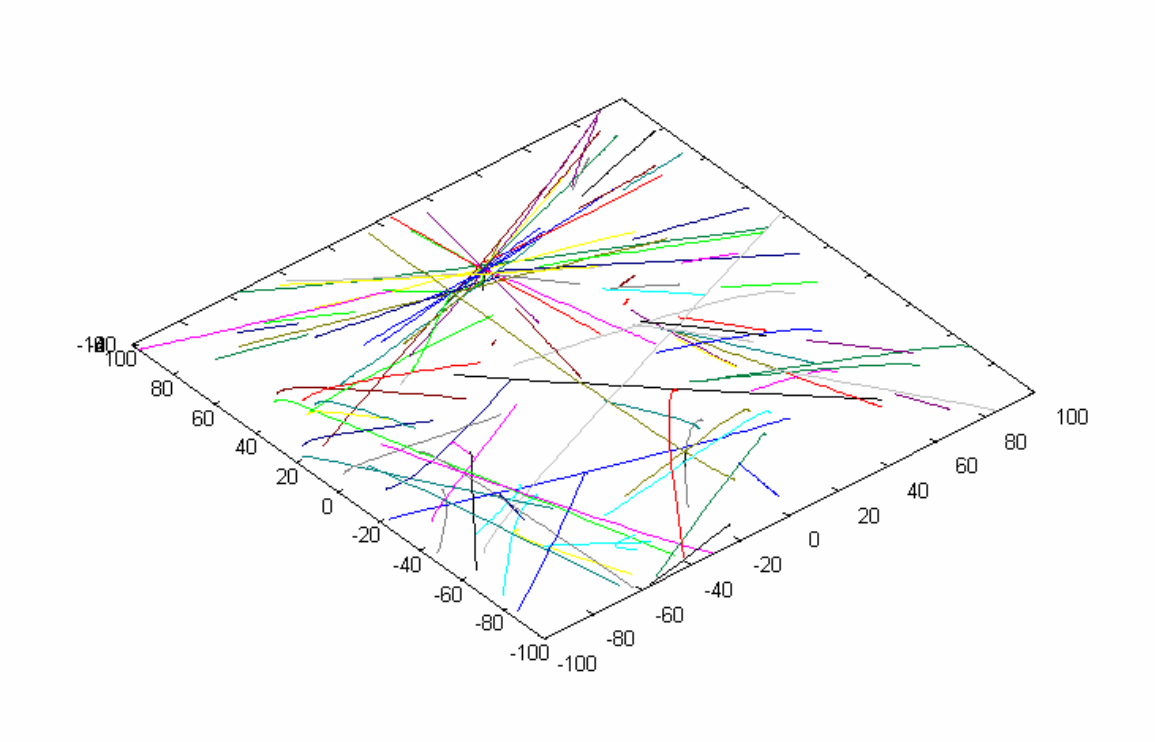
Decay of 0.005% of Total Pressure per Second

Notice the chaotic nature of the system with this low decay value. The shock wave has emanated from the point -100, -100 and proceeded to push the particles out of the area.



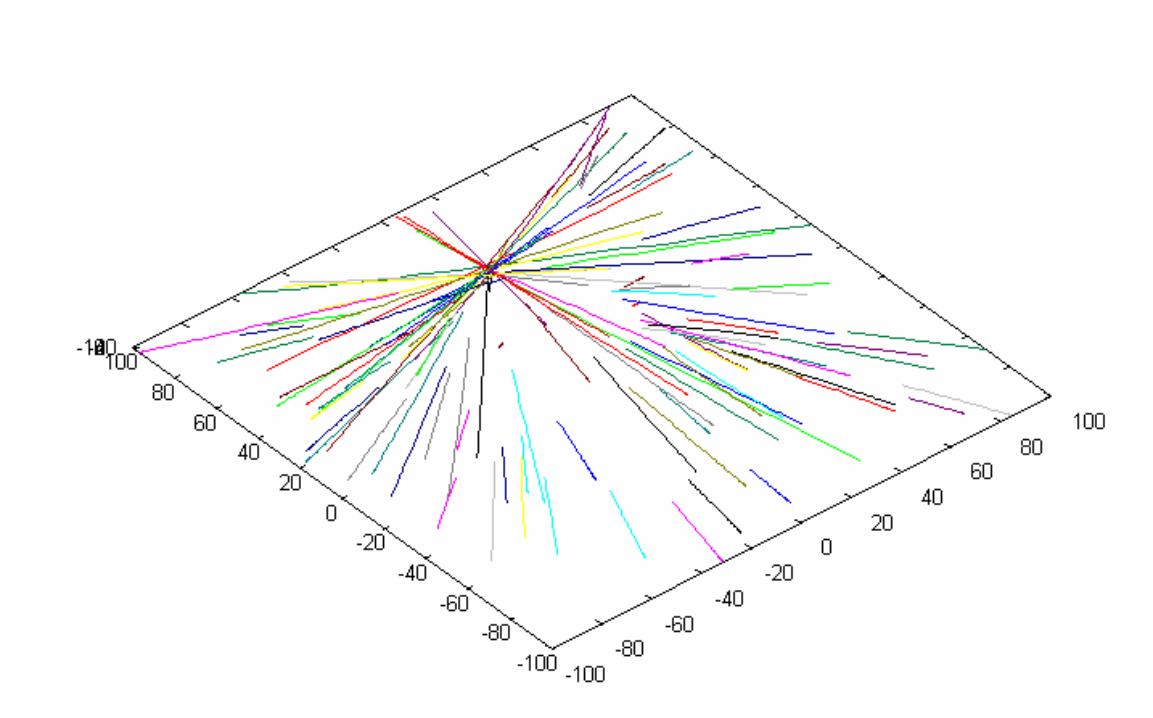
Decay of 0.05% of Total Pressure per Second

This decay is an order of magnitude faster and so there is some noticeable gravitational interactions taking place, e.g. the top of the graph. However, most of the particles simply leave for deep space (notice the curving motions of particles are they are pushed away along the edges).



Decay of 0.5% of Total Pressure per Second

This came as a surprise to us: the first simulation in which a center particle had formed despite the shock wave. We were very excited upon seeing this test, as it indicates that both central masses and moving bodies are present (note the trajectories of particles that run perpendicular to the massive central-body gravitational force).



Decay of 5.0% of Total Pressure per Second

At this point force decays at ~5% every second. Obviously, there are no oppositional forces here – it is as though the shock wave never passed through the system to disrupt the creation of the massive central body. Such a decay value is, then, too large for our system and defines the upper bound of useful decay values.

Conclusions

Our simulation provided us a method to study how accretion discs (the basis of new solar systems) form. By designing a computer program, we were able to analyze both gravitational and hydrodynamical forces within the system. In this environment, we introduced a shock wave (or pressure wave) that was controlled with a number of variables. After experimenting with the variables, we found that the most important factor was the decay of the shock wave, i.e. the time that elapsed between the passing of the shock wave and the formation of a massive protostar that would spur centripetal motion in the accretion disc. Regarding the theory of solar system formation in violent environments, our experiments demonstrated that the force and position of the shock wave was irrelevant to the creation of the accretion disc, and that the rate of decay of the shock wave was of primary interest. In other words, the positioning of the forces is unimportant, since the decay factor determines the ability of the system to have centripetal motion; vital for the production of a future solar system. Since the pressure forces are likely quite large in such a theory, the gas would need to be dense to facilitate dispersion and mimic the decay factor of our simulation

Most Significant Achievement

Our most significant achievement in this project was the introduction of the spatial optimizations to make the code more efficient. Upon implementing an $O(N^2)$ method, we decided to discover a more efficient means of calculating the influence of particles upon on one another. The spatial tree is a well studied optimization that can be implemented gracefully using recursion. By adding appropriate multipole acceptability criteria, we could decide when to approximate a certain area of particles and avoid iterating through each one. This allowed us to add a higher resolution of particles to the simulations. Having worked with N-body methods in the past, we felt that our improvements set this year's work apart from that of previous years.

References

Jeff J Hester, Stephen J Desch, Kevin R Healy, Laurie A Leshin. (2004). The Cradle of the Solar System. http://eagle.la.asu.edu/hester/papers/science_perspective.pdf

Oxley, Stephen. (2001). Modelling the Capture Theory for the Origin of Planetary Systems. http://www.droxley.freeserve.co.uk/

Press, William et al. (2002) Numerical Recipes in C++: The Art of Scientific Computing Second Edition. Cambridge: Cambridge.

Semenov, Dimitri A. (1996). The Opacity Model of Henning & Stognienko. http://www.mpia-hd.mpg.de/homes/henning/Dust_opacities/Opacities/opacities.html

New Theory Proposed for Solar System Formation. (2004) *Universe Today*, May 21.

Research Notes

12/1/04

We researched the various methods for nebula simulation today. I am familiar with grid based methods since a particle-mesh simulation with FFTW last year. However, since we are interested in the SPH method and know it is often used in these types of projects, we will investigate it as well.

12/5/04

I have found information on different SPH simulations, some involving fluid flows and a number that address the system on distributed computers. There are a number of SPH texts that seem out of reach currently, so research will continue.

12/7/04

We found and have decided to use a dissertation that is published online and uses SPH to model the capture theory, another system of solar system formation. Since it was published in 2001, we feel it is sufficiently up-to-date in its analysis of numerical methods for the system, and it contains an interesting tree-base code description that we intend to implement.

12/12/04

Despite being busy with schoolwork, we have developed the skeleton of the project as it is usually explained:

1. Update Centers of Mass
2. Update Smoothing Lengths

3. Update Densities
4. Update Internal Energies
5. Update Accelerations
6. Integrate

We will fill these sections in with code later.

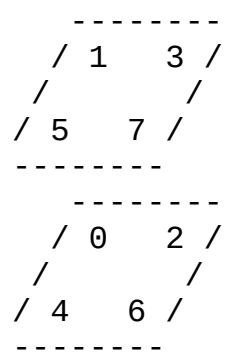
1/3/05

Looked over our primary source again. It is a recursive implementation; and the pseudocode is mired in Fortran-77 style iterative notation. Thankfully, since we are using C++, we can implement full recursion. I have developed a description of how our code will handle the recursive subdivisions:

```
Documentation: The N-ary Tree

     The tree represents a three-dimensional `gridding' of
the particle universe.  Assume that the space is
represented via the standard right-handed coordinate
system.

      |X ^+
      |
      |
      |_______ Y >+
     /
    /
   / Z \/+


     The grid is created as cubes.  Each cube contains eight
children.  The children are as follows:

name: {x-distance, y-distance, z-distance}

`0' as a distance represents the minimum.  `1' represents
the maximum.

0: {0,0,0}
1: {0,0,1}
```

```
2: {0,1,0}
3: {0,1,1}
4: {1,0,0}
5: {1,0,1}
6: {1,1,0}
7: {1,1,1}
```

In other words, the children would be arranged on two different planes (orthogonal to the Z-axis) as follows:



1/20/05

Created a working version of the tree and a density traversal based off of it. This was more difficult than anticipated, as I had a bug that I had thought was some kind of race-condition bug, but was actually caused by flipping a greater-than sign. I have set up Emacs to highlight these in the future to avoid wasting any more time on the issue.

1/23/05

Using the same traversal system as the density, I implemented the gravitational force calculations today. I then checked by setting up a universe with only two particles and having them collide. This test ensures that vectors are calculated in the correct directions.

