NUMERICAL SIMULATION OF BINARY FORMATION IN OPEN STAR CLUSTERS

by
Mike Williams

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Masters of Science (Applied Physics).

Golden, Colorado

Date _____

Signed: _____
Mike Williams

Approved: _____
Dr. F. E. Cecil
Professor of Physics
Thesis Advisor

Approved: _____
Dr. James McNeil
Professor and Head
Thesis Advisor

Golden, Colorado

Date _____

_____
Dr. James McNeil
Professor and Head,
Department of Physics

ABSTRACT

We have generated a three-dimensional (3D) dynamic numerical simulation of binary star formation in open star clusters. Several integration methods were tested, and the adaptive fourth order Runge-Kutta method was selected. A softening method was used to prevent excessive error from close interactions between stars. Other methods of handling close interactions, such as a quasi analytical approach, were investigated and then rejected. Care was taken in the selection of initial conditions so as to reasonably approximate the conditions of an open star cluster. Ultimately, an insufficient number of binary stars were formed through dynamic interactions to account for the large number of binaries found in nature, suggesting that these binaries are primordial.

TABLE OF CONTENTS

ABSTRACT iii

LIST OF FIGURES vi

LIST OF TABLES vii

ACKNOWLEDGMENT viii

Chapter 1 INTRODUCTION 1

1.1 Importance of Binary Systems 1
1.2 Existing Work 2
1.3 Goals of This Work 3
1.4 Definition of a Simulation 3

Chapter 2 SELECTION OF NUMERICAL TECHNIQUE 5

2.1 Summary of Methods 5
2.1.1 Taylor Series 5
2.1.2 Hermite 6
2.1.3 Fourth Order Runge Kutta 7
2.1.4 Verlet 8
2.1.5 Fourth Order Verlet 9
2.1.6 ODE45 and ODE15 10
2.2 Numerical Comparison of Methods 11
2.3 Stiffness 14
2.4 Details of the Numerical Solution 16
2.4.1 Scaling of Units 16
2.4.2 Adaptive Time Stepping 18
2.4.3 Softening 20
2.4.4 Binary Counting 22

Chapter 3 QUASI ANALYTICAL SOLUTION 23

3.1 Kepler's Problem 23
3.2 Kepler's Problem Solved in Time 24

3.3 Translation From Center of Mass Frame to Lab Frame 25
3.4 Limitations of This Approximation 26

Chapter 4 CALCULATION OF BINARY FORMATION RATE 29

4.1 Initial conditions of a Star Cluster 29
4.1.1 Mass Distribution 29
4.1.2 Spatial Distribution 31
4.1.3 Energy Distribution 32
4.2 Details of the Simulation Runs 33
4.3 Numerical Results 33
4.4 Comparison to Observational Data 36
4.5 Conclusions of the Study 38

Chapter 5 WORK FOR THE FUTURE 39

REFERENCES 41

APPENDIX A DERIVATION OF ODE SOLVING TECHNIQUES 43

A.1 Third Order Taylor Expansion 43
A.2 Hermite 45
A.3 RK4 50
A.4 Verlet 53
A.4.1 Velocity Approximation 53
A.5 Verlet4 54

APPENDIX B NEWTON'S METHOD 59

APPENDIX C SOURCE CODE 61

CD . . . . . Back Pocket

LIST OF FIGURES

2.1 Percent Error in R 12

2.2 Percent Error in R Zoom 13

2.3 Percent Error in R×Time 14

2.4 Order of Convergence 15

2.5 Small Systems 19

2.6 Softening Error 21

3.1 R Diagram 27

4.1 Midpoint Binning 30

LIST OF TABLES

2.1 Comparison Of Method Properties 11

4.1 Numerical Results for 200 Star Runs 34

4.2 Numerical Results for 100 Star Runs 35

4.3 Column Heading for Tables 4.1 and 4.2 36

4.4 Percent of Stars in Binary Systems 37

ACKNOWLEDGMENT

Without the guidance of my advisers Dr. Cecil and Dr. McNeil, the support of my girlfriend Stephanie Wyandt, and the screaming fast computer time provided by the Chemistry Department and Dr. Whilloughby, this work would not exist.

Chapter 1

INTRODUCTION

1.1 Importance of Binary Systems

A great deal of what is known about astronomy is knowledge gained from the study of binary stars. The physics of a binary star make it possible to identify the masses of stars in the pair. This in turn allows for determining mass luminosity relationships and knowledge of masses of non-binary stars. The mass of stars is important to all of astronomy. From theories of dark matter to the detection of black holes, everything requires knowledge of the mass.

Binaries are the smallest of all astronomical structures, and as such, understanding them is key to the understanding of larger structures such as star clusters, galaxies, and galaxy clusters (Larson, 1997). Most stars are part of a binary or larger system, with the trend being that younger stars are more likely to have a partner than older stars (Abt, 1999). Abt (1999) further suggests that there is a relationship between age and what type of partner a star has. His observations of star clusters show that in young clusters high mass stars tend to have low mass partners, but in older clusters there is a slight tendency for high mass stars to have high mass partners. Presumably this is due to some sort of ionization and reforming process.

1.2 Existing Work

There are two ways that a binary can form. Either a cloud of gas could condense into two stars (a primordial binary) or two unassociated stars could become bound in a three body interaction. However, since the time scale of astronomical events is much longer than any observational time it is impractical for us to observe either directly. So, it is necessary to learn about them via simulations. Both formation methods have been studied numerically, and in some cases both have been studied in the same model (Zwart, 1997).

Most of the work in star cluster simulation seems to be centered around studying the aggregate behavior of large clusters with around 3×10^4 stars (Zwart, 1998). Such approaches typically use one of two major n-body codes, Kira or one of Aarseth's N-Body# series (there are at least 6, N-Body1 - N-Body6). These codes are designed for large numbers of stars and as such take special care to avoid close interactions and the stiffness they cause [1]. As a result they do not treat binary formation accurately (Aarseth, 2003). This approximation is justifiable because on the scale of the cluster binary interactions are not considered important. Another feature of the Kira and N-Body# codes is that they use the Hermite integration method[2]. Large number simulations also typically use special purpose computers known as GRAPE, which calculate the gravitational force in hardware (Zwart, 1998).

We have found two papers studying the behavior of binary stars in open star clusters. Zwart (2000) studies binary formation in small star clusters. However Zwart uses Kira to do so, and as noted above, Kira does not take close interactions into proper account. Kroupa (2001) studies the evolution of binary periods, but not

[1] See section 2.3 for details on stiffness.
[2] See section 2.1 for more details on integration methods.

their formation.

1.3 Goals of This Work

There seems to be a lack of direct numerical simulations of binary formation. The goal of this thesis is to study a small system of around 100 stars and examine the binary formation within that system. Also, most studies are done on special purpose computers with large black box type codes. This study will attempt to produce a code which is simple and transparent in nature and that will run effectively on a desktop computer. Finally, it has been very difficult to learn about numerically solving the n-body problem from the existing literature. This paper is intended to be readily understood by anyone with a college background in math and science.

1.4 Definition of a Simulation

In order to simulate the movement of stars it is necessary to solve their equations of motion. Under the classical assumption of point particles[3], a good starting point is Newton's second law, $\vec{F} = m\vec{a}$. Expressing this in terms of the spatial coordinate $\vec{r}_i$ and solving for the acceleration, yields:

$$\vec{a}_i = \frac{d^2\vec{r}_i}{dt^2} = \frac{1}{m_i}\sum_{j\neq i}\overrightarrow{F}_{ij}, \tag{1.1}$$

where

[3]Actual stars have finite radii so this approximation will break down as the distance between stars approaches that radius. Also the classical assumption is that Special and General relativistic affects are negligible.

$$\overrightarrow{F}_{ij} = -\frac{Gm_i m_j}{\left|\vec{R}_{ij}\right|^2}\hat{R_{ij}}, \tag{1.2}$$

$\vec{R_{ij}} = \vec{r_i} - \vec{r_j}$ and G is the universal gravitational constant.

Unfortunately, it is not possible to solve this general problem analytically for more than two bodies[4]. In the absence of an analytical solution there still remains a numerical solution. If that numerical solution obeys appropriate physical constraints, it can be called a simulation.

[4]Although analytical solutions do exist for very specific problems such as four equal mass particles arranged on the corners of a square.

Chapter 2

SELECTION OF NUMERICAL TECHNIQUE

Eight methods of solving the gravitational Ordinary Differential Equation (ODE) were tested for this study. These methods are a second order Taylor series, a third order Taylor series, Verlet, Fourth order Verlet, Fourth order Runge Kutta, Hermite, ODE45 and ODE15. ODE45 and ODE15 are black box Matlab functions.

Several test problems with known solutions were used to test these methods. The simplest of these is a one kilogram (kg) object in a circular orbit around the sun with a radius of one astronomical unit (Au). Such an object should have a period of one year and should remain one Au from the sun. The difference between the simulated radius and the known radius at any given time is a good indicator of error. Also, the system should have constant energy. The difference between the calculated energy and the starting energy of the system is another indication of the error.

2.1 Summary of Methods

The following subsections provide the basic equations and order of convergence of each method tested. For a detailed derivation of methods see Appendix A. Table 2.1 summarizes all of the method properties.

2.1.1 Taylor Series

The first methods attempted are based on a simple Taylor series. In this method the position and velocity are solved for separately, turning the N equations from

equation 1.1 into $2N$ equations of the form

$$\overrightarrow{r}\,(t+\Delta t)=\vec{r}(t)+\vec{r}^{\,\prime}\,(t)\,\Delta t+\frac{1}{2}\vec{r}^{\,\prime\prime}\,(t)\,\Delta t^2+\frac{1}{6}\vec{r}^{\,\prime\prime\prime}\,(t)\,\Delta t^3+O\left(\Delta t^4\right), \tag{2.1}$$

$$\overrightarrow{r}^{\,\prime}\,(t+\Delta t)=\vec{r}^{\,\prime}(t)+\vec{r}^{\,\prime\prime}\,(t)\,\Delta t+\frac{1}{2}\vec{r}^{\,\prime\prime\prime}\,(t)\,\Delta t^2+O\left(\Delta t^3\right). \tag{2.2}$$

Where $\vec{r}^{\,\prime}(t)=\frac{d\vec{r}}{dt}$, $\vec{r}^{\,\prime\prime}(t)=\frac{d^2\vec{r}}{dt^2}$, etc.

Here the series is truncated after $\vec{r}^{\,\prime\prime\prime}$because that is the last term known analytically. $\vec{r}^{\,\prime\prime\prime}$ is shown by Aarseth (2003) to be:

$$\vec{r}\,(t)^{\prime\prime\prime}=-\frac{m\vec{r}\,(t)^{\prime}}{|\vec{r}|^3}-3a\,(t)\,r\,\vec{(}t)^{\prime\prime} \tag{2.3}$$

$$a=\vec{r}\,(t)\cdot\frac{\vec{r}^{\,\prime}\,(t)}{|\vec{r}|^2}. \tag{2.4}$$

See Appendix A for a derivation.

If instead the series is truncated at $\vec{r}^{\,\prime\prime}$, you get the somewhat more intuitive second order Taylor series approximation. This method only achieves first order convergence at the expense of two evaluations of the force/force derivative per time step.

2.1.2 Hermite

Hermite integration is the same as the third order Taylor series except that it is a two step method. The Hermite equation is given by Aarseth (2003) as:

$$\overrightarrow{r}\left(t+\Delta t\right)=\vec{r}_{Taylor3}(t)+\overrightarrow{\Delta r\left(t\right)}. \tag{2.5}$$

$$\overrightarrow{r}^{'}\left(t+\Delta t\right)=\vec{r}_{Taylor3}^{'}(t)+\overrightarrow{\Delta r^{'}}\left(t\right). \tag{2.6}$$

$$\overrightarrow{\Delta r\left(t\right)}=\Delta t^{2}\left[-\frac{-9F_{T}+9F\left(t\right)+\left(7F^{'}\left(t\right)+2F_{T}^{'}\right)\Delta t}{60m}\right]+O\left(\Delta t^{6}\right). \tag{2.7}$$

$$\overrightarrow{\Delta r^{'}\left(t\right)}=\Delta t\left[-\frac{-6F_{T}+6F\left(t\right)+\left\{5F^{'}\left(t\right)+F_{T}^{'}\right\}\Delta t}{12m}\right]+O\left(\Delta t^{5}\right). \tag{2.8}$$

Where $\vec{r}_{Taylor3}$ is defined by equation 2.1, $\vec{r}_{Taylor3}^{'}$ is defined by equation 2.2, and F_T and $F_T^{'}$ are temporary force and force derivatives evaluated at $\vec{r}_{Taylor3}$. This method achieves fourth order convergence at the expense of four evaluations of the force/force derivative per time step.

2.1.3 Fourth Order Runge Kutta

Fourth order Runge Kutta (RK4) is the best of the algorithms that solve for $\vec{r}$ and $\vec{r^{'}}$ separately. It is a four step method and achieves fourth order accuracy. The fourth order Runge Kutta equations are:

$$\vec{r_{1}}=\vec{r^{'}}\left(t\right)\Delta t,$$

$$\vec{r_{1}^{'}}=\frac{1}{m}\vec{F}\left(\vec{r}(t)\right)\Delta t,$$

$$\vec{r_2} = \left(\vec{r'}\,(t) + \frac{1}{2}\vec{r_1'}\right)\Delta t,$$

$$\vec{r_2'} = \frac{1}{m}\vec{F}\left(\vec{r}(t) + \frac{1}{2}\vec{r_1}\right)\Delta t,$$

$$\vec{r_3} = \left(\vec{r'}\,(t) + \frac{1}{2}\vec{r_2'}\right)\Delta t,$$

$$\vec{r_3'} = \frac{1}{m}\vec{F}\left(\vec{r}(t) + \frac{1}{2}\vec{r_2}\right)\Delta t,$$

$$\vec{r_4} = \left(\vec{r'}\,(t) + \vec{r_3'}\right)\Delta t,$$

$$\vec{r_4'} = \frac{1}{m}\vec{F}\left(\vec{r}\,(t) + r_3\right)\Delta t,$$

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \frac{1}{6}\left(\overrightarrow{r_1} + 2\left(\overrightarrow{r_2} + \overrightarrow{r_3}\right) + \overrightarrow{r_4}\right), \tag{2.9}$$

$$\vec{r'}\,(t + \Delta t) = \vec{r'}\,(t) + \frac{1}{6}\left(r_1' + 2\left(\vec{r_2'} + \vec{r_3'}\right) + \vec{r_4'}\right). \tag{2.10}$$

RK4 achieves fourth order convergence at the expense of four evaluations of the force per time step.

2.1.4 Verlet

By contrast with the methods listed above, Verlet is unique in that it does not split the N equations into 2N equations. Instead it solves directly for $\vec{r}$ and requires

you to approximate $\vec{r}'$ separately. The Verlet equation as shown by Erolessi (1997) is:

$$\overrightarrow{r}\left(t+\Delta t\right)=2\overrightarrow{r}\left(t\right)-\overrightarrow{r}\left(t-\Delta t\right)+\overrightarrow{r''}\left(t\right)\Delta t^{2}+O\left(\Delta t^{4}\right). \tag{2.11}$$

The velocity can be approximated by the equation:

$$\begin{aligned}\vec{r'}\left(t+\Delta t\right) &= \frac{25\overrightarrow{r}\left(t\right)-48\vec{r}\left(t-\Delta t\right)+36\vec{r}\left(t-2\Delta t\right)}{12\Delta t}\\ &\quad -\frac{16\vec{r}\left(t-3\Delta t\right)+3\vec{r}\left(t-4\Delta t\right)}{12\Delta t},\end{aligned} \tag{2.12}$$

which is accurate to $O\left(\Delta t^{4}\right)$ at the expense of keeping five past positions.

As you can see, this convenience comes at the expense of time step dependence. If you wish to change the time step, it is necessary to perform a single step with another method to reset $r(t-\Delta t)$ to match with the new Δt. Worse than that, other methods require knowledge of $\vec{r}'$ which is not known explicitly and must be approximated, introducing unnecessary error into the solution. The advantage is that you get second order convergence in one step, and the consideration of the previous position results in exceptional stability.

Verlet achieves second order convergence at the expense of one evaluation of the force per time step.

2.1.5 Fourth Order Verlet

Fourth order Verlet (Verlet4) is a four step method based on the original Verlet algorithm. It has all the advantages and disadvantages of Verlet, but achieves fourth order accuracy. Fourth order Verlet is based on the equations:

$$\vec{r_V} = 2\vec{r}\,(t) - \vec{r}\,(t-\Delta t) + \frac{1}{m}\vec{F}\,(\vec{r}\,(t))\,\Delta t^2,$$

$$\vec{r_{t1}} = \vec{r}_V + \frac{1}{12}\left\{\frac{1}{m}\vec{F}\,(\vec{r}\,(t-\Delta t)) - \frac{1}{m}\vec{F}\,(\vec{r}\,(t))\right\}\Delta t^2,$$

$$\vec{r_{t2}} = \vec{r}_V + \frac{1}{12}\left\{\frac{1}{m}\vec{F}\,(\vec{r}_{t1}) + \frac{1}{m}\vec{F}\,(\vec{r}\,(t-\Delta t)) - 2\frac{1}{m}\vec{F}\,(\vec{r}\,(t))\,,\right\}\Delta t^2$$

$$\begin{aligned}\vec{r}\,(t+\Delta t) &= \vec{r}_V + \frac{1}{12}\left\{\frac{1}{m}\vec{F}\,(\vec{r}_{t2}) + \frac{1}{m}\vec{F}\,(\vec{r}\,(t-\Delta t)) - 2\frac{1}{m}\vec{F}\,(\vec{r}\,(t))\right\}\Delta t^2 \\ &\quad + O\left(\Delta t^6\right). \end{aligned} \tag{2.13}$$

As with Verlet, when you want to know the velocity, use equation 2.12.