2/5/05

Found a bug that had gone unnoticed involving the process used to index subcubes of the tree. The problem was fixed by creating a single macro that is called every time the indexing system is needed.

2/21/05

There is not a lot of time left until the science fair, so I have quickly put together a shock wave simulator that the GA can adjust to create systems. Systems of ~1000 particles seem to run in a reasonable amount of time. The shock waves emanate from a point and apply themselves to the pressure value of all particles in their range. A traversal is then performed to analyze the forces of these particles's pressure upon the other particles (it is a repulsive acceleration defined in the primary source as Pressure / Density).

2/22/05

I realized today that collisions are probably the most annoying detail to track in a gravitational simulation. This is probably not apparent since it would seem that the collision detection process is fairly straightforward. However, that is an assumption that works only in real-time. This simulation is real-time, but works in the keyframe / timestep system. In other words, particles can sling by one another during the timestep and gain giant acceleration coefficients when they should have actually run into one another. I would like to add some vector entanglement code that would sort this out correctly, but time is going to force me to work upon the other forces instead. For the time being, we must be sure to keep timesteps short, keep the particle-proximity check large (currently 3 units) and watch for particles with overzealous trajectories. My vector entanglement system (this isn't done in industry simulations since they have

supercomputers) is going to keep track of trajectories and then traverse the spatial tree observing where trajectories would intersect particle-laden cubes. Prior knowledge should keep it O(N) scalable.

3/1/05

Normally we set the world up and watch it implode because of gravity. This is a good thing; it demonstrates how a massive body is formed. However, we need both a massive body and an accretion disc, and the accretion disc needs angular momentum. This all came to light during our first test, in which we setup the shock wave system (runs faster than the speed of sound to keep the simulation time desktop-accessible) to begin at the origin of the system and go outward. As expected, the system exploded, leaving a blank hole in its center. Obviously, the shock waves will be from an external source, but the position of this source is what we need to investigate. It should be angled in such a way as to provide a majority of its forces orthogonal to the inward forces that we usually see.

An initial test was done upon the corner of the system, but yielded no discernible differences. We realized the problem had come from the introduction of (as recommended) kernel smoothing upon the pressure forces. Overall pressure forces were increased in a further study. That is a noticeable conclusion: *pressure forces must be of a magnitude comparable to that of gravity / other forces*. However, since the shock wave must come from a 'larger body', this can be readily assumed beforehand.

3/8/05

Overall, it appears that similar rotational and translational forces can be arranged

via any permutation of angle and force. I believe the most important factor is "when the gravitational forces should take over", i.e. the decay of the shock wave as it spreads throughout the system.

Responding to these results, we defined a SHOCK_DECAY value in our simulation that will do this in tests. When SHOCK_DECAY is small, tests tend towards entropy. When SHOCK_DECAY is large, tests usually sort themselves into a pattern characteristic of the original center-body setup.

3/9/05

I ran some shock wave tests today. I began with a decay value of 5, progressed to 50, 500 and 5000, which had the most effective balance of angular momentum and center-body creation. At a decay value of 50000 (approximately 5% of the maximum possible pressure), the angular motion is no longer visible. This brackets the decay within an order of magnitude measure and provides us some interesting data. From this data I conclude that the decay value is of primary importance in our simulation. As applied to a "violent environment" simulation, this would suggest that it was necessary for large shock waves (of the kind generated by massive bodies) to disperse quickly, i.e. with little viscosity or other pressures to oppose them. So I have answered my own major question, finding out what kind of forces can lead to the creation of an accretion disc. We will move from working on the code to working on the writings for Regional Science Fair.

Code

accels.cpp

```
// Gravitational/Hydro accel calculation

#include "system.hpp"
#include "kernel.hpp"

void
System::accelsCalc()
{
  int i;
  FOR_ALL_PARTICLES(i)
    {
      if(!ps[i].Is_Active())
        continue;
      tree.accelsTraverse(&ps[i]);
      // std::cout << ps[i].aGet(0) << " " << ps[i].aGet(1) << " " <<
ps[i].aGet(2) << "\n";
      if(use_accel_checking)
        {
          int j;
          FOR_ALL_PARTICLES(j)
            {
              if(!ps[j].Is_Active() || i==j)
                continue;
              FP r = d3d(&ps[i], &ps[j]);
              FP rx = ps[j].x_get() - ps[i].x_get();
              FP ry = ps[j].y_get() - ps[i].y_get();
              FP rz = ps[j].z_get() - ps[i].z_get();
              ps[i].a_check_add( ((G * ps[j].Get_Mass() * rx) /
(r*r*r)),
                                 ((G * ps[j].Get_Mass() * ry) /
(r*r*r)),
                                 ((G * ps[j].Get_Mass() * rz) /
(r*r*r)) );
            }
        }
        //pressure hack
        //FOR_ALL_PARTICLES(i)
        {
           //if(!ps[i].Is_Active())
            //  continue;
           int j;
           FOR_ALL_PARTICLES(j)
           {
              if(!ps[j].Is_Active())
                 continue;
              FP pressure_a[3];
              FP hij = symSmoothLens(ps[i].hGet(), ps[j].hGet());
              FP coeff = kernel(d3d(&ps[i], &ps[j]), ps[j].hGet()) *
```

```
                        (ps[j].Get_Pressure() /
                        pow(ps[j].densityGet(),2));

              rvect(pressure_a, coeff, &ps[i], &ps[j]);
              ps[i].aSub(pressure_a);      //get away from strong
pressure
           }
         }
    }
}

// calc graviational & hydrodynamical forces
void
Tree::accelsTraverse(Particle *p)
{
  int i;
  FP a[3];
  // single case
  if ((particle != NULL) && (particle->Is_Active()))
    {
      FP rij = d3d(p, particle);
      FP hij =  symSmoothLens(p->hGet(), particle->hGet());
      FP ghij = symSmoothLens(p->ghGet(), particle->ghGet());
      // gravitational softening
      if( (rij < 2*p->ghGet()) && (rij < 2*ghij))
        {
          rvect(a, G*particle->Get_Mass()*gKernel(rij,hij), p,
particle);
          p->aSub(a);
          if(rij < 2*particle->ghGet())
            {
              // will interact again under j, symmetry will be done
            }
          else // must ensure symmetry
            {
              // apply force from i to j
              // add, rather than subtract, since this is in the
opposite direction
              rvect(a, G*p->Get_Mass()*gKernel(rij, hij), p, particle);
              particle->aAdd(a);
              // point-cluster force should be negated
              rvect(a, G*p->Get_Mass()*(1.0/pow(rij, 3)), p, particle);
              particle->aSub(a);
            }
        }
      else
        {
          rvect(a, G*particle->Get_Mass()*(1.0/pow(rij,3)), p,
particle);
          p->aSub(a);
        }
    }
  // recurse to children
  else if(subdivided)
    {
      // SPHMac takes precedence
      // is this cube w/in the 4*h box?
```

```
      if(SPHMac(4.0*p->hGet(), p))
        {
          // do actual tests
          for(i=0; i<8; i++)
            {
              if(subcubes[i]->SPHMac(4.0*p->hGet(), p))
                subcubes[i]->accelsTraverse(p);
            }
        }
      // might not need to do tests, try BHMac
      else
        {
          if(BHMac(p))
            {
              FP rij = d3d(p, &cm);
              rvect(a, G*cm.Get_Mass()*(1.0/pow(rij, 3)), p, &cm);
              p->aSub(a);
            }
          else
            {
              // BH_MAC is no good, do actual tests
              for(i=0; i<8; i++)
                {
                  subcubes[i]->accelsTraverse(p);
                }
            }
        }
    }
}
```

center_mass.cpp

```
// center of mass stuff in system and tree

#include "system.hpp"
#include "tree.hpp"

void
System::CenterMass_Update()
{
  // update the center of mass information
  // runs recursively on master tree
  tree.Mass_Traverse();
}

// update the masses of the tree & its children via a traversal
void
Tree::Mass_Traverse()
{
  int i;
  m = 0;        // refind total masses

  // single case
  if (particle != NULL)
    {
      m += particle->Get_Mass();
      cm.Set_Pos(m * particle->x_get(),
```

```
                m * particle->y_get(),
                m * particle->z_get());
    }
  // recurse to children
  else if(subdivided)
    {
      for(i=0; i<8; i++)
        {
          subcubes[i]->Mass_Traverse();
          m += subcubes[i]->Mass_Get();
          // check for the subcubes' cm values
          cm.Add_Pos(subcubes[i]->Mass_Get() * subcubes[i]->CM_Get()-
>x_get(),
                     subcubes[i]->Mass_Get() * subcubes[i]->CM_Get()-
>y_get(),
                     subcubes[i]->Mass_Get() * subcubes[i]->CM_Get()-
>z_get());
        }
    }
  // CM finalize
  // this check is only applicable to massless parts of tree that were
created anyway
  // since the points are all multiplied by their mass, it's not really
necessary
  // but i figure a divide by zero could bail on some machs.
  // TODO: save some space & cycles by flagging empty tree branches
  if(m!=0)
    cm.Div_Pos(m);
}

// little recursive debugger
void
Tree::CM_Traverse_Print() const
{
  static int i = 0;
  std::cout << "Tree Branch: " << ++i << "\n";
  cm.Coords_Print();
  if(subdivided)
    {
      int j;
      for(j=0; j<8; j++)
        subcubes[j]->CM_Traverse_Print();
    }
}
```

data.cpp

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
#include <cstdio>
#include "maths.hpp"
#include "system.hpp"
#include "data.hpp"
```

```
// remain open until program ends
std::ofstream outfile;

void
System::Data_Read()
{
  FP h;                       // initial smoothing length
  int i;
  std::ifstream infile;

  // ------------------------------------------------------------------
--------------
  // simulation variables
  std::cout << "Opening ./input.dat ...\n";
  infile.open(filename_input, std::ios::in);

  infile >> t_final;
  std::cout << "Time to run is " << t_final << ".\n";
  infile >> t_step;
  std::cout << "Timestep is " << t_step << ".\n";

  infile >> h;
  // std::cout << "Initial smoothing length is " << h << ".\n";
  // set for all particles below
  infile >> use_var_h;
  std::cout << (use_var_h? "Using " : "Not using ") << "variable
smoothing lengths.\n";
  infile >> use_accel_checking;
  std::cout << (use_accel_checking? "Using " : "Not using ") <<
"acceleration checking.\n";

  std::cout << "Closing ./input.dat ...\n";
  infile.close();
  //------------------------------------------------------------------
-------------
  // particle system data
  std::cout << "Opening ./ps.dat ...\n";
  infile.open(filename_system, std::ios::in);

  FP mass = 0;
  FP x,y,z;

  FOR_ALL_PARTICLES(i)
    {
      infile >> x >> y >> z;
      infile >> mass;
      if(infile.bad())
      {
        std::cerr << "data.cc ERROR: ps.dat errored at ";
        std::cerr << i << ".\n";
        std::exit(1);
      }
      ps[i].Set_Mass(M_UNIFIED);
      ps[i].Set_Pos(x,y,z);
      // debug
      // std::cout << "Extending borders: " << x << " " << y << " " <<
z << std::endl;
```

```
      // ps[i].Set_Smoothing_Length(h);