Verlet4 achieves fourth order convergence at the expense of three evaluations of the force per time step.

2.1.6 ODE45 and ODE15

ODE45 and ODE15 are Matlab functions for solving differential equations. ODE15 is specifically designed for stiff problems[1]. According to Matlab documentation, they achieve first through fifth order convergence according to the numbers in their titles. These functions performed well on simple test problems. However they failed on more complex problems. For instance, when long simulation time is used, which would require more time steps, they were unable to meet their own error tolerance

[1]See section 2.3 for further information on stiffness.

Method	N^2 Function Evaluations	Order of Convergence
Taylor	2	1
Hermite	4	4
RK4	4	4
Verlet	1	2
Verlet4	3	4

Table 2.1. Comparison Of Method Properties

limits, issuing obscure error messages and failing to do the calculation accurately. They completely failed to solve the three body problem, regardless of the number of time steps required. As such, Matlab packages were rejected, and we chose to develop and test the preceding methods which handle the complexities of this problem and where the convergence and stability properties are transparent.

2.2 Numerical Comparison of Methods

The above methods were tested based on a two body system consisting of a one kg object in an earth-like orbit about a solar mass object. Such a system should have a constant separation between the bodies, a constant energy, and a period of one year. The simulation was run for two periods. Deviations from the energy and the constant radius give a good estimation to the error.

The stability of Verlet and Verlet4 can easily be seen from figure 2.1 which is a plot of the percent error in r

$$100 \times \text{Max}\left(\text{Abs}\left(\frac{R - R_0}{R_0}\right)\right)$$

versus the time step. The time step must be larger than 20% of the period before

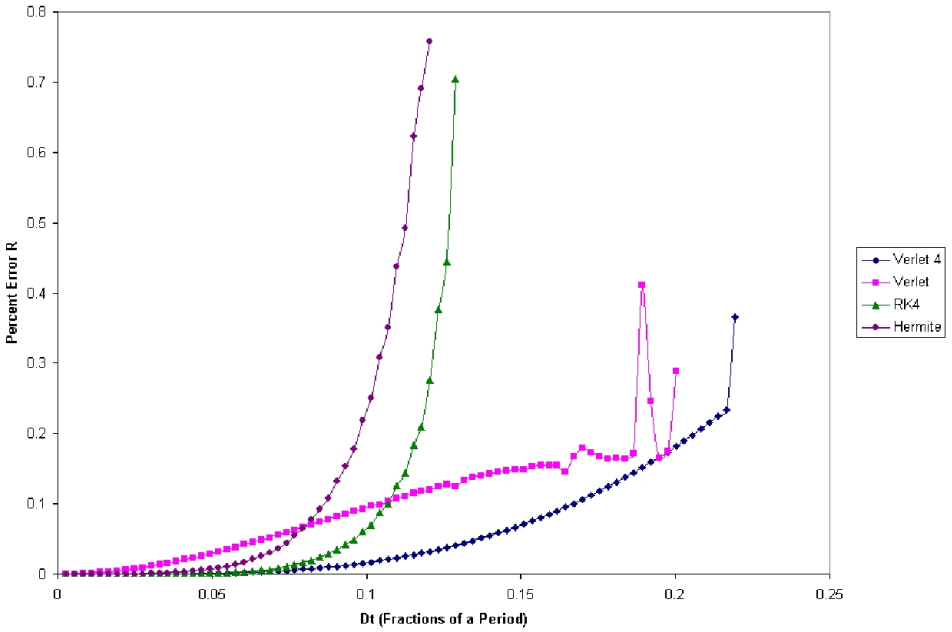

Figure 2.1. Percent Error in R

critical failure occurs. Figure 2.2 is the same data as figure 2.1 except it focusses on the small time step section of the graph to illustrate where Verlet4 surpasses RK4. The advantages of each method is most easily understood in figure 2.3, which represents the product of the percent error in R and the time to execute.

Figure 2.4 shows the order of convergence of the methods. The order of convergence is a measure of how much smaller the error gets when the time step is reduced. To calculate the order of convergence, we run the simulation twice with different time steps tracking the maximum error each time. Then apply:

$$\frac{\log\left(\frac{E_2}{E_1}\right)}{\log\left(\frac{\Delta t_2}{\Delta t_1}\right)}. \tag{2.14}$$

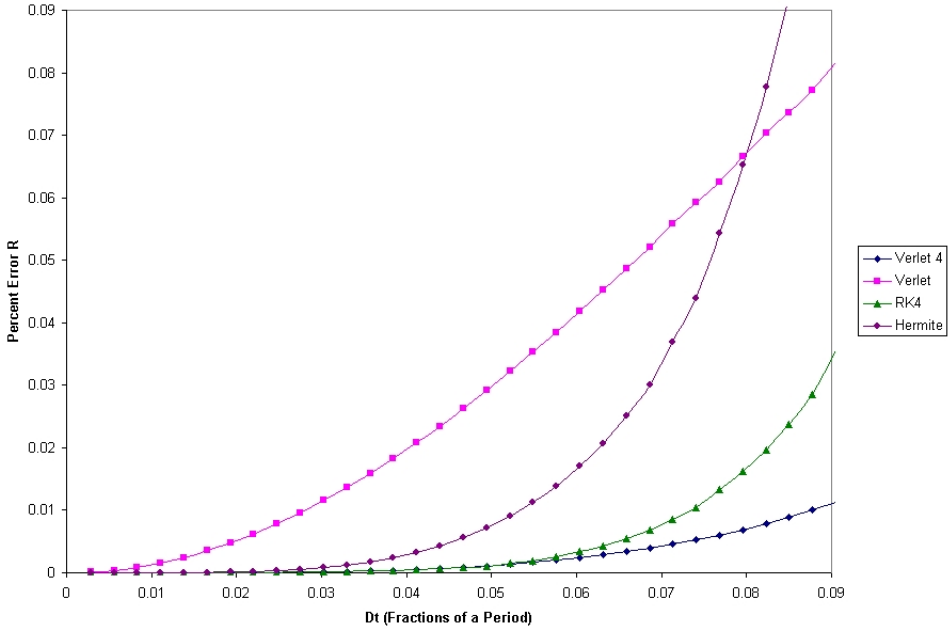

Figure 2.2. Percent Error in R Zoom

Figure 2.4 is a bit misleading. At a glance, it seems that RK4 achieves 26th order convergence and is the best method imaginable. This is an illusion. Equation 2.14 is only valid in the region where the method is working. If the time step is too large, the numerical solution no longer resembles the analytical solution. For instance, the orbit might change from circular to hyperbolic. In such a situation the method is not guaranteed to fail in a predictable way.

Looking at these figures, it seems obvious that Verlet4 is the best choice of integrating methods. It is necessary to make the time step larger than 20% of the period before Verlet4 fails. However, Verlet type algorithms have the significant draw back that they are dependent on knowledge of past positions. This makes it difficult to change time steps in order to handle close interactions and binary formation. As such,

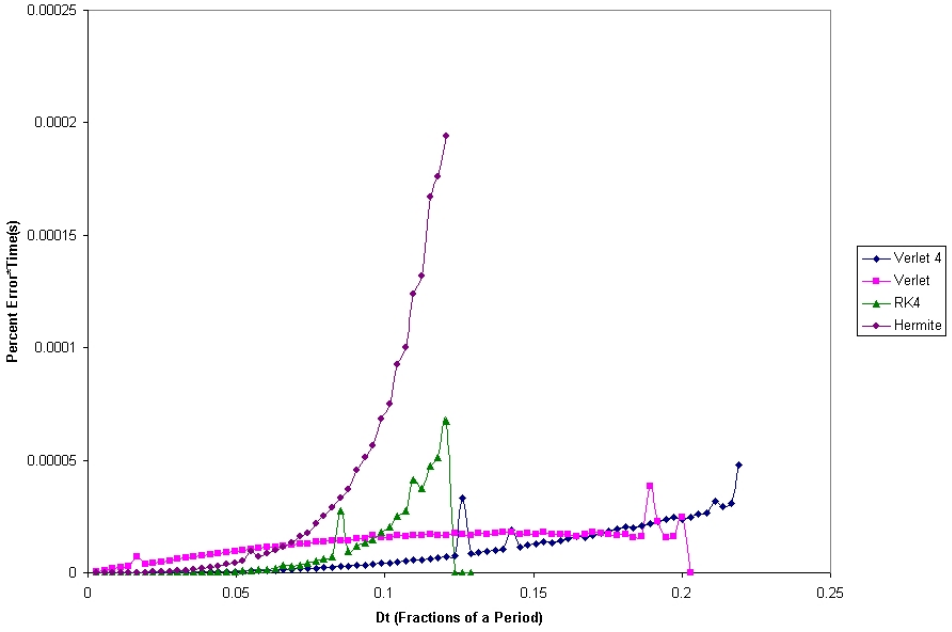

Figure 2.3. Percent Error in R×Time

any large scale dynamic simulation must be carried out with RK4, which performs nearly as well as Verlet4.

2.3 Stiffness

All of the methods presented operate under the same basic assumption. They all advance from $t_0 \rightarrow t_1$ in a series of time steps. During those time steps they assume that the forces, velocities, jerks, etc. are all constant. With proper choice of time step that assumption is valid. One must simply be careful to pick a time step that is small relative to the rate of change of the system. The problem of stiffness arises when a small portion of the system is changing much more rapidly than the whole system. This happens most frequently in near collisions between two or more stars,

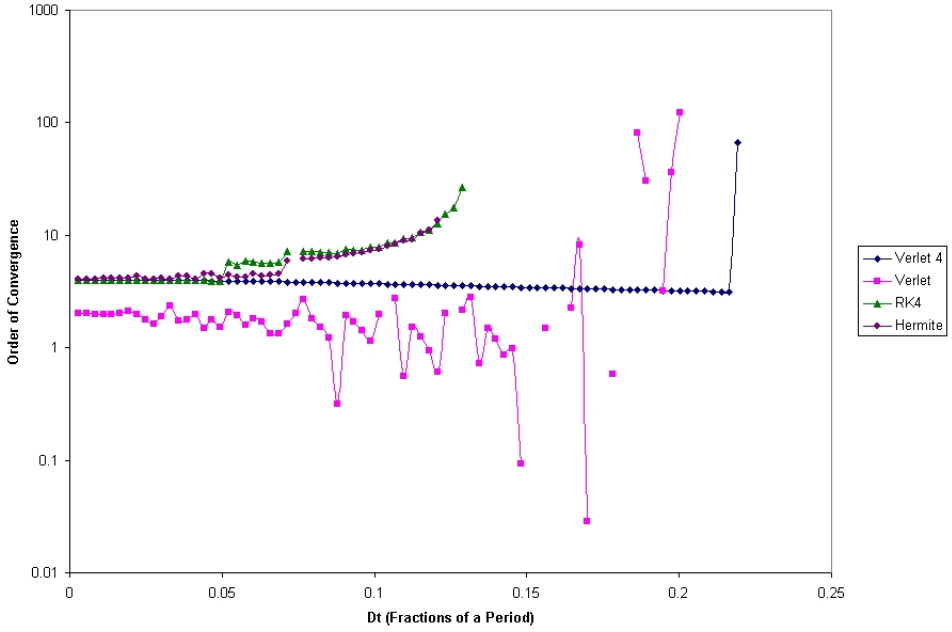

Figure 2.4. Order of Convergence

or when binary systems are formed.

An example of this is if two stars were captured into a binary with a period of a year, it would be necessary to evaluate them with a time step smaller than two months. If the whole system requires a time step of 10 years and the local system requires a time step of two months, it is not practical to treat the local system correctly.

A similar problem arises when two stars are on a near collision course. When the stars are far away, the forces on them are relatively constant and the numerical methods work well. However, as the stars get closer together the velocities grow to where the acceleration at the end of a time step is significantly greater than the acceleration at the beginning, or what it would have been in the middle. Now imagine our two stars passing each other in the middle of the time step. In reality the force

changes direction. However in our model the force does not change direction until the next time step. This causes the particle to gain energy dramatically. Often the stars gain escape velocity and are shot away from each other.

It is possible to handle the two body case with a quasi analytical solution[2]. However, a simpler approach is to use a dynamic time step. Still there is the possibility of an interaction that would cause the time step to approach zero. To limit the possibility of this happening it is common to use a regulated force. That is instead of a $\frac{1}{R^2}$-type force, use a $\frac{1}{(R+c)^2}$ force where c is a constant. Such an approach is called softening (Aarseth, 2003).

2.4 Details of the Numerical Solution

The following subsection describes the implementation of scaling, adaptive time stepping, softening, and binary counting.

2.4.1 Scaling of Units

A computer performs floating point operations most accurately when working with numbers that are near one. To avoid needless floating point errors it is therefore necessary to scale the quantities of a problem so that they are around one. This is most easily done by choosing units that are natural to the problem. Having done so, all units will drop out of the calculation. To see this start with equation 1.1, which when combined with the force equation 1.2 is:

$$\frac{d^2\overrightarrow{r}_i}{dt^2} = \sum_{j \neq i} -\frac{Gm_j}{|\vec{r_i} - \vec{r_j}|^2}\hat{R_{ij}} \tag{2.15}$$

[2]See chapter 3

Now define units of time.

$$t_0 = \sqrt{\frac{r_0^3}{Gm_0}}. \tag{2.16}$$

Where $m_0 = \bar{m}$ is the average mass and r_0 is a characteristic length scale of the distribution.

These units define three unitless quantities which we will use to perform the actual calculations in code.

$$\mu = \frac{m}{m_0} \tag{2.17}$$

$$\tau = \frac{t}{t_0} \tag{2.18}$$

$$\rho = \frac{r}{r_0} \tag{2.19}$$

In these dimensionless parameters equation 2.15 becomes

$$\frac{r_0 d^2\overrightarrow{\rho}_i}{dt^2} = \sum_{j \neq i} -\frac{G\mu_j m_0}{|\vec{\rho_i} - \vec{\rho_j}|^2 r_0^2} \hat{R_{ij}}. \tag{2.20}$$

Which doesn't seem like an improvement. However we have not yet considered the time.

$$\begin{aligned} \frac{d^2\overrightarrow{\rho}_i}{dt^2} &= \frac{d^2\overrightarrow{\rho}_i}{d\tau^2} \times \left(\frac{d\tau}{dt}\right)^2 \\ &= \frac{1}{t_0^2} \times \frac{d^2\overrightarrow{\rho}_i}{d\tau^2} \end{aligned}$$

So equation 2.20 becomes

$$\frac{d^2\overrightarrow{\rho}_i}{d\tau^2} = \sum_{j\neq i} -\frac{t_0^2 m_0}{r_0^3}\frac{G\mu_j m_0}{|\vec{\rho_i} - \vec{\rho_j}|^2}\hat{R_{ij}}.$$

Substitute for t_0

$$\frac{d^2\overrightarrow{\rho}_i}{d\tau^2} = \sum_{j\neq i} -\frac{\frac{r_0^3}{Gm_0}m_0}{r_0^3}\frac{G\mu_j}{|\vec{\rho_i} - \vec{\rho_j}|^2}\hat{R_{ij}}$$

and simplify to get

$$\frac{d^2\overrightarrow{\rho}_i}{d\tau^2} = \sum_{j\neq i} -\frac{\mu_j}{|\vec{\rho_i} - \vec{\rho_j}|^2}\hat{R_{ij}}. \tag{2.21}$$

Which is a pure number independent of units and approximately one, assuming appropriate choice of r_0 and m_0 . Calculations are now performed in these coordinates with equation 2.21.

2.4.2 Adaptive Time Stepping

All of the methods tested depend on a time step. Selecting a fixed time step would be a difficult problem. A large time step could be used so that the simulation would finish quickly, however, that could result in excessive error. On the opposite extreme a very small time step could be used, which would guarantee a small error but take a long time. The situation is more difficult still since conditions within the simulation vary with time. What initially might be a very small time step could prove inadequate to handle evolving conditions. Also the system might expand to the point where the initial time step is unnecessarily small, and thus a waste of resources.

Local Adaption Methods Since it would be impractical to attempt this sort of simulation with a fixed time step, some form of adaption is needed. At first local adaption methods were attempted. Such methods break down the system into separate small systems each with their own appropriate time step. This was appealing because calculating the force is an N^2 problem. By breaking the system down into smaller systems the computations to calculate the force are of order $\sum_i n_i^2$ where $N = \sum_i n_i$, and $N^2 \geq \sum_i n_i^2$. Figure 2.5 illustrates this concept.

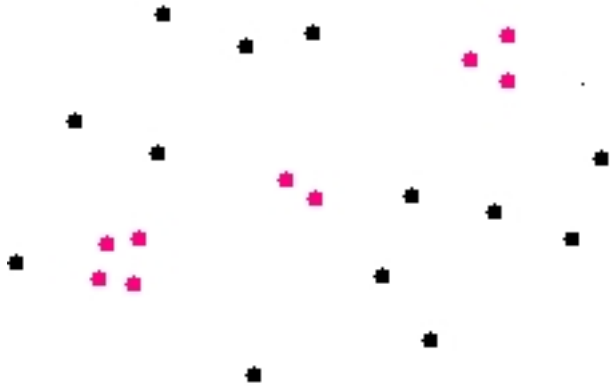

Figure 2.5. Small Systems

In figure 2.5 the red dots are to represent small clusters of tightly bound stars, and the black dots are the widely spaced main distribution of stars. Therefore the motion of each group of red stars would be determined by their local forces, a small time step, and an acceleration on the center of mass caused by all the other stars. Meanwhile the black stars would march along at a large time step saving on computations.

This method proved to be inappropriate. The results of test problems using local adaption methods differed significantly from those where a fixed, small, global time step was used. Accordingly, such local adaption methods were rejected in favor of a

more conventional but more resource intensive global adaption approach.