    }

  std::cout << "Closing ./ps.dat ...\n";
  infile.close();


}

void
System::dataWriteBegin()
{
  std::cout << "Opening outfile" << filename_output << "...\n";
  char buf[20];
  int i;
  FOR_ALL_PARTICLES(i)
    {
      sprintf(buf, "%s%d", filename_output, i);

      outfile.open(buf, std::ios::out);
      outfile << "\n";
      outfile.close();
    }
    std::ofstream out2("./output/percent_regression.dat");
    out2 << "\n";
    out2.close();
}

void
System::dataWriteEnd()
{
  char buf[20];
  int i;
  FOR_ALL_PARTICLES(i)
    {
      sprintf(buf, "%s%d", filename_output, i);

      outfile.open(buf, std::ios::app);
      outfile << "\n";
      outfile.close();
    }
}
void
System::Data_Write()
{
  int i;
  char buf[20];

  FOR_ALL_PARTICLES(i)
    {
      if(!ps[i].Is_Active())
        continue;

      sprintf(buf, "%s%d", filename_output, i);
```

```
      outfile.open(buf, std::ios::app);
      outfile << ps[i].x_get() << "\t" << ps[i].y_get() << "\t" <<
ps[i].z_get() << "\n";
      outfile.close();
    }
  // std::cout << ps[i].x_get() << "\t" << ps[i].y_get() << "\t" <<
ps[i].z_get() << "\n";
}
```

data.hpp

```
#ifndef _DATA_H_
#define _DATA_H_

static const char * filename_input = "./input.dat";
static const char * filename_output = "./output/output.";
static const char * filename_system = "./ps.dat";

#endif
```

debug.hpp

```
// turn most stdout off unless doing a debug compilation
#ifndef _DEBUG_HPP_
#define _DEBUG_HPP_

#ifdef DEBUG
#define DEBUG_MSG(X) {std::cerr << X;}
#else
#define DEBUG_MSG(X) ;
#endif

// fatals work under opt compile also
#define FATAL(X) {std::cerr << X; exit(1);}

#endif
```

density.cpp

```
// calculates density with Monaghan and Lattanzio method (1985)
#include <iostream>
#include "system.hpp"
#include "tree.hpp"
#include "kernel.hpp"
void
System::Densities_Update()
{
  int i;

  FOR_ALL_PARTICLES(i)
    {
      if(!ps[i].Is_Active())
        continue;
      ps[i].densitySet(0);
      tree.densityTraverse(&ps[i]);
      // std::cout << ps[i].densityGet() << std::endl;
    }
```

```
}

// update the density of particle i
void
Tree::densityTraverse(Particle *p)
{
  int i;

  // single case
  if ((particle != NULL) && (particle->Is_Active()))
    {
      FP rij = d3d(particle, p);
      FP hij = symSmoothLens(p->hGet(), particle->hGet());
      if((rij < 2*p->hGet()) && (rij < 2*hij))
        {
          p->densitySet((p->densityGet() + particle-
>Get_Mass()*kernel(rij,hij)));

          // add contribution to j, particles will not interact again
          if(!(rij < 2*particle->hGet()))
            particle->densitySet( particle->densityGet() + p-
>Get_Mass()*kernel(rij,hij));
        }
    }
  // recurse to children
  else if(subdivided)
    {
      for(i=0; i<8; i++)
        {
          subcubes[i]->densityTraverse(p);
        }
    }
}
```

distance_vector.cpp

```
#include <iostream>
#include <iterator>
#include "distance_vector.hpp"
#include "system.hpp"

DistanceVector::DistanceVector()
{
  target = NULL;
}

DistanceVector::~DistanceVector()
{
}

const Particle *
DistanceVector::QSortReduced(FP mDesired)
{
  int size = pptrs.size();
  return QSortReducedInternal(pptrs, mDesired, 0.0);
}
```

```
// average the distances of `list' between these points
const FP
DistanceVector::averageDistanceInternal(std::list<const Particle *> pp,
int& ppSize)
{
  assert(ppSize > 0);

  FP mTot = 0.0;
  std::list<const Particle *>::const_iterator i;
  for(i=pp.begin(); i!=pp.end(); i++)
    mTot += (*i)->distGet();
  mTot /= ppSize;
  return mTot;
}

// sorts by distance, adding to mFoundAlready, returns particle that
makes mDesired
const Particle *
DistanceVector::QSortReducedInternal(std::list<const Particle *> pp,
FP& mDesired, FP mFoundAlready)
{
  int ppSize = pp.size();

  // most likely terminal point
  if(ppSize == 1)
    return *pp.begin();

  FP pivot = averageDistanceInternal(pp, ppSize);
  const Particle *rvalue;
  int sameAsPivot = 0;
  FP mFoundLeft = 0;

  std::list<const Particle *>::const_iterator i;

  std::list<const Particle *> l(0);
  std::list<const Particle *> r(0);

  for(i=pp.begin(); i!=pp.end(); i++)
    {
      // compare to pivot
      if((*i)->distGet() == pivot)
        sameAsPivot++;

      if((*i)->distGet() < pivot)
        {
          l.push_back(*i);
          mFoundLeft += (*i)->Get_Mass();
        }
      else
        r.push_back(*i);
    }

  // if all particles are same distance!
  // unlikely terminal point
  // TODO: unlikely
  if(ppSize == sameAsPivot)
    return *pp.begin();
```

```
  else if((mFoundAlready + mFoundLeft) > mDesired)
    rvalue = QSortReducedInternal(l, mDesired, mFoundAlready);
  else
    rvalue = QSortReducedInternal(r, mDesired, mFoundAlready +
mFoundLeft);

  return rvalue;
}

void
DistanceVector::printDistances() const
{
  for(std::list<const Particle *>::const_iterator i=pptrs.begin();
i!=pptrs.end(); i++)
    std::cout << d3d( (*i), target) << std::endl;
}
```

distance_vector.hpp

```
// A set of Particle* w/ reduced quicksort algorithm

#ifndef _DISTANCE_VECTOR_H_
#define _DISTANCE_VECTOR_H_

#include <iostream>
#include <iterator>
#include <list>
#include "particle.hpp"

class DistanceVector
{
public:
  DistanceVector();
  ~DistanceVector();
  const Particle* GreedyQuickSort();
  void Set_Target(const Particle *p) {target = p;}
  void Add(Particle* ptr) { pptrs.push_back(ptr); }
  void Add(DistanceVector dv) { pptrs.insert(pptrs.begin(),
dv.pptrs.begin(), dv.pptrs.end()); }
  // void Print() const {std::copy(pptrs.begin(), pptrs.end(),
std::ostream_iterator<const Particle*>(std::cout," "));}
  void print() const {for(std::list<const Particle *>::const_iterator
i=pptrs.begin(); i!=pptrs.end(); i++) (*i)->Coords_Print();}
  void printDistances() const;
  void printSize() const {std::cout << pptrs.size() << std::endl;}
  const Particle * QSortReduced(FP mDesired);
  // const Particle * QSortReduced(int answer);
private:
  const FP averageDistanceInternal(std::list<const Particle *> pp, int&
ppSize);
  // const Particle * QSortReducedInternal(std::list<const Particle *>
pptrs, int answer, int l, int r);
  const Particle * QSortReducedInternal(std::list<const Particle *> pp,
FP& mDesired, FP mFoundAlready);
  const Particle *target;
  std::list<const Particle *> pptrs;
```

```
};

#endif
```

htree.cpp

```
#include "htree.hpp"
#include "system.hpp"
#include "kernel.hpp"

// constants
const FP PRUNE_K = 2.0;            // ~10 more leaves than particles

// macros
#define SUBCUBE_NDX(X,Y,Z) ((X) + 2*(Y) + 4*(Z))

HTree::HTree()
{
  _subdivided = 0;

  for(int i=0; i<3; i++)
    {
      _min[i] = 0;
      _max[i] = 0;
      _mid[i] = 0;
    }
  _particle = 0;
  _subcubes[0] = 0;
  _parent = NULL;
  _pruneme = 1;                    // mark prunable
  _h_avr = 0;
  _h_total = 0;
  _n_particles = 0;
}

HTree::~HTree()
{
  // death to trees and children
  int i;

  if(_subdivided)
    for(i=0; i<8; i++)
      delete _subcubes[i];
}

void
HTree::reset()
{
    _subdivided = 0;

    for(int i=0; i<3; i++)
      {
        _min[i] = 0;
        _max[i] = 0;
        _mid[i] = 0;
      }
```

```
    _particle = 0;
    _subcubes[0] = 0;


    destroy_branches();
}

void
HTree::destroy_branches()
{
  if(_subdivided)
    {
      int i;
      for(i=0; i<8; i++)
        {
          _subcubes[i]->destroy_branches();
          delete _subcubes[i];
        }
    }
}


HTree *
HTree::htree_choose_subdivision(const Particle *p) const
{
  assert(_subcubes[0] != NULL);

  int i;
  bool cmps[3] = {0,0,0};

  if (p->x_get() > _mid[0])
    cmps[0] = 1;
  if (p->y_get() > _mid[1])
    cmps[1] = 1;
  if (p->z_get() > _mid[2])
    cmps[2] = 1;

  i = SUBCUBE_NDX(cmps[0], cmps[1], cmps[2]);
  // std::cout << "Chose subdivision: " << i << std::endl;
  return _subcubes[i];
}

void
HTree::subdivide()
{
  // make sure we aren't already _subdivided and there is no particle
in square that will be lost
  assert(_subdivided == 0);
  assert(_particle == NULL);

  int i,j,k,ndx;

  // throw the flag
  _subdivided = 1;

  // set the _midpoints
  for(int d=0; d<3; d++)
```

```
    _mid[d] = FP_AVR(_min[d], _max[d]);

  // Boundaries_Print();
  // std::cout << x__min << " "
  //           << x__mid << " "
  //           << x__max << std::endl;

  // literally i=x, j=y, k=z
  for(i=0; i<2; i++)
    for(j=0; j<2; j++)
      for(k=0; k<2; k++)
        {
          ndx = SUBCUBE_NDX(i, j, k);
          _subcubes[ndx] = new HTree;

          if(i==0)
            {
              _subcubes[ndx]->_min[0] = _min[0];
              _subcubes[ndx]->_max[0] = _mid[0];
            }
          else
            {
              _subcubes[ndx]->_min[0] = _mid[0];
              _subcubes[ndx]->_max[0] = _max[0];
            }

          if(j==0)
            {
              _subcubes[ndx]->_min[1] = _min[1];
              _subcubes[ndx]->_max[1] = _mid[1];
            }
          else
            {
              _subcubes[ndx]->_min[1] = _mid[1];
              _subcubes[ndx]->_max[1] = _max[1];
            }
          if(k==0)
            {
              _subcubes[ndx]->_min[2] = _min[2];
              _subcubes[ndx]->_max[2] = _mid[2];
            }
          else
            {
              _subcubes[ndx]->_min[2] = _mid[2];
              _subcubes[ndx]->_max[2] = _max[2];
            }
        }
  for(i=0; i<8; i++)
    _subcubes[i]->set_parent(this);
}

void
HTree::particle_insert(Particle *p)
{
  HTree *subcube = NULL;
  Particle *lost_child = NULL;
```