Global Adaption Methods To use a conventional global adaption approach first select an appropriate target error ϵ_T. This is the error that is acceptable for any given time step. Obviously this should be a small number, however, as $\epsilon_T \to 0$ the real time required for the simulation to run approaches infinity. Conversely as ϵ_T becomes large the system creates more error and results become unreliable. Some trial and error is required for selecting ϵ_T. Now calculate $r\left(t+\Delta t\right)$ twice, once with a single time step, $r\left(t+1\Delta t\right)$, and again with two time steps of $\frac{\Delta t}{2}$, $r\left(t+2\frac{\Delta t}{2}\right)$. Assume the result from the two steps to be correct and calculate the global error by

$$\epsilon = \sqrt{\sum_{i=1}^{N}\left(r\left(t+2\frac{\Delta t}{2}\right)-r\left(t+1\Delta t\right)\right)^2}. \tag{2.22}$$

Now calculate a new time step by the equation

$$\Delta t_{i+1} = \Delta t_i\left(\frac{\epsilon_T}{\epsilon}\right)^{\gamma},$$

where $\gamma < 1$ and chosen to reflect the order of convergence of the method being used. For this simulation $\gamma = 0.2$ was used. Naturally since you have already calculated $r\left(t+2\frac{\Delta t}{2}\right)$ this value should be used as $r\left(t+\Delta t\right)$.

2.4.3 Softening

Softening is the approach where the normal $\frac{1}{R^2}$ type force, is replaced by a force of the type

$$\frac{1}{\left(R+c\right)^2}. \tag{2.23}$$

This is justifiable because it only significantly affects the force in the neighborhood around $R = c$. Figure 2.6 is a plot of the percent error made by the softening equation as a function of R measured in units of c.

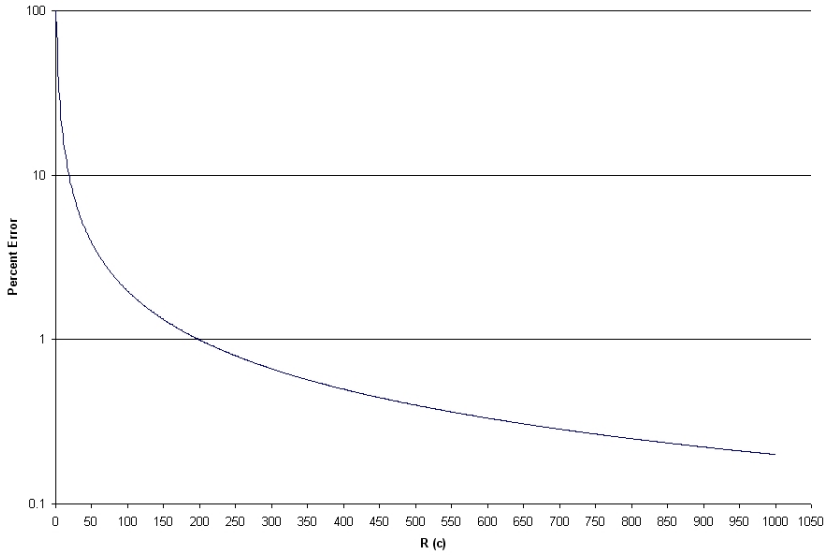

Figure 2.6. Softening Error

Since real stars are not point particles the $\frac{1}{R^2}$ type force is invalid as R approaches the radius of the stars themselves. In such a case frictional and tidal forces would be prevalent. Since this study makes no attempt at simulating frictional or tidal force between stars, the actual value of the force between two stars in such a close approach is irrelevant, and thus the softening approach is a valid method for restricting numerical error.

2.4.4 Binary Counting

In order to study binary formation it is necessary to define what a binary is. For the purpose or this study we have defined a binary system as two stars which are bound, that is have a negative total energy in their center of mass frame, and who's nearest neighboring star is grater than three times the pair radius away.

To express this mathematically begin by defining the local energy as

$$E_{ij} = \frac{1}{2} m_i V_{i,\mathrm{cm}}^2 + \frac{1}{2} m_j V_{j,\mathrm{cm}}^2 - \frac{G m_i m_j}{\left|\overrightarrow{R_{ij}}\right|}.$$

Now define the distance to the nearest neighbor as

$$R_k = \min_{k \neq i, k \neq j} \left(\min \left(R_{ik}, R_{jk} \right) \right).$$

Finally count the pair as a binary

$$\mathrm{IF}\left(\left(R_k < 3 R_{ij} \right) \mathrm{AND} \left(E_{ij} < 0 \right) \right). \tag{2.24}$$

Chapter 3

QUASI ANALYTICAL SOLUTION

Even with the best methods close encounters between stars can take too much effort to treat accurately. Such encounters can take thousands more calculations than the rest of the system. Then once a binary forms it is necessary to continue to perform those thousands of calculations just to ensure that the binary does not spontaneously ionize due to numerical error. To handle such situations we have developed a quasi analytical solution to the two body interaction. Even though we did not elect to use this technique in our final calculations, the details of the method are presented here for future use.

3.1 Kepler's Problem

The Newtonian two body problem has an analytical solution defined by Kepler's equation

$$\frac{\alpha}{|\vec{r}|} = 1 + \epsilon \cos\theta, \tag{3.1}$$

where:

- $\alpha = \frac{l^2}{\mu k}$;
- $\epsilon = \sqrt{1 + \frac{2El^2}{\mu k^2}}$;
- $\mu = \frac{m_1 m_2}{m_1 + m_2}$ =the reduced mass;

- $k = Gm_1m_2$;
- l =the two body angular momentum;
- E =the total local energy;
- θ =the angle of orbital rotation measured from the point of closest approach.

This solution parameterizes the system in terms of the angle of rotation about the center of mass θ and the distance between the two stars $|\vec{r}|$. It is therefore possible to translate two stars into the center of mass frame, rotate them, and then translate them back into the lab frame. The cost for the analytical solution is the absence of explicit time behavior.

3.2 Kepler's Problem Solved in Time

As shown by Goldstein (2002), one can solve for time in terms of an integral over theta using conservation of angular momentum.

$$t = \frac{l^3}{\mu k^2} \int_{\theta_0}^{\theta_1} \frac{d\theta}{1 + \epsilon \cos(\theta - \theta')}, \tag{3.2}$$

where $\theta'$is the angle of closest approach and Kepler's convention $\theta' = 0$.

Evaluating this integral, yields

$$t = \frac{l^3}{\mu k^2} \frac{2\left(\operatorname{arctanh}\frac{(\epsilon-1)\tan\frac{\theta_0}{2}}{\sqrt{\epsilon^2-1}} - \operatorname{arctanh}\frac{(\epsilon-1)\tan\frac{\theta_1}{2}}{\sqrt{\epsilon^2-1}}\right)}{(\epsilon^2-1)^{\frac{3}{2}}} + \frac{\epsilon\left(\frac{\sin\theta_1}{1+\epsilon\cos\theta_1} - \frac{\sin\theta_0}{1+\epsilon\cos\theta_0}\right)}{\epsilon^2-1}, \tag{3.3}$$

which is valid for $\epsilon > 1$. For $\epsilon < 1$, $\sqrt{\epsilon^2 - 1}$ is imaginary. Using the identity $\operatorname{arctanh}(ix) = i \arctan x$, equation 3.3 simplifies to

$$t = \frac{l^3}{\mu k^2} \frac{2\left(\arctan\frac{(\epsilon-1)\tan\frac{\theta_0}{2}}{\sqrt{1-\epsilon^2}} - \arctan\frac{(\epsilon-1)\tan\frac{\theta_1}{2}}{\sqrt{1-\epsilon^2}}\right)}{(1-\epsilon^2)^{\frac{3}{2}}} + \frac{\epsilon\left(\frac{\sin\theta_1}{1+\epsilon\cos\theta_1} - \frac{\sin\theta_0}{1+\epsilon\cos\theta_0}\right)}{\epsilon^2-1}. \quad (3.4)$$

The result is transcendental in θ_1 so inverting the equation analytically is impossible. Regardless, for any given time it is possible to numerically solve the equation for θ_1 using Newton's method. Once you obtain θ_1, all you need to do is rotate the two stars about the center of mass to that angle, translate the center of mass according to its velocity and distant forces, and then translate the system back into the lab frame.

3.3 Translation From Center of Mass Frame to Lab Frame

The translation from the lab frame to the center of mass frame is covered in most mechanics texts and will not be reproduced here (Goldstein, 2002). What is not typically covered explicitly is the translation from the center of mass frame to the lab frame.

First define:

- $\vec{N}$ as the vector normal to the plane of the orbit;
- $\vec{r}^{1}_{\text{cm}}(t)$ and $\vec{r}^{2}_{\text{cm}}(t)$ as the position of star 1 and 2 in the center of mass frame at t;
- $\vec{C}$ as the position of the center of mass;
- $\vec{V}_{\text{cm}}$ as the velocity of the center of mass;
- $\vec{A}_{\text{cm}}$ as the acceleration of the center of mass due to external forces.