```
  if(_subdivided)
    {
      subcube = htree_choose_subdivision(p);
      subcube->particle_insert(p);
    }
  else
    {
      if(_particle == NULL)      // nobody else here
        {
          // std::cout << "success\n";
          particle_set(p);      // this
          return;
        }
      // other particle is here
      assert(_particle != NULL);

      // if we don't check to see if they're in the same place then
it's going to keep subdividing
      // forever.  if this is the case, set one to NULL and the other
to the combined mass of the
      // two

      if( *_particle == *p)
        {
          DEBUG_MSG("HTree: Particles collided, IGNORED\n");
          // particle->Add_Mass(p->Get_Mass());
          // p->Disable();
          // return;               // our work is done
        }
      lost_child = _particle;         // find this a home
      _particle = NULL;               // no more particle in subdiv
cube

      subdivide();              // throw flag
      subcube = htree_choose_subdivision(lost_child);
      // debug
      // std::cout << subcube->Print__Subdivided() << std::endl;
      subcube->particle_insert(lost_child);
      subcube = htree_choose_subdivision(p);
      subcube->particle_insert(p);
    }
}

void
HTree::boundaries_extend(FP pnew[3])
{
  // FP new[3] = {x,y,z};
  FP old[3];
  FP add = 0;

  for(int i=0; i<3; i++)
    old[i] = _max[i] - _min[i];

  for(int i=0; i<3; i++)
    {
      if(pnew[i] < _min[i])
        {
```

```
          if((_min[i] - pnew[i]) > add)
            add = (_min[i] - pnew[i]);
          _min[i] = pnew[i];
        }
      if(pnew[i] > _max[i])
        {
          if((pnew[i] - _max[i]) > add)
            add = (pnew[i] - _max[i]);
          _max[i] = pnew[i];
        }
    }

  // add now holds the greatest change in size
  // compare current to old sizes and make sure the change adds up to
`add'
  FP todo[3];
  for(int i=0; i<3; i++)
    {
      todo[i] = 0.5 * (add = ((_max[i] - _min[i]) - old[i]));
      _min[i] -= todo[i];
      _max[i] += todo[i];
    }
}

void
HTree::boundaries_print() const
{
  for(int i=0; i<3; i++)
    std::cout << _min[i] << " " << _mid[i] << " " << _max[i] <<
std::endl;
}

// set a particle for this tree's cube
void
HTree::particle_set(Particle *p)
{
  assert(_particle == NULL);
  _particle = p;
}

void
HTree::smoothing_lengths_update()
{
  int i;
  _h_total = 0;                     // refind h
  _h_avr = 0;
  _n_particles = 0;
  // single case
  if (_particle != NULL) {
    _h_total += _particle->hGet();
    _n_particles++;
  }

  // recurse to children, if they exist
  else if(_subdivided) {
    for(i=0; i<8; i++) {
      _subcubes[i]->smoothing_lengths_update();
```

```
      _n_particles += _subcubes[i]->_n_particles;
      _h_total += _subcubes[i]->_h_total;
    }
  }

  // make this an avr.
  if( _n_particles != 0.0)
    _h_avr = _h_total / _n_particles;

  // set this in the leaf now
  // _n_particles = n_particles;

  // now see if this should be pruned
  if(!(_h_avr > (PRUNE_K * (_max[0] - _min[0]))))
    do_not_prune();

  // return _h_avr;
}

// run through parents and turn off the _pruneme
void
HTree::do_not_prune()
{
  _pruneme = 0;
  if(_parent != NULL)
    _parent->do_not_prune();
}

void
HTree::prune_tree()
{
  // see if subcubes need pruning
  // NOTE NOTE NOTE if not done in this order, the destructor will beat
us to it
  if(_subdivided)
    {
      for(int i=0; i<8; i++)
        _subcubes[i]->prune_tree();
    }
  // this points to the old tree, ready to be pruned
  if(_pruneme)
    {
      if(_particle!=NULL)
        _particle->Disable();
      if(_subdivided)
        {
          _subdivided = 0;
          for(int i=0; i<8; i++)
            delete _subcubes[i];
        }
    }
}

// update the density of particle i
// *particle list is infeasible in large sims*
// this is a N traversal algorithm
void
```

```
HTree::density_energy_update()
{
  density_energy_sum(_h_avr);
  _density = _density_sum;
  _energy = _energy_sum / _density_sum;

  if(_subdivided)
    {
      for(int i=0; i<8; i++)
        {
          _subcubes[i]->density_energy_update();
        }
    }

  // if(_density != 0)
  //   std::cout << _density << "\t" << _energy << "\n";
}

void
HTree::density_energy_sum(FP hg)
{
  int i;
  _density_sum = 0;
  _energy_sum = 0;

  // single case
  if ((_particle != NULL) && (_particle->Is_Active())) {
    FP rig = symSmoothLens(_particle->hGet(), hg);
    _density_sum += (_particle->Get_Mass() * (kernel(rig, _particle-
>hGet())));
    _energy_sum += (_particle->energy_get() * _density_sum);
  }

  // recurse to children, if they exist
  else if(_subdivided) {
    for(i=0; i<8; i++) {
      _subcubes[i]->density_energy_sum(hg);
      _density_sum += _subcubes[i]->_density_sum;
      _energy_sum += _subcubes[i]->_energy_sum;
    }
  }
}

void
HTree::density_traverse(Particle *p)
{
  int i;

  // single case
  if ((_particle != NULL) && (_particle->Is_Active()))
    {
      FP rij = d3d(_particle, p);
      FP hij = symSmoothLens(p->hGet(), _particle->hGet());
      if((rij < 2*p->hGet()) && (rij < 2*hij))
        {
          p->densitySet((p->densityGet() + _particle-
>Get_Mass()*kernel(rij,hij)));
```

```
          // add contribution to j, _particles will not interact again
          if(!(rij < 2*_particle->hGet()))
            _particle->densitySet( _particle->densityGet() + p-
>Get_Mass()*kernel(rij,hij));
        }
    }
  // recurse to children
  else if(_subdivided)
    {
      for(i=0; i<8; i++)
        {
          _subcubes[i]->density_traverse(p);
        }
    }
}
```

htree.hpp

```
// Radiative Energy Transfer Tree
// this is an gutted gravity tree w/ some differences
// note: making a generic tree separate from data became annoying,
since all methods really assume
// the two are joined

#ifndef _HTREE_HPP_
#define _HTREE_HPP_

#include <iostream>
#include <cstdlib>
#include "tree.hpp"

class HTree
{
public:
  HTree();                       // main constructor
  HTree(Tree *head);           // add in cache data from gravity tree
  ~HTree();
  void reset();
  void subdivide();
  void particle_insert(Particle *p);
  void boundaries_extend(FP pnew[3]);
  void boundaries_print() const;
  void particle_set(Particle *p);
  void prune_tree();
  void smoothing_lengths_update();
  void density_energy_update();
  void density_traverse(Particle *p);
private:
  HTree* htree_choose_subdivision(const Particle *p) const;
  void destroy_branches();
  void density_energy_sum(FP hg);
  void set_parent(HTree *t) { _parent = t; }
  void do_not_prune();
  FP h_total() { return _h_total; }
  bool _subdivided;
  FP _min[3];
```

```
  FP _max[3];
  FP _mid[3];
  HTree *_parent;
  HTree *_subcubes[8];
  Particle *_particle;
  int _n_particles;     // number of particles inside
  bool _pruneme;
  FP _h_total;              // leaf total smoothing length
  FP _h_avr;                // leaf average smoothign length
  FP _density_sum;          // leaf density sum of particles
  FP _energy_sum;           // leaf energy sum of particles
  FP _density;
  FP _energy;
};

#endif
```

hydro.cpp

```
// Hydrodynamics stuff
```

hydro.hpp

```
// hydrodynamics
// mjs
#ifndef _HYDRO_HPP_
#define _HYDRO_HPP_

// control artificial viscosity strength
const FP alpha = 0.5;
const FP beta = 1.0;
// prevents divergence when particles are close together
const FP nu = 0.01;

#endif
```

integrate.cpp

```
// Euler Method, 1st order integration
#include <iostream>
#include <iomanip>
#include <fstream>
#include "system.hpp"

// integrate the accelerations and velocities
void
System::Integrate()
{
  int i;
  FP r_total = 0;
  FP r_iter = 0;
  FOR_ALL_PARTICLES(i)
    {
      if(!(ps[i].Is_Active()))
        continue;
      ps[i].integrate(t_step, use_accel_checking);
      ps[i].check_most_massive();
```

```
    }
   ps[0].print_most_massive();
   FOR_ALL_PARTICLES(i)
   {
    FP rval = ps[i].test_vs_protostar();
    if(rval != -1)
    {
      r_iter++;
      r_total += rval;

    }
   }

   std::ofstream out("./output/percent_regression.dat", std::ios::app);
   out << r_total / r_iter << "\n";
}

void
Particle::integrate(const FP tstep, const bool& accel_check)
{
  int i;
  for(i=0; i<3; i++)
    {
      // v[i] += (tstep * a[i]);
      v[i] += (tstep * a[i]);
    }
  if(accel_check)
    {
      // std::cout << a[i] << std::endl;
      FP xcheck, ycheck, zcheck;
      xcheck = 100*(a[0]-a_check[0])/a_check[0];
      ycheck = 100*(a[1]-a_check[1])/a_check[1];
      zcheck = 100*(a[2]-a_check[2])/a_check[2];
      // std::cout << std::setprecision(1);
      std::cout << "Error %:" << xcheck << "\t" << ycheck << "\t" <<
zcheck << std::endl;
    }

  x += (v[0] * tstep);
  y += (v[1] * tstep);
  z += (v[2] * tstep);

  #ifdef ENABLE_BOUNDS
  if(x < -BOUND || x > BOUND)
       Disable();
  if(y < -BOUND || y > BOUND)
       Disable();
  if(z < -BOUND || z > BOUND)
       Disable();
  #endif
}
```

kernel.cpp

```
// Kernel/W, expands particle into a spatially spread model
#include <cmath>
#include "maths.hpp"
```

```
// for three dimensions
static const FP norm_const = 1.0/PI;
static const FP dim = 3.0;

// Monaghan and Lattanzio method (1985) (B-spline)
FP
kernel(FP r, FP h)
{
  FP u = r/h;
  if(u >= 2.0)
    return 0.0;

  FP coeff = (norm_const / pow(h, dim));

  if (u >= 1.0)
    return coeff * ((1/4) * pow(2.0 - u, 4/3));
  else                          // 0<=u<1
    return coeff * (1 - ((3/2) * pow(u,2)) + ((3/4) * pow(u,3)));
}

// gravitational kernel
// particle/sphere interaction, integrating out of sphere
// contains appropriate multiples to work in the equation
G*M*gkernel*r_vector
FP
gKernel(const FP r, const FP h)
{
  FP u = r/h;
  if(u <= 1)
    return ((1.0/pow(h,3.0))*((4.0/3.0) - (6.0/5.0)*pow(u,2.0) +
(1.0/2.0)*pow(u,3.0)));
  else if (u <= 2)
    return ((1.0/pow(r,3.0))*((-1.0/15.0) + (8.0/3.0)*pow(u,3.0) -
3.0*pow(u,4.0) + (6.0/5.0)*pow(u,5.0) - (1.0/6.0)*pow(u,6.0) ));
  else // u > 2
    return 1.0/pow(r, 3.0);
}
```

kernel.hpp