Now solve for θ_1 using ether equation 3.3 or 3.4. Find $\hat{r}^2_{\rm cm}(t+\Delta t)$ by rotating $\vec{r}^2_{\rm cm}(t)$ about $\vec{N}$ an angle $\theta_1 - \theta_0$. Use equation 3.1 to find $|\vec{r}|$ at θ_1. Find

$$\left|\vec{r}^1_{\rm cm}(t+\Delta t)\right| = \frac{m_2\,|\vec{r}|}{m_1+m_2} \tag{3.5}$$

$$\left|\vec{r}^2_{\rm cm}(t+\Delta t)\right| = \frac{m_1\,|\vec{r}|}{m_1+m_2}. \tag{3.6}$$

Next $\vec{r}^1_{\rm cm}(t+\Delta t) = -\left|\vec{r}^1_{\rm cm}(t+\Delta t)\right| \hat{r}^1_{\rm cm}(t+\Delta t)$ and $\vec{r}^2_{\rm cm}(t+\Delta t) = \left|\vec{r}^2_{\rm cm}(t+\Delta t)\right| \hat{r}^1_{\rm cm}(t+\Delta t)$. Finally translate the center of mass by

$$\overrightarrow{r_{\rm cm}} = \overrightarrow{r_{\rm cm}} + \overrightarrow{V_{\rm cm}}t + \frac{1}{2}\vec{A}_{\rm cm}t^2 \tag{3.7}$$

$$\vec{V_{\rm cm}} = \vec{V}_{\rm cm} + \vec{A_{\rm cm}}t. \tag{3.8}$$

Which is accurate in the approximation that $\vec{A}_{\rm cm}$ is constant. Finally translate back to the lab frame with $\vec{r_1} = \vec{r}^1_{\rm cm}(t+\Delta t) + \vec{C}$.

3.4 Limitations of This Approximation

The above calculation is only an approximation. The analytical solution is only strictly valid in a non accelerating reference frame.

Figure 3.1 depicts three stars and the vectors between them. The force on star 1 is:

$$\vec{F}_1 = \vec{F}_{12} + \vec{F}_{13}$$

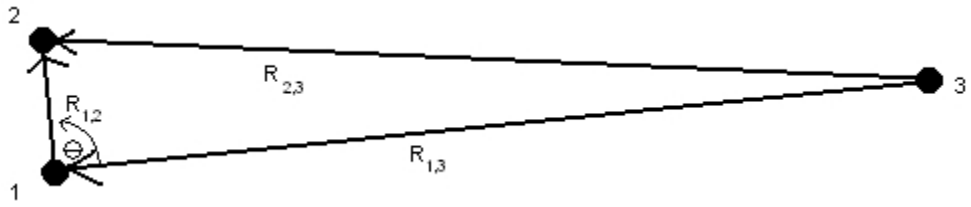

Figure 3.1. R Diagram

$$\begin{aligned}
\left|\vec{F}_1\right| &= \left|\vec{F}_{12} + \vec{F}_{13}\right| \\
&= \sqrt{\left|\vec{F}_{12}\right|^2 + 2\vec{F}_{12}\cdot\vec{F}_{13} + \left|\vec{F}_{13}\right|^2} \\
&= \sqrt{\frac{k_{12}^2}{\vec{R}_{12}^4} + 2\frac{k_{12}k_{13}}{R_{12}^2 R_{13}^2}\cos\theta + \frac{k_{13}^2}{\vec{R}_{13}^4}} \\
&= \frac{1}{R_{12}^2}\sqrt{k_{12}^2 + 2\frac{k_{12}k_{13}R_{12}^2}{R_{13}^2}\cos\theta + \frac{k_{13}^2 R_{12}^4}{\vec{R}_{13}^4}}
\end{aligned}$$

Now recall the Taylor series expansion

$$\sqrt{a+b} = \sqrt{a} + \frac{b}{2\sqrt{a}} + \ldots \tag{3.9}$$

Applying equation 3.9 with $a = k_{12}^2$ and $b = 2\frac{k_{12}k_{13}R_{12}^2}{R_{13}^2}\cos\theta + \frac{k_{13}^2R_{12}^4}{\vec{R}_{13}^4}$ yields

$$\left|\vec{F}_1\right| \approx \frac{1}{R_{12}^2}\left(k_{12} + \frac{1}{2k_{12}}\left(2\frac{k_{12}k_{13}R_{12}^2}{R_{13}^2}\cos\theta + \frac{k_{13}^2R_{12}^4}{\vec{R}_{13}^4}\right)\right)$$

$$\left|\vec{F}_1\right| = \left|\vec{F}_{12}\right| + O\left(\frac{\left|\vec{R}_{12}\right|}{\left|\vec{R}_{13}\right|}\right)^2 \tag{3.10}$$

As can be seen in equation 3.10 the above solution creates an error proportional to $\left(\frac{\left|\vec{R}_{12}\right|}{\left|\vec{R}_{13}\right|}\right)^2$. This can be generalized to the case where $\vec{R}_{13}$ is not the vector to a star but to the center of mass of several stars. The approximation is valid in the case where $R_{12} \ll R_{13}$. In such a case $\frac{R_{12}}{R_{13}} \ll 1$ and $\left(\frac{R_{12}}{R_{13}}\right)^2$ is even more so. As such the motion of the two stars relative to one another is dominated by local interactions which can be approximated by the above calculation. Conversely, to the other stars in the system the binary is very much like a single star and thus the center of mass can be translated accordingly.

Chapter 4

CALCULATION OF BINARY FORMATION RATE

4.1 Initial conditions of a Star Cluster

The following subsections describe the selection of initial conditions so as to resemble those of an actual star cluster. Considered are the mass, spatial, and energy distributions. For all distributions a simple numerical integration technique has been used. The midpoint approximation is used to approximate an integral by

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{c} f\left(a + \frac{b-a}{c2} + (i-1)\frac{b-a}{c}\right)\frac{b-a}{c}, \tag{4.1}$$

where c is an arbitrary, small number of bins, and f is a generic function. Of course as $c \to \infty$ this approximation becomes exact. Figure 4.1 is a graphical representation of this method.

4.1.1 Mass Distribution

The initial mass function (IMF) of stars has been shown by Larson (2003) to follow a three term power law. For the purpose of simplicity we will limit our stars to fall between half a solar mass and ten solar masses, which is the domain of one term of the power law

$$n(m) = a\left(\frac{m}{m_0}\right)^{-x}, \tag{4.2}$$

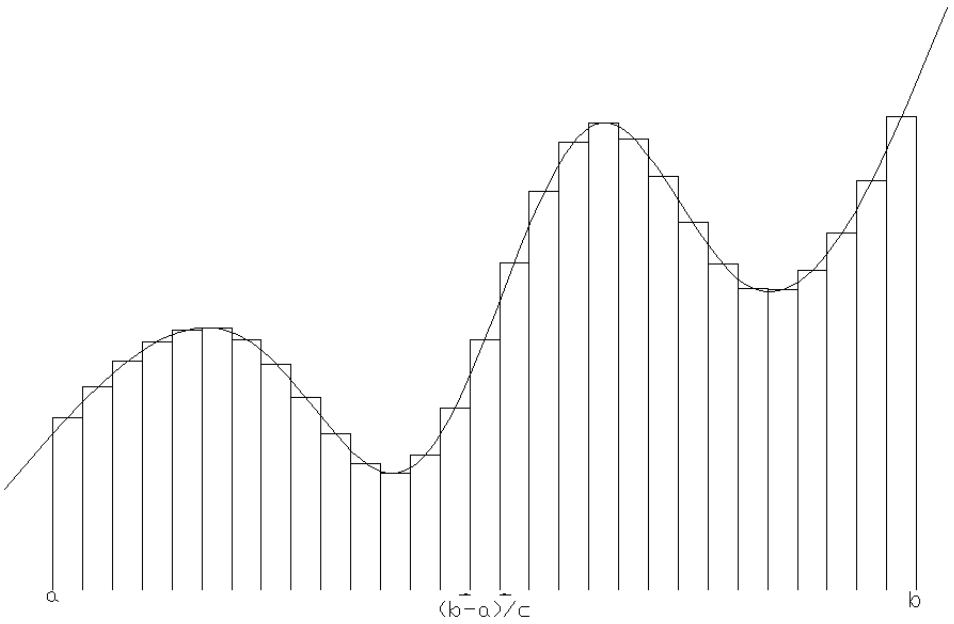

Figure 4.1. Midpoint Binning

where a is a normalization constant, m_0 is a characteristic mass, and $x = 1.35$ (Larson, 2003). To normalize it is necessary to select an upper and lower bound for the mass, m_{max} and m_{min} respectively. Now evaluate

$$N = \int_{m_{min}}^{m_{max}} n\left(m\right)$$

and solve for a.

$$a = \frac{N}{2.85714\left(\frac{m_{min}}{1m_{min}^{1.35}} - \frac{m_{max}}{m_{max}^{1.35}}\right)}$$

Use equation 4.2 and the midpoint approximation to select N masses and distribute them randomly over the N stars.