```
#ifndef _KERNEL_HPP_
#define _KERNEL_HPP_

FP kernel(FP r, FP h);
FP gKernel(const FP r, const FP h);


#endif
```

mac.cpp

```
// Multipole Acceptibility Criteria
// decide whether to use particle-particle or particle-cube interaction
#include "tree.hpp"
#include "particle.hpp"
#include "system.hpp"
```

```
#include <cmath>

const FP theta = 1.0 / pow(3.0, 0.5);

// barnes-hut mac
// see if tree's side / r < theta
bool
Tree::BHMac(const Particle *p)
{
  FP s = x_max - x_min;
  FP r = d3d(p, &cm);

  if( (s/r) < theta)
    return true;
  else
    return false;
}

// test to see if tree's bounds are w/in particle's box of side length
k
// this is approximate a 2h sphere.
bool
Tree::SPHMac(const FP& k, const Particle *p_i)
{
  FP k2 = k/2.0;
  FP x_min_box = p_i->x_get() - k2;
  FP x_max_box = p_i->x_get() + k2;
  FP y_min_box = p_i->y_get() - k2;
  FP y_max_box = p_i->y_get() + k2;
  FP z_min_box = p_i->z_get() - k2;
  FP z_max_box = p_i->z_get() + k2;

  // case 1: box is larger than tree cube
  if((x_min_box < x_min) && (x_max_box > x_max) &&
     (y_min_box < y_min) && (y_max_box > y_max) &&
     (z_min_box < z_min) && (z_max_box > z_max))
    return true;
  // case 2: one of the sides of the box overlaps into tree cube
  if(((x_min < x_max_box) && (x_min > x_min_box)) || ((x_max <
x_max_box) && (x_max > x_min_box)))
    if(((y_min < y_max_box) && (y_min > y_min_box)) || ((y_max <
y_max_box) && (y_max > y_min_box)))
      if(((z_min < z_max_box) && (z_min > z_min_box)) || ((z_max <
z_max_box) && (z_max > z_min_box)))
        return true;
  return false;
}

// if( fabs(k * p_i->x_get() - x_min) > 0.0)
//   if( fabs(k * p_i->y_get() - y_min) > 0.0)
//     if( fabs(k * p_i->z_get() - z_min) > 0.0)
//       return true;
// return false;
```

mac.hpp

```
#ifndef _MAC_HPP_
```

```
#define _MAC_HPP_

#include "tree.hpp"
#include "particle.hpp"



#endif
```

main.cpp

```
// summary:
// read in settings from `./input.dat'
// output data to `./output.dat'
// if you really need to change it go to data.h
#include "system.hpp"

int
main(int argc, char* argv[])
{
  System world;

  world.Data_Read();
  world.dataWriteBegin();
  // world.Properties_Assign_Init();
  while( world.Time_Update() )
    {
      world.treeCreate();
      world.CenterMass_Update();
      world.Smoothing_Lengths_Update();
      world.Densities_Update();
      world.htree_create();
      world.internal_energies_calc();
      // world.Temperatures_Update();
      world.Pressures_Update();
      world.accelsCalc();
      world.Integrate();
      world.Data_Write();
    }
  // debugging stuff
  // world.Debug();
  world.dataWriteEnd();
  return 0;
}
```

maths.hpp

```
#ifndef _MATHS_H_
#define _MATHS_H_

#include <cmath>

// define floating point precision
#define FP double
#define PI 3.14159265358979323846
#define G 6.67300e-11
#define FASTDIV2(X) ((X)<<1)
```

```
#define FP_AVR(X,Y) (((X)+(Y))/(2.0))
// constantly reproduceable seed
#define SEED 0
// high bits random
#define RAND_HI(L,H) ((((((FP)H) - ((FP)L)) * rand()) / (RAND_MAX+1.0))
+ ((FP)L))

#endif

// deprecated
#if 0
#define BINARY_ITER(I,N) (((int)(I/N))%N)
#endif
```

opacity.cpp

```
// opacity calcs for nebula gases

#include "maths.hpp"
#include "debug.hpp"
#include <iostream>

#define ROWS 27
#define COLS 20

FP log_R_lookup[COLS] = {-4.0, -3.5, -3.0, -2.5, -2.0, -1.5, -1.0, -
0.5, 0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5};
FP log_T_lookup[ROWS] = {4.10, 4.05, 4.00, 3.95, 3.90, 3.85, 3.80,
3.75, 3.70, 3.65, 3.60, 3.55, 3.50, 3.45, 3.40, 3.35, 3.30, 3.25, 3.20,
3.15, 3.10, 3.05, 3.00, 2.95, 2.90, 2.85, 2.80};

FP opacity_table[ROWS][COLS] = {
  {-1.493, -1.479, -1.439, -1.342, -1.147, -0.840, -0.430, 0.023,
0.476, 0.887, 1.217, 1.453, 1.618, 1.742, 1.851, 1.966, 2.096, 2.228,
2.374, 2.527},
  {-1.509, -1.497, -1.447, -1.325, -1.097, -0.767, -0.350, 0.078,
0.467, 0.773, 0.987, 1.135, 1.247, 1.344, 1.447, 1.570, 1.717, 1.885,
2.056, 2.237},
  {-1.529, -1.498, -1.429, -1.285, -1.054, -0.729, -0.354, -0.009,
0.257, 0.441, 0.571, 0.674, 0.768, 0.871, 0.991, 1.137, 1.311, 1.517,
1.721, 1.927},
  {-1.521, -1.482, -1.405, -1.267, -1.062, -0.802, -0.553, -0.357, -
0.216, -0.107, -0.010, 0.092, 0.205, 0.337, 0.490, 0.671, 0.880, 1.115,
1.351, 1.589},
  {-1.510, -1.475, -1.414, -1.326, -1.223, -1.126, -1.046, -0.971, -
0.892, -0.799, -0.687, -0.554, -0.402, -0.230, -0.037, 0.182, 0.425,
0.676, 0.948, 1.222},
  {-1.524, -1.527, -1.546, -1.593, -1.657, -1.713, -1.742, -1.725, -
1.655, -1.539, -1.387, -1.210, -1.016, -0.804, -0.575, -0.326, -0.048,
0.231, 0.544, 0.857},
  {-1.669, -1.790, -1.943, -2.111, -2.275, -2.412, -2.496, -2.500, -
2.418, -2.268, -2.075, -1.862, -1.631, -1.386, -1.120, -0.826, -0.500,
-0.154, 0.207, 0.541},
  {-2.112, -2.321, -2.537, -2.751, -2.952, -3.122, -3.229, -3.241, -
3.147, -2.978, -2.757, -2.509, -2.235, -1.933, -1.593, -1.213, -0.819,
-0.437, -0.087, 0.225},
```

```
  {-2.765, -2.997, -3.224, -3.443, -3.644, -3.812, -3.921, -3.932, -
3.836, -3.647, -3.387, -3.067, -2.690, -2.273, -1.851, -1.450, -1.078,
-0.751, -0.441, -0.160},
  {-3.522, -3.742, -3.951, -4.147, -4.323, -4.465, -4.546, -4.530, -
4.387, -4.113, -3.735, -3.307, -2.875, -2.469, -2.098, -1.759, -1.446,
-1.176, -0.892, -0.628},
  {-4.297, -4.478, -4.642, -4.783, -4.889, -4.939, -4.912, -4.783, -
4.545, -4.220, -3.853, -3.485, -3.136, -2.814, -2.509, -2.214, -1.926,
-1.699, -1.440, -1.218},
  {-4.940, -5.033, -5.086, -5.100, -5.080, -5.030, -4.945, -4.817, -
4.640, -4.420, -4.167, -3.896, -3.617, -3.330, -3.027, -2.718, -2.420,
-2.333, -2.127, -1.958},
  {-5.182, -5.169, -5.147, -5.121, -5.102, -5.094, -5.086, -5.059, -
4.996, -4.883, -4.695, -4.413, -4.019, -3.521, -2.979, -2.498, -2.169,
-2.169, -2.169, -2.169},
  {-5.203, -5.207, -5.231, -5.274, -5.333, -5.404, -5.469, -5.503, -
5.465, -5.279, -4.870, -4.270, -3.608, -3.014, -2.596, -2.363, -2.245,
-2.245, -2.245, -2.245},
  {-5.387, -5.460, -5.550, -5.650, -5.745, -5.812, -5.818, -5.688, -
5.319, -4.700, -3.990, -3.351, -2.885, -2.625, -2.507, -2.454, -2.428,
-2.428, -2.428, -2.428},
  {-5.796, -5.896, -5.971, -6.004, -5.974, -5.832, -5.479, -4.881, -
4.174, -3.529, -3.055, -2.802, -2.698, -2.658, -2.641, -2.631, -2.623,
-2.623, -2.623, -2.623},
  {-6.133, -6.117, -6.060, -5.894, -5.515, -4.903, -4.194, -3.559, -
3.134, -2.939, -2.869, -2.844, -2.834, -2.828, -2.825, -2.822, -2.820,
-2.820, -2.820, -2.820},
  {-6.131, -5.888, -5.409, -4.724, -4.012, -3.463, -3.186, -3.088, -
3.057, -3.045, -3.040, -3.037, -3.036, -3.034, -3.033, -3.032, -3.029,
-3.029, -3.029, -3.029},
  {-5.011, -4.281, -3.702, -3.420, -3.329, -3.301, -3.291, -3.288, -
3.285, -3.284, -3.284, -3.285, -3.285, -3.284, -3.284, -3.280, -1.881,
-1.881, -1.881, -1.881},
  {-3.496, -3.448, -3.434, -3.429, -3.428, -3.427, -3.427, -3.427, -
3.428, -3.428, -3.429, -2.133, -1.936, -1.756, -1.653, -0.974, -0.587,
-0.587, -0.587, -0.587},
  {-3.423, -3.423, -3.423, -3.424, -3.424, -3.425, -2.346, -2.092, -
1.862, -1.269, -0.940, -0.857, -0.646, -0.495, -0.407, -0.374, -0.343,
-0.343, -0.343, -0.343},
  {-3.427, -3.427, -2.203, -1.957, -1.479, -1.045, -0.844, -0.743, -
0.649, -0.551, -0.468, -0.431, -0.406, -0.401, -0.400, -0.399, -0.399,
-0.399, -0.399, -0.399},
  {-1.263, -1.091, -0.889, -0.755, -0.671, -0.586, -0.512, -0.471, -
0.454, -0.453, -0.452, -0.452, -0.452, -0.452, -0.452, -0.452, -0.452,
-0.452, -0.452, -0.452},
  {-0.746, -0.680, -0.602, -0.534, -0.501, -0.499, -0.498, -0.498, -
0.498, -0.498, -0.498, -0.498, -0.498, -0.498, -0.498, -0.498, -0.498,
-0.498, -0.498, -0.498},
  {-0.577, -0.542, -0.540, -0.539, -0.538, -0.538, -0.538, -0.538, -
0.538, -0.538, -0.538, -0.538, -0.538, -0.538, -0.538, -0.538, -0.538,
-0.538, -0.538, -0.538},
  {-0.571, -0.570, -0.570, -0.570, -0.570, -0.570, -0.570, -0.570, -
0.570, -0.570, -0.570, -0.570, -0.570, -0.570, -0.570, -0.570, -0.570,
-0.570, -0.570, -0.570},
  {-0.590, -0.590, -0.590, -0.590, -0.590, -0.590, -0.590, -0.590, -
0.590, -0.590, -0.590, -0.590, -0.590, -0.590, -0.590, -0.590, -0.590,
-0.590, -0.590, -0.590},
```

```
};

FP
opacity_lookup_high(FP T, FP R)
{
  int i,j;
  FP logR = log(R);
  FP logT = log(T);

  if(logR < log_R_lookup[0])
    FATAL("opacity.cpp: R value less than lookup table\n");
  if(logR > log_R_lookup[COLS-1])
    FATAL("opacity.cpp: R value greater than lookup table\n");

  for(i=1; i<COLS; i++)
    {
      if(logR < log_R_lookup[i])
        break;
    }

  FP col_ratio = (logR - log_R_lookup[i-1]) / (log_R_lookup[i] -
log_R_lookup[i-1]);

  if(logT < log_T_lookup[0])
    FATAL("opacity.cpp: T value less than lookup table\n");
  if(logT > log_T_lookup[ROWS-1])
    FATAL("opacity.cpp: T value greater than lookup table\n");

  for(j=1; j<ROWS; j++)
    {
      if(logT < log_T_lookup[j])
        break;
    }
  FP row_ratio = (logT - log_T_lookup[j-1]) / (log_T_lookup[j] -
log_T_lookup[j-1]);

  std::cout << col_ratio << " " << row_ratio << "\n";

  FATAL("High temp Ross-values not implemented.  Do so in
opacity.cpp\n");
}

// LOW TEMPS
// Henning & Stognienko data, this is a good tabbed dataset
// taken from
// http://www.mpia-
hd.mpg.de/homes/henning/Dust_opacities/Opacities/Ralf/Pol/all_spec.comp
.Gofn
#define LOW_LOOKUP_SIZE 86
FP T_lookup[LOW_LOOKUP_SIZE] = {10.0000, 20.0000, 30.0000, 40.0000,
50.0000, 60.0000, 70.0000, 80.0000, 90.0000, 100.000, 125.000, 150.000,
170.000, 171.000, 172.000, 173.000, 174.000, 175.000, 200.000, 225.000,
250.000, 275.000, 300.000, 325.000, 350.000, 373.000, 374.000, 375.000,
376.000, 377.000, 400.000, 425.000, 450.000, 475.000, 500.000, 525.000,
550.000, 573.000, 574.000, 575.000, 576.000, 577.000, 600.000, 625.000,
650.000, 675.000, 678.000, 679.000, 680.000, 681.000, 682.000, 700.000,
725.000, 750.000, 775.000, 800.000, 825.000, 850.000, 875.000, 900.000,
```

```
925.000, 950.000, 975.000, 1000.00, 1050.00, 1100.00, 1150.00, 1200.00,
1250.00, 1300.00, 1329.00, 1330.00, 1331.00, 1332.00, 1333.00, 1350.00,
1393.00, 1394.00, 1395.00, 1396.00, 1397.00, 1400.00, 1406.00, 1407.00,
1408.00, 1408.00};

FP kappa_lookup[LOW_LOOKUP_SIZE] = {0.0224760, 0.0871040, 0.192000,
0.335260, 0.515700, 0.730740, 0.976460, 1.24850, 1.54260, 1.85480,
2.69310, 3.57250, 4.26910, 4.30320, 4.33720, 3.32610, 3.35370, 3.38120,
4.04460, 4.64230, 5.16300, 5.60990, 5.99500, 6.33280, 6.63730, 6.89780,
6.90880, 6.91970, 5.95230, 5.96150, 6.16840, 6.38780, 6.60530, 6.82310,
7.04260, 7.26460, 7.48920, 7.69800, 7.70710, 7.71630, 2.33590, 2.33930,
2.41770, 2.50540, 2.59530, 2.68710, 2.69830, 2.70200, 2.70570, 2.20180,
2.20490, 2.26080, 2.33960, 2.41940, 2.50000, 2.58130, 2.66290, 2.74480,
2.82670, 2.90860, 2.99040, 3.07180, 3.15290, 3.23350, 3.39310, 3.55010,
3.70440, 3.85580, 4.00420, 4.14960, 4.23260, 4.23550, 4.23830, 3.76660,
3.76930, 3.81510, 3.92980, 3.93240, 3.93510, 1.57370, 1.57450, 1.57690,
1.58160, 1.58240, 1.58310, 0.00000};

FP
opacity_lookup_low(FP T)
{
  int i;
  if(T < T_lookup[0])
    {
      FATAL("opacity.cpp: kappa not defined this low, get more
data\n");
    }
  if(T > T_lookup[LOW_LOOKUP_SIZE-1])
    {
      FATAL("opacity.cpp: should've use other method, something's
wrong\n");
    }
  for(i=0; i<LOW_LOOKUP_SIZE; i++)
    {
      if(T < T_lookup[i])
        break;
    }
  FP ratio = (T - T_lookup[i-1]) / (T_lookup[i] - T_lookup[i-1]);

  return (kappa_lookup[i-1] + (ratio * (kappa_lookup[i+1] -
kappa_lookup[i])));
}
```

opacity.hpp

```
#ifndef _OPACITY_HPP_
#define _OPACITY_HPP_

FP opacity_lookup_high(FP T, FP R);
FP opacity_lookup_low(FP T);

#endif
```

particle.cpp

```
#include "particle.hpp"
#include "system.hpp"
```

```
#include "opacity.hpp"

//protostar
Particle* __protostar;

Particle::Particle()
{
  active = 1;
  x=0;
  y=0;
  z=0;
  m=1;
  density=1;
  P=1;
  T=100;
  h=1;
  gh=100;
  a[0]=0;
  a[1]=0;
  a[2]=0;
  a_check[0]=0;
  a_check[1]=0;
  a_check[2]=0;
  v[0]=0;
  v[1]=0;
  v[2]=0;
  u=0;
  opacity = 0;
}

bool
operator==(const Particle &a, const Particle &b)
{
  assert(a.active && b.active);
  if(d3d(&a,&b) <= PARTICLE_PROX)
    return true;
  else
    return false;
}

void
Particle::distSet(const Particle *p)
{
  dist = d3dsquared(*this, *p);
}

void
Particle::a_check_add(const FP ax, const FP ay, const FP az)
{
  a_check[0] += ax;
  a_check[1] += ay;
  a_check[2] += az;
}

bool
Particle::temp_low()
{
```

```
  return ((T < 1408)?1:0);
}

// temps shouldn't be getting this low
void
Particle::opacity_calc_low_temp()
{
  opacity = opacity_lookup_low(T);
}

void
Particle::opacity_calc_high_temp()
{
  FP R = (density / pow(T/1.0e6,3));
  opacity = opacity_lookup_high(T,R);
}

void
Particle::check_most_massive() const
{
  if(__protostar == NULL)
     __protostar = (Particle *)this;
  else if(m > __protostar->Get_Mass())
     __protostar = (Particle *)this;
}

FP
Particle::test_vs_protostar() const
{
  if(!Is_Active() || !__protostar->Is_Active())
     return -1;
  FP r_vs = d3d(this, __protostar);
  FP v_towards[3], v_projected[3];
  rvect(v_towards, 1, this, __protostar);

  project(v_projected, v_towards, v);

  FP v_mag_squared =
        v_projected[0]*v_projected[0] +
        v_projected[1]*v_projected[1] +
        v_projected[2]*v_projected[2];
  FP a_mag = sqrt(a[0]*a[0] +
                  a[1]*a[1] +
                  a[2]*a[2]);

  if(r_vs == 0)
    return -1;
  FP a_vs = v_mag_squared / r_vs;
  if(a_vs == 0)
    return -1;
  return (fabs(a_mag - a_vs) / a_vs);
}

void
Particle::print_most_massive() const
{
  __protostar->Coords_Print();
```

```
}
```

particle.hpp

```
#ifndef _PARTICLE_HPP_
#define _PARTICLE_HPP_

#include <iostream>
#include <cassert>
#include "maths.hpp"
#include "debug.hpp"

class Particle
{
public:
  Particle();
  void print_most_massive() const;
  FP test_vs_protostar() const;
  void check_most_massive() const;
  bool temp_low();
  void opacity_calc_low_temp();
  void opacity_calc_high_temp();
  FP energy_get() { assert(active); return u; }
  void integrate(const FP tstep, const bool& accel_check);
  FP aGet(const int d) const {return a[d];}
  void aSub(const FP pa[]) { assert(active); a[0] -= pa[0]; a[1] -=
pa[1]; a[2] -= pa[2];}
  void aAdd(const FP pa[]) { assert(active); a[0] += pa[0]; a[1] +=
pa[1]; a[2] += pa[2];}
  void a_check_add(const FP x, const FP y, const FP z);
  void densitySet(const FP& dp) {assert(active); density = dp;}
  FP densityGet() const {assert(active); return density;}
  void distSet(const Particle *p);
  FP distGet() const { return dist;}
  void Coords_Print() const {std::cout << x << " " << y << " " << z <<
"\n";}
  void Set_Pressure(const FP pressure){assert(active); P = pressure;}
  FP Get_Pressure() const {assert(active); return P;}
  void Set_Mass(const FP mass){assert(active); m = mass;}
  void Add_Mass(const FP mass){assert(active); m += mass;}
  FP Get_Mass() const {return m;}
  void Get_Pos(FP &i, FP &j, FP &k) const {assert(active); i=x; j=y;
k=z;}
  void Set_Pos(const FP i, const FP j, const FP k)  {assert(active);
x=i; y=j; z=k;}
  // only for center of mass stuff
  void Add_Pos(const FP i, const FP j, const FP k)  {assert(active);
x=i; y=j; z=k;}
  void Div_Pos(const FP dvsr) {assert(active); x/=dvsr; y/=dvsr;
z/=dvsr;}
  FP hGet() const { assert(active); return h; }
  FP ghGet() const { assert(active); return gh; }
  void Set_Smoothing_Length(const FP dh){ assert(active); h = dh; }
  void gSmoothingLengthSet(const FP dgh){ assert(active); gh = dgh; }
  // void Disable() { assert(active); active = 0; std::cout <<
"Particle disabled\n";}
```

```
  void Disable() { active = 0; DEBUG_MSG("Particle disabled\n");}
  void enable() { active = 1; }
  bool Is_Active() const { return active; }
  FP x_get() const {assert(active); return x;}
  FP y_get() const {assert(active); return y;}
  FP z_get() const {assert(active); return z;}
  void Set_Opacity(const FP o) { opacity = o; }
  FP Get_Opacity() const { return opacity; }
  friend bool operator==(const Particle &a, const Particle &b);
private:
  bool active;          // flag to turn off particles that have
collided
  FP x,y,z;       // position
  FP m;                 // mass
  FP density;
  FP P;                 // pressure
  FP T;                 // temperature
  FP h;                 // smoothing length
  FP gh;                // gravitational smoothing/softening length
  FP dist;              // distance, used in h calculation
  FP a[3];              // acceleration
  FP a_check[3];        // acccleration checking
  FP v[3];              // velocity
  FP u;                 // internal energy
  FP opacity;
};

#endif
```

pressure.cpp

```
#include "system.hpp"