4.1.2 Spatial Distribution

We wish to distribute the stars in such a way that they obey the equation described by Chen (1998) which is a fit to observational data.

$$\rho(r) = \rho_0 \exp\left(-\frac{r}{r_0}\right), \tag{4.3}$$

where $\rho(r)$ is the mass density of stars. To achieve this, first select stellar masses in such a way that they obey equation 4.2. The total number of stars can be calculated from equation 4.3 by multiplying by r^2 and integrating over all space.

$$\begin{aligned} N &= \int_0^\infty r^2 \rho_0 \exp\left(-\frac{r}{r_0}\right) \\ &= 2r_0^3 \rho_0. \end{aligned}$$

Therefore $\rho_0 = \frac{N}{2r_0^3}$ and the number density of stars is given by

$$n(r) = \frac{N}{2r_0^3} r^2 \exp\left(-\frac{r}{r_0}\right). \tag{4.4}$$

As this distribution extends over all space, it is numerically necessary to define an outer bound. That bound is called r_1, and is defined as the radius such that

$$\begin{aligned} 1 &= \int_{r_1}^\infty n(r)\, dr \\ &= \frac{N}{2r_0^2} r^2 \exp\left(-\frac{r_1}{r_0}\right)\left(2r_0^2 + 2r_0 r_1 + r_1^2\right) \end{aligned}$$

Numerically solve this equation for r_1. Now using r_1 as an upper bound, use

equation 4.4 and the midpoint approximation to select the number of stars at any given radius. The actual position of any given star is then given by a random unit vector times the selected radius.

4.1.3 Energy Distribution

To provide a reasonable velocity distribution first apply the Virial theorem

$$E_{\text{kin}} = -\frac{U}{2}. \tag{4.5}$$

Where E_{kin} is the total kinetic energy and U is the total potential energy.

Assume that the kinetic energy E_{kin} follows a Boltzmann distribution of the form

$$n\left(E\right) = AE^2 \exp\left(-\frac{E}{kT}\right). \tag{4.6}$$

Normalizing so that the total energy is $3NkT$ gives $A = \frac{N}{2k^3T^3}$. Knowing the total energy from the Virial theorem, we solve for T.

Again we are confronted with a distribution that is infinite. Similarly to the r-distribution, we will limit this distribution to be from $0 \rightarrow E_1$ where

$$\begin{aligned} 1 &= \int_{E_1}^{\infty} n\left(E\right) dE \\ &= \frac{N}{2k^2T^2} \exp\left(-\frac{E_1}{kT}\right)\left(2E_1^2 + 2E_1kT + 2k^2T^2\right). \end{aligned}$$

We numerically solve this equation for E_1. Now using E_1 as an upper bound, we use equation 4.6 and the midpoint approximation to select the number of stars at any given kinetic energy. That energy connects directly with a velocity by the

equation $E_{\text{kin}} = \frac{1}{2}mV^2$ which determines the magnitude of the velocities for all stars. Similarly to the r-distribution, the actual velocity is then given by a random unit vector multiplied by that magnitude.

4.2 Details of the Simulation Runs

During extensive test runs, it was decided that the process of simulating a star cluster was affected by random computer error. Three people running three independent simulators with the same initial conditions would agree for only a short while, after which the average results were similar, however exact details were different. That is the average number of binaries was similar for all three simulators, however which stars were binaries at which times varied somewhat. Accordingly, it was decided to study this problem in a statistical way. The results presented are thus averages over many simulations. Also given the statistical nature of this simulation the quasi analytical solution was abandoned in favor of a simpler softening approach. Finally, the need for an adaptive method required the selection of RK4 for these runs.

4.3 Numerical Results

Tables 4.1 and 4.2 summarize the results of the 59 data runs of 200 stars and the 16 data runs of 100 stars respectively. For the sake of eloquent presentation the columns have been labeled with letters only. Table 4.3 lists the descriptions and units of the column contents.

A	B	C	D	E	F	G	H	I	J	K	L	M
0.1	10	1	0.08	1.93	2.70	1.93	2.70	0.50	4.54E+03	1.10E-02	5.46	3.03
0.5	10	1	0.89	1.57	2.12	1.57	2.12	0.60	4.22E+04	9.19E-02	5.58	3.01
1	10	1	2.53	1.23	1.62	1.23	1.62	-0.20	1.29E+05	1.64E-01	6.15	3.23
1	2	3	7.59	1.78	2.21	1.78	2.21	-0.10	1.22E+05	1.56E-01	7.62	3.52
1.5	10	1	4.65	1.31	1.68	1.31	1.68	0.20	1.69E+05	2.69E-01	7.19	3.66
2	5	1	7.14	1.67	2.15	1.67	2.15	1.20	3.61E+05	3.86E-01	6.36	3.88
3	5	1	13.16	1.15	1.78	1.15	1.78	0.20	4.59E+05	4.94E-01	6.48	3.85
4	5	1	20.21	1.74	2.52	1.74	2.52	1.00	8.69E+05	6.34E-01	7.30	4.57
10	2	1	80.01	1.04	1.96	1.04	1.96	0.50	3.78E+06	1.65E+00	7.41	4.80
Min				1.04	1.62	1.04	1.62	-0.20	4.54E+03	1.10E-02	5.46	3.01
Max				1.93	2.70	1.93	2.70	1.20	3.78E+06	1.65E+00	7.62	4.80
Average				1.49	2.08	1.49	2.08	0.43	6.60E+05	4.28E-01	6.62	3.73

Table 4.1. Numerical Results for 200 Star Runs

A	B	C	D	E	F	G	H	I	J	K	L	M
0.1	4	3	0.24	1.68	2.04	3.36	4.08	-1.50	5.57E+03	1.64E-02	7.48	4.64
1	5	3	7.59	1.31	1.64	2.62	3.28	0.00	1.53E+05	1.62E-01	7.74	4.99
2	5	3	21.43	1.07	1.60	2.14	3.19	-0.20	4.17E+05	3.05E-01	7.45	4.44
4	2	3	60.63	2.15	3.46	4.31	6.93	0.00	1.55E+06	6.23E-01	7.49	3.90
Min				1.07	1.60	2.14	3.19	-1.50	5.57E+03	1.64E-02	7.45	3.90
Max				2.15	3.46	4.31	6.93	0.00	1.55E+06	6.23E-01	7.74	4.99
Average				1.55	2.19	3.11	4.37	-0.43	5.30E+05	2.77E-01	7.54	4.49

Table 4.2. Numerical Results for 100 Star Runs

Column Label	Contents
A	Core Radius (LY)
B	Number of Measurements (#)
C	Simulated Time (Code Units)
D	Simulated Time (Millions of Years)
E	Average Number of Binaries (#)
F	Upper 95% Confidence Limit of the Mean (#)
G	Average Percent of Stars in Binary Systems (%)
H	Average Percent of Stars in Binary Systems using UCL (%)
I	Average Initial Binaries - Final Binaries (#)
J	Average Binary Life Time (Years)
K	Average Binary Radius (Light Years)
L	Average Primary Mass (Solar Mass)
M	Average Secondary Mass (Solar Mass)

Table 4.3. Column Heading for Tables 4.1 and 4.2

Of note in table 4.1 is that the maximum average number of binaries, even using the 95% upper confidence limit (UCL) is 1.35%. Also notice that the average change in binaries, that is the average of column I, is positive 0.40; indicating a net tendency to form binaries.

4.4 Comparison to Observational Data

Table 4.4 summarizes observation of the percent of stars that are in binary systems in star clusters. Binaries in stars clusters are quite common, with a minimum of 26.31% of stars in binary system. Even using the upper 95% confidence interval of the mean the largest simulated number of binaries is only 6.93%.

System	Observed Points	Binaries	Total Stars	Stars in Binaries	% Binary	Source
Hyades	197	76	273	152	55.68	(Perryman, 1998)
Orion Nebula Cluster	26	4	30	8	26.67	(Abt, 1999)
Alpha Persei	14	4	18	8	44.44	(Abt, 1999)
IC 4665	15	4	19	8	42.11	(Abt, 1999)
Praescape	21	9	30	18	60.00	(Abt, 1999)
Coma	18	6	24	12	50.00	(Abt, 1999)
NGC 2024,NGC2068,NGC 2071	99	15	114	30	26.32	(Peter, 1998)

Table 4.4. Percent of Stars in Binary Systems

4.5 Conclusions of the Study

Given that the UCL for the maximum simulated number of binaries is 6.93% and the minimum number of binaries found in a studied cluster is 26.32% it is clear that dynamic interactions are not the primary source of binary stars in open star clusters. Binaries in clusters must be explained some other way. Most likely they would be due to primordial binaries.