#define SOUND_SPEED 1.0
#define SHOCK_X     -100
#define SHOCK_Y     -100
#define SHOCK_Z     -100

#define SHOCK_MAX   1000000
#define SHOCK_K     SHOCK_MAX/10.0
#define SHOCK_DECAY 5500

Particle shock_particle;

void
System::Pressures_Update()
{
  shock_particle.Set_Pos(SHOCK_X,SHOCK_Y,SHOCK_Z);
  int i,j;
  FP specific_heats_ratio;
  FOR_ALL_PARTICLES(i)
  {
     if(!ps[i].Is_Active())
         continue;
     FP p; //pressure
     FP dist = d3d(&ps[i], &shock_particle);
     FP shock_dist = t_elapsed*SOUND_SPEED;
```

```
      FP diff = fabs(dist - shock_dist);
      p = SHOCK_MAX - diff*SHOCK_K - t_elapsed*SHOCK_DECAY;
      if (p<0)
         p = 0;
      ps[i].Set_Pressure(p);
      //std::cout << diff << " " << p << "\n";
      //std::cout << shock_dist << "\n";
  }
}
```

smooth_length.cpp

```
// Sets variable smoothing length info for all particles
// This usually isn't done, but it's a good idea to check and make
// sure our higher resolution isn't needed in large sims
// Anyways, make 2h = d(N(55))

#include <iostream>
#include "system.hpp"
#include "distance_vector.hpp"
#include "mac.hpp"

const int nParticlesh = 55;
const FP a_init = 1.2;           // 20% increase in space search each
time

// i is index to ps particle
void
System::smoothingLengthUpdate(const int& i)
{
  FP a = a_init;
  FP mDesired = nParticlesh * ps[i].Get_Mass();
  FP mFound = 0.0;
  DistanceVector dv;            // use this to get the distances and
sort them

  // make sure mDesired isn't more than is in the universe ...
  if( mDesired > (tree.Mass_Get() - ps[i].Get_Mass()))
    mDesired = (tree.Mass_Get() - ps[i].Get_Mass());

  while( mFound < mDesired)
    {
      mFound = 0.0;
      dv = tree.Smoothing_Lengths_Tree_Traverse(a, &ps[i], mFound);
      a *= a_init;
    }
  const Particle *h_particle = dv.QSortReduced(mDesired);
  ps[i].Set_Smoothing_Length(0.5 * d3d(&ps[i], h_particle));
}

// gravitational calculation
void
System::gSmoothingLengthUpdate(const int& i)
{
  // order of magnitude less than h
  ps[i].gSmoothingLengthSet(0.1 * ps[i].hGet());
}
```

```
void
System::Smoothing_Lengths_Update()
{
  DistanceVector dv;            // use this to get the distances and
sort them
  static int reit = 0;
  ++reit;

  // only do this if requested
  if (!use_var_h && reit > 1)
    return;

  int i;
  FOR_ALL_PARTICLES(i)
    {
      if(!ps[i].Is_Active())
        continue;
      smoothingLengthUpdate(i);
      gSmoothingLengthUpdate(i);
    }
}

// run through tree, picking up particles w/in 2*a*h_i of p_i
// calculate their distances to p_i and place that in `dist'
// and add up their masses to massTot
const DistanceVector
Tree::Smoothing_Lengths_Tree_Traverse(const FP& a, const Particle* p_i,
FP& massTot)
{
  DistanceVector dv;            // begins empty

  // is this cube w/in the 4*a*hi box?
  if(!SPHMac(4.0*a*p_i->hGet(), p_i))
    return dv;

  // no particles here
  if(subdivided)
    {
      for(int i=0; i<8; i++)
        {
          // note: actual particle d3 checking is done when particles
are inserted.
          dv.Add(subcubes[i]->Smoothing_Lengths_Tree_Traverse(a, p_i,
massTot));
        }
    }
  else
    {
      if(particle != NULL)
        if(particle != p_i)     // don't add self!
          {
            // particle->dist = d3dsquared(*p_i, *particle);
            particle->distSet(p_i);
            if( particle->distGet() < 2.0*a*p_i->hGet())
              {
                dv.Add(particle);
```

```
                massTot += particle->Get_Mass();
              }
          }
    }
  return dv;
}

const DistanceVector
Tree::gSmoothingLengthsTreeTraverse(const FP& a, const Particle* p_i,
FP& massTot)
{
  DistanceVector dv;            // begins empty

  // is this cube w/in the 4*a*hi box?
  if(!SPHMac(4.0*a*p_i->ghGet(), p_i))
    return dv;

  // no particles here
  if(subdivided)
    {
      for(int i=0; i<8; i++)
        {
          // note: actual particle d3 checking is done when particles
are inserted.
          dv.Add(subcubes[i]->Smoothing_Lengths_Tree_Traverse(a, p_i,
massTot));
        }
    }
  else
    {
      if(particle != NULL)
        if(particle != p_i)     // don't add self!
          {
            // particle->dist = d3dsquared(*p_i, *particle);
            particle->distSet(p_i);
            if( particle->distGet() < 2.0*a*p_i->ghGet())
              {
                dv.Add(particle);
                massTot += particle->Get_Mass();
              }
          }
    }
  return dv;
}
```

system.cpp

```
#include <iostream>
#include <climits>
#include "maths.hpp"
#include "system.hpp"

// access htree multiparticles, Id, Dimension, Boolean 0/1
#define MULTI_PARTICLE(I,D,B) (((I*7)+1+(D*2))+B)
// access htree bases for multiparticles
#define MAIN_PARTICLE(I) (I*7)
```

```
System::System()
{
  std::cout << "The elapsed time is set to 0\n";
  t_elapsed = 0;
}

// TODO: check whether inline is good
FP
d3d(const Particle *a, const Particle *b)
{
  FP x,x2,y,y2,z,z2;
  a->Get_Pos(x,y,z);
  b->Get_Pos(x2,y2,z2);

  return sqrt( pow(x-x2,2) +  pow(y-y2,2)  + pow(z-z2,2));
}

// faster w/o the root
FP
d3dsquared(const Particle &a, const Particle &b)
{
  FP x,x2,y,y2,z,z2;
  a.Get_Pos(x,y,z);
  b.Get_Pos(x2,y2,z2);

  return ( pow(x-x2,2) +  pow(y-y2,2)  + pow(z-z2,2));
}

// this is necessary for accelerated calculations
// uses simplest possible implementation
FP
symSmoothLens(FP h1, FP h2)
{
  return ( 0.5 * (h1 + h2) );
}


void
System::Debug()
{
  std::cout << tree.Mass_Get() << "\n";
  tree.CM_Traverse_Print();
}

void
rvect(FP array[], FP k, const Particle *a, const Particle *b)
{
  array[0] = k*(a->x_get() - b->x_get());
  array[1] = k*(a->y_get() - b->y_get());
  array[2] = k*(a->z_get() - b->z_get());
}

void project(FP array[], const FP v[], const FP u[])
{
  FP u_mag_squared = ( u[0]*u[0] + u[1]*u[1] + u[2]*u[2]);
  FP k = v[0]*u[0] + v[1]*u[1] + v[2]*u[2];
  k /= u_mag_squared;
```

```
  for(int i=0; i<3; i++)
          array[i] = u[i] * k;
}

void
System::treeCreate()
{
  int i;

  tree.reset();

  // std::cout << "Creating GTree\n";

  FOR_ALL_PARTICLES(i)
    {
      if(!ps[i].Is_Active())
        continue;
      tree.Boundaries_Extend(ps[i].x_get(), ps[i].y_get(),
ps[i].z_get());
    }

  // need to do this twice, the first is for the boundaries
  // std::cout << "Inserting particles...\n";
  FOR_ALL_PARTICLES(i)
    {
      if(!ps[i].Is_Active())
        continue;
      tree.Particle_Insert(&ps[i]);

    }
  // std::cout << "Finished inserting particles.\n";
}

void
System::htree_create()
{
  int i;

  htree.reset();

  // std::cout << "Creating HTree\n";

  // set htree properties &c.
  FOR_ALL_PARTICLES(i)
    {
      if(!ps[i].Is_Active())
        continue;
      FP pos[3] = {ps[i].x_get(), ps[i].y_get(), ps[i].z_get()};

      // ORIGINAL PARTICLE
      hps[MAIN_PARTICLE(i)].enable();
      hps[MAIN_PARTICLE(i)].Set_Pos(pos[0], pos[1], pos[2]);
      hps[MAIN_PARTICLE(i)].Set_Smoothing_Length(ps[i].hGet());
      hps[MAIN_PARTICLE(i)].Set_Mass(ps[i].Get_Mass());
      htree.boundaries_extend(pos);
      for(int d=0; d<3; d++)
        {
```

```
          FP newpos[3] = {pos[0], pos[1], pos[2]};
          // PLUS ONE
          // std::cout << ps[i].hGet() << "\n";
          newpos[d] = pos[d] + ps[i].hGet();
          hps[MULTI_PARTICLE(i,d,0)].enable();
          hps[MULTI_PARTICLE(i,d,0)].Set_Pos(newpos[0], newpos[1],
newpos[2]);

hps[MULTI_PARTICLE(i,d,0)].Set_Smoothing_Length(ps[i].hGet());
          hps[MULTI_PARTICLE(i,d,0)].Set_Mass(0);
          htree.boundaries_extend(newpos);
          // MINUS ONE
          newpos[d] = pos[d] - ps[i].hGet();
          hps[MULTI_PARTICLE(i,d,1)].enable();
          hps[MULTI_PARTICLE(i,d,1)].Set_Pos(newpos[0], newpos[1],
newpos[2]);

hps[MULTI_PARTICLE(i,d,1)].Set_Smoothing_Length(ps[i].hGet());
          hps[MULTI_PARTICLE(i,d,1)].Set_Mass(0);
          htree.boundaries_extend(newpos);
        }
    }

  // need to do this twice, the first is for the boundaries
  // std::cout << "Inserting particles...\n";
  FOR_ALL_PARTICLES(i)
    {
      if(!ps[i].Is_Active())
        continue;
      // ORIGINAL PARTICLE
      htree.particle_insert(&hps[MAIN_PARTICLE(i)]);
      for(int d=0; d<3; d++)
        {
          // PLUS ONE
          htree.particle_insert(&hps[MULTI_PARTICLE(i,d,0)]);
          // MINUS ONE
          htree.particle_insert(&hps[MULTI_PARTICLE(i,d,1)]);
        }
    }

  // now calculate the smoothing lengths
  htree.smoothing_lengths_update();
  htree.prune_tree();
  // std::cout << "Finished with HTree.\n";
}

void
System::internal_energies_calc()
{
  int i;
  htree.density_energy_update();
  FOR_ALL_HPARTICLES(i)
    {
      if(!hps[i].Is_Active())
        continue;
      htree.density_traverse(&hps[i]);
      opacity_update(&hps[i]);
```

```
    }
}

// interp. b/t assrted data pts. for gases by temperature.
void
System::opacity_update(Particle *p)
{
  if(p->temp_low()) {
    p->opacity_calc_low_temp();
    return;
  }
  // temp must be high
  p->opacity_calc_high_temp();
}
```

system.hpp

```
#ifndef _SYSTEM_H_
#define _SYSTEM_H_

#include "maths.hpp"
#include "particle.hpp"
#include "tree.hpp"
#include "htree.hpp"

// hardcode n particles
#define N_PARTICLES 1000
#define M_UNIFIED 1e7

// how close two particles are before they collide
#define PARTICLE_PROX 3.0

//bounds
#define ENABLE_BOUNDS
#define BOUND 100

// spin through all the particles.
// of course, this shouldn't be happening often...
#define FOR_ALL_PARTICLES(I) for(I=0;I<N_PARTICLES;I++)
#define FOR_ALL_HPARTICLES(I) for(I=0;I<(7*N_PARTICLES);I++)

class System
{
public:
  System();
  void Data_Read();
  void Properties_Assign_Init();
  bool Time_Update();
  void treeCreate();
  void htree_create();
  void CenterMass_Update();
  void Smoothing_Lengths_Update();
  void gSmoothingLengthUpdate(const int& i);
  void smoothingLengthUpdate(const int& i);
  void Densities_Update();
  void Pressures_Update();
  void accelsCalc();
```

```
  void internal_energies_calc();
  void Integrate();
  void Data_Write();
  void dataWriteBegin();
  void dataWriteEnd();
  void Debug();
  void opacity_update(Particle *p);
private:
  FP Kernel(FP r, FP h);
  Particle ps[N_PARTICLES];      // particle system
  Particle hps[N_PARTICLES * 7]; // hydrodynamical particle system
  FP forceSymmetrizeSmoothingLengths(FP h1, FP h2);
  // FP d3d(FP a, FP b, FP c, FP a2, FP b2, FP c2); // 3d dist.
  Tree tree;                  // gravity tree
  HTree htree;              // hydrodynamics tree
  FP t_elapsed;               // time
  FP t_step;                  // time step
  FP t_final;                 // final time
  bool use_var_h;       // use variable smoothing lengths?
  bool use_accel_checking;      // check NlogN algorithm vs. N*N
algorithm?
};


FP d3d(const Particle *a, const Particle *b);
FP d3dsquared(const Particle &a, const Particle &b);
FP symSmoothLens(FP h1, FP h2);
void rvect(FP array[], FP k, const Particle *a, const Particle *b);
void project(FP array[], const FP v[], const FP u[]);
#endif
```

timestep.cpp

```
// timestep.cc
// info about the timestep and integration
#include <iostream>
#include "system.hpp"

// this function updates the current time and checks if there
// is any left.
bool System::Time_Update()
{
  t_elapsed += t_step;
  std::cout << "Time elapsed: " << t_elapsed
          << " of " << t_final <<  ".\n";

  if((t_final-t_elapsed) <= 0)
    return false;
  else
    return true;
}
```

tree.cpp

```
// The N-Ary tree that assists with force calculation
// See `tree.txt' for more information
```

```
#include <iostream>
#include <cassert>
#include "tree.hpp"

#define SUBCUBE_NDX(X,Y,Z) ((X) + 2*(Y) + 4*(Z))

Tree::Tree()
{
    subdivided = 0;

    x_min = 0;
    y_min = 0;
    z_min = 0;
    x_max = 0;
    y_max = 0;
    z_max = 0;
    x_mid = 0;
    y_mid = 0;
    z_mid = 0;

    particle = 0;
    subcubes[0] = 0;

    m = 0;
    cm.Set_Pos(0,0,0);
  }

Tree::~Tree()
{
  // death to trees and children
  int i;

  if(subdivided)
    for(i=0; i<8; i++)
      delete subcubes[i];
}

void
Tree::reset()
{
    subdivided = 0;

    x_min = 0;
    y_min = 0;
    z_min = 0;
    x_max = 0;
    y_max = 0;
    z_max = 0;
    x_mid = 0;
    y_mid = 0;
    z_mid = 0;

    particle = 0;
    subcubes[0] = 0;

    m = 0;
    cm.Set_Pos(0,0,0);
```

```
    destroyBranches();
}

void
Tree::destroyBranches()
{
  if(subdivided)
    {
      int i;
      for(i=0; i<8; i++)
        {
          subcubes[i]->destroyBranches();
          delete subcubes[i];
        }
    }
}

Tree *
Tree::Tree_Choose_Subdivision(const Particle *p) const
{
  assert(subcubes[0] != NULL);

  int i;
  bool cmps[3] = {0,0,0};

  if (p->x_get() > x_mid)
    cmps[0] = 1;
  if (p->y_get() > y_mid)
    cmps[1] = 1;
  if (p->z_get() > z_mid)
    cmps[2] = 1;

  i = SUBCUBE_NDX(cmps[0], cmps[1], cmps[2]);
  // std::cout << "Chose subdivision: " << i << std::endl;
  return subcubes[i];
}

void
Tree::Subdivide()
{
  // make sure we aren't already subdivided and there is no particle in
square that will be lost
  assert(subdivided == 0);
  assert(particle == NULL);

  int i,j,k,ndx;

  // throw the flag
  subdivided = 1;

  // set the midpoints
  x_mid = FP_AVR(x_min, x_max);
  y_mid = FP_AVR(y_min, y_max);
  z_mid = FP_AVR(z_min, z_max);

  // Boundaries_Print();
```

```
  // std::cout << x_min << " "
  //           << x_mid << " "
  //           << x_max << std::endl;

  // literally i=x, j=y, k=z
  for(i=0; i<2; i++)
    for(j=0; j<2; j++)
      for(k=0; k<2; k++)
        {
          ndx = SUBCUBE_NDX(i, j, k);
          subcubes[ndx] = new Tree;
          if(!i)
            {
              // std::cout << "x min on " << ndx << std::endl;
              subcubes[ndx]->x_min = x_min;
              subcubes[ndx]->x_max = x_mid;
            }
          else
            {
              subcubes[ndx]->x_min = x_mid;
              subcubes[ndx]->x_max = x_max;
            }
          if(!j)
            {
              // std::cout << "y min on " << ndx << std::endl;
              subcubes[ndx]->y_min = y_min;
              subcubes[ndx]->y_max = y_mid;
            }
          else
            {
              subcubes[ndx]->y_min = y_mid;
              subcubes[ndx]->y_max = y_max;
            }
          if(!k)
            {
              // std::cout << "z min on " << ndx << std::endl;
              subcubes[ndx]->z_min = z_min;
              subcubes[ndx]->z_max = z_mid;
            }
          else
            {
              subcubes[ndx]->z_min = z_mid;
              subcubes[ndx]->z_max = z_max;
            }

        }
}

void
Tree::Particle_Insert(Particle *p)
{
  Tree *subcube = NULL;
  Particle *lost_child = NULL;

  if(subdivided)
    {
      subcube = Tree_Choose_Subdivision(p);
```

```
      subcube->Particle_Insert(p);
    }
  else
    {
      if(particle == NULL)      // nobody else here
        {
          // std::cout << "success\n";
          Particle_Set(p);      // this
          return;
        }
      // other particle is here
      assert(particle != NULL);

      // if we don't check to see if they're in the same place then
it's going to keep subdividing
      // forever.  if this is the case, set one to NULL and the other
to the combined mass of the
      // two
      if( *particle == *p)
        {
          DEBUG_MSG("Particles collided\n");
          particle->Add_Mass(p->Get_Mass());
          //particle->check_most_massive();
          p->Disable();
          return;               // our work is done
        }
      lost_child = particle;         // find this a home
      particle = NULL;               // no more particle in subdiv cube

      Subdivide();              // throw flag
      subcube = Tree_Choose_Subdivision(lost_child);
      // debug
      // std::cout << subcube->Print_Subdivided() << std::endl;
      subcube->Particle_Insert(lost_child);
      subcube = Tree_Choose_Subdivision(p);
      subcube->Particle_Insert(p);
    }
}

void
Tree::Boundaries_Extend(FP x, FP y, FP z)
{
  FP xsold = x_max - x_min;
  FP ysold = y_max - y_min;
  FP zsold = z_max - z_min;
  FP add = 0;

  if (x < x_min)
    {
      if((x_min - x) > add)
        add = (x_min - x);
      x_min = x;
    }
  if (y < y_min)
    {
      if((y_min - y) > add)
        add = (y_min - y);
```

```
      y_min = y;
    }
  if (z < z_min)
    {
      if((z_min - z) > add)
        add = (z_min - z);
      z_min = z;
    }
  if (x > x_max)
    {
      if((x - x_max) > add)
        add = (x - x_max);
      x_max = x;
    }
  if (y > y_max)
    {
      if((y - y_max) > add)
        add = (y - y_max);
      y_max = y;
    }
  if (z > z_max)
    {
      if((z - z_max) > add)
        add = (z - z_max);
      z_max = z;
    }
  // add now holds the greatest change in size
  // compare current to old sizes and make sure the change adds up to
`add'
  FP todox = 0.5 * (add - ((x_max - x_min) - xsold));
  FP todoy = 0.5 * (add - ((y_max - y_min) - ysold));
  FP todoz = 0.5 * (add - ((z_max - z_min) - zsold));

  x_min -= todox;
  x_max += todox;
  y_min -= todoy;
  y_max += todoy;
  z_min -= todoz;
  z_max += todoz;
}

void
Tree::Boundaries_Print() const
{
  std::cout << "x_min " << x_min << " x_mid " << x_mid << " x_max "<<
x_max << ".\n";
  std::cout << "y_min " << y_min << " y_mid " << y_mid << " y_max "<<
y_max << ".\n";
  std::cout << "z_min " << z_min << " z_mid " << z_mid << " z_max "<<
z_max << ".\n";
}

// set a particle for this tree's cube
void
Tree::Particle_Set(Particle *p)
{
  assert(particle == NULL);
```

```
  particle = p;
}
```

tree.hpp

```
// Modified Barnes-Hut tree (Stephen Oxley)

#ifndef _TREE_HPP_
#define _TREE_HPP_

#include <iostream>
#include <vector>
#include "particle.hpp"
#include "maths.hpp"
#include "distance_vector.hpp"

class Tree
{
public:
  Tree();
  ~Tree();
  // reset
  void reset();

  // boundaries
  void Boundaries_Extend(FP x, FP y, FP z);
  void Boundaries_Print() const;
  // subdivisions
  void Subdivide();
  bool Subdivided_Print() {return subdivided;}
  // particles
  void Particle_Insert(Particle *p);
  void Particle_Set(Particle *p);
  // center of mass
  void Mass_Traverse();
  FP Mass_Get() const {return m;}
  const Particle* CM_Get() const {return &cm;}
  void CM_Traverse_Print() const;
  // smoothing lengths
  const DistanceVector Smoothing_Lengths_Tree_Traverse(const FP& a,
const Particle* p_i, FP& massTot);
  const DistanceVector gSmoothingLengthsTreeTraverse(const FP& a, const
Particle* p_i, FP& massTot);
  // density
  void densityTraverse(Particle *p);
  // macs
  bool SPHMac(const FP& k, const Particle *p_i);
  bool BHMac(const Particle *p);
  // accels
  void accelsTraverse(Particle *p);
private:
  void destroyBranches();
  Tree* Tree_Choose_Subdivision(const Particle *p) const;
  bool subdivided;
  FP x_min, y_min, z_min;
  FP x_max, y_max, z_max;
  FP x_mid, y_mid, z_mid;
```

```
  Tree *subcubes[8];
  Particle *particle;
  FP m;                         // the collected mass of the tree
  Particle cm;
};

#endif
```