Chapter 5

WORK FOR THE FUTURE

Given the clear conclusion of this study, being that dynamic interactions are not the primary cause of binary stars, there is little more to be done in that respect. However given an n-body code there are more things to study other than binary formation. Interesting things might be:

- Evolution of the cluster core radius.
- Evolution of the shape of the cluster. Perhaps a net angular momentum will cause it to flatten out.
- Evolution of primordial binary orbit features such as radius or period.
- The conditions resulting in the formation of a binary star.
- Binary evolution in a binary rich cluster.
- Binary formation in smaller clusters.

REFERENCES

Aarseth, Sverre J. 2003. *Gravitational N-Body Simulations*. Cambrige University Press.

Abt, Helmut A. 1999. Binaries in the Praesepe and Coma Star Clusters and Their Implications for Binary Evolution. *The Astrophysical Journal*, **521**, 682.

Chen, B. 1998. The spatial distribution and luminosity function of the open cluster NGC 4815. *Astronomy and Astrophysics*, **331**, 916–924.

Cheney, Ward. 1999. *Numerical Mathematics and Computing*. Brooks/Cole Publishing Company.

Erolessi, Furio. 1997. *The Verlet Algorithm*. http://www.fisica.uniud.it/˜ercolessi/md/md/mode21.html.

Goldstein, Herbert. 2002. *Classical Mechanics Third Edition*. Addison Wesley.

Koonin, S., & Meredith, D. 1990. *Computational Physics*. Addison-Wesley,Reading.

Kroupa, Pavel. 2001. On the Origin of the Distribution of Binary Star Periods. *The Astrophysical Journal*, **555**, 945.

Larson, Richard B. 1997. *Formation of Small and Large Stellar Systems*. Structure and Evolution of Stellar Systems, proceedings of a conference in Petrozavodsk, Russia.

Larson, Richard B. 2003. *The Stellar Initial Mass Function and Beyond*. Galactic Star Formation Across the Stellar Mass Spectrum ASP Conference Series Vol 287.

Perryman, M. 1998. The Hyades: distance, structure, dynamics, and age. *Astronomy and Astrophysics*, **331**, 681–686.

Peter, Monka. 1998. Binary Stars in the Orion Trapezium Cluster Core. *The Astrophysical Journal*, **500**, 825–837.

Zwart, Simon F. Porteges. 1997. *Stellar Eolution and Dynamics in Star Clusters*. Cornell University Library: arXiv:astro-ph/9710209 v1.

Zwart, Simon F. Porteges. 2000. *Star Cluster Ecology IVa: Dissection of an Open Star Cluster–Photometry*. Cornell University Library: arXiv:astro-ph/0005248 v1.

Zwart, Simon F. Portegies. 1998. On the Dissolution of Evolving Star Clusters. *Astronomy and Astrophysics*, **337**, 363–371.

APPENDIX A

DERIVATION OF ODE SOLVING TECHNIQUES

The main text provided the basic equations for all of the methods. However, the derivation of the methods is interesting, and as such is contained here.

A.1 Third Order Taylor Expansion

The only interesting thing to derive related to the Taylor expansion method is the force derivative $\vec{F}'$ which is stated in equation 2.3. Begin by expressing the x component of the force explicitly as

$$Fx_{i,j} = -\frac{k_{i,j}\left(x_i\left(t\right) - x_j(t)\right)}{\left(\left(x_i\left(t\right) - x_j\left(t\right)\right)^2 + \left(y_i\left(t\right) - y\left(t\right)\right)^2 + \left(z_i\left(t\right) - z_j\left(t\right)\right)^2\right)^{\frac{3}{2}}}$$

where $k_{i,j} = Gm_i m_j$.

Take the derivative with respect to t and get

$$\begin{aligned} Fx'_{i,j} \;=\; & -\frac{k_{i,j}\left(x'_i\left(t\right) - x'_j(t)\right)}{\left(\left(x_i\left(t\right) - x_j\left(t\right)\right)^2 + \left(y_i\left(t\right) - y\left(t\right)\right)^2 + \left(z_i\left(t\right) - z_j\left(t\right)\right)^2\right)^{\frac{3}{2}}} \\ & + \frac{3k_{i,j}\left(x_i\left(t\right) - x_j(t)\right)}{\left(\left(x_i\left(t\right) - x_j\left(t\right)\right)^2 + \left(y_i\left(t\right) - y\left(t\right)\right)^2 + \left(z_i\left(t\right) - z_j\left(t\right)\right)^2\right)^{\frac{5}{2}}} \\ & *\left(\left(x_i\left(t\right) - x_j(t)\right)\left(x'_i\left(t\right) - x'_j(t)\right) + \left(y_i\left(t\right) - y_j(t)\right)\left(y'_i\left(t\right) - y'_j(t)\right)\right. \\ & \left. + \left(z_i\left(t\right) - z_j(t)\right)\left(z'_i\left(t\right) - z'_j(t)\right)\right) \end{aligned}$$

Now define:

- $Vx_{i,j} = x_i^{'}(t) - x_j^{'}(t)$,
- $Vy_{i,j} = y_i^{'}(t) - y_j^{'}(t)$,
- $Vz = z_i^{'}(t) - z_j^{'}(t)$,
- $Rx_{i,j} = x_i(t) - x_j(t)$,
- $Ry_{i,j} = y_i(t) - y_j(t)$,
- $Rz_{i,j} = z_i(t) - z_j(t)$,
- $|R_{i,j}| = \left((x_i(t) - x_j(t))^2 + (y_i(t) - y(t))^2 + (z_i(t) - z_j(t))^2\right)^{\frac{1}{2}}$

and substitute to get;

$$Fx_{i,j}^{'} = -\frac{k_{i,j} Vx_{i,j}}{|R_{i,j}|^3} + \frac{3k_{i,j} Rx_{i,j}\left(Rx_{i,j} Vx_{i,j} + Ry_{i,j} Vy_{i,j} + Rz_{i,j} Vz_{i,j}\right)}{|R_{i,j}|^5}.$$

As this is one component of a vector, the others will follow cyclic permutations and the general vector will be

$$\vec{F}_{i,j}^{'} = -\frac{k_{i,j}\overrightarrow{V}_{i,j}}{\left|\overrightarrow{R}_{i,j}\right|^3} + \frac{3k_{i,j}\overrightarrow{R}_{i,j}\left(\overrightarrow{R}_{i,j}\cdot\overrightarrow{V}_{i,j}\right)}{\left|\overrightarrow{R}_{i,j}\right|^5}$$

Where $\vec{V}_{i,j} = \vec{V}_i - \vec{V}_j$ and $\vec{R}_{i,j} = \vec{R}_i - \vec{R}_j$.
Finally define $a = \frac{\left(\vec{R}_{i,j}\cdot\vec{V}_{i,j}\right)}{\left|\vec{R}_{i,j}\right|^3}$ and simplify to get

$$\vec{F}_{i,j}^{'} = -\frac{k_{i,j}\overrightarrow{V}_{i,j}}{\left|\overrightarrow{R}_{i,j}\right|^3} - 3a\vec{F}_{i,j}. \tag{A.1}$$

A.2 Hermite

Hermite integration is described by Aarseth (2003). To derive Hermite integration begin by writing the Taylor expansion for the force and force derivatives in terms of time.

$$\vec{F}\left(t+\Delta t\right)=\vec{F}\left(t\right)+\vec{F}^{'}\left(t\right)\Delta t+\frac{1}{2}\vec{F}^{''}\left(t\right)\Delta t^{2}+\frac{1}{6}\vec{F}^{'''}\left(t\right)\Delta t^{3}+O\left(\Delta t^{4}\right),$$

$$\vec{F}^{'}\left(t+\Delta t\right)=\vec{F}^{'}\left(t\right)+\vec{F}^{''}\left(t\right)\Delta t+\frac{1}{2}\vec{F}^{'''}\left(t\right)\Delta t^{2}+O\left(\Delta t^{3}\right).$$

Now solve the $\vec{F}^{'}\left(t+\Delta t\right)$ equation for $\vec{F}^{''}\left(t\right)$ and get

$$\vec{F}^{''}\left(t\right)=\frac{1}{\Delta t}\left(\vec{F}^{'}\left(t+\Delta t\right)-\vec{F}^{'}\left(t\right)-\frac{1}{2}\vec{F}^{'''}\left(t\right)\Delta t^{2}+O\left(\Delta t^{3}\right)\right)^{'}.$$

Substitute this into the $\vec{F}\left(t+\Delta t\right)$ equation and simplify

$$\begin{aligned}\vec{F}\left(t+\Delta t\right) &= \vec{F}\left(t\right)+\vec{F}^{'}\left(t\right)\Delta t \\ &+\frac{1}{2}\left(\vec{F}^{'}\left(t+\Delta t\right)-\vec{F}^{'}\left(t\right)-\frac{1}{2}\vec{F}^{'''}\left(t\right)\Delta t^{2}+O\left(\Delta t^{3}\right)\right)\Delta t \\ &+\frac{1}{6}\vec{F}^{'''}\left(t\right)\Delta t^{3}+O\left(\Delta t^{4}\right),\end{aligned}$$

$$\vec{F}\left(t+\Delta t\right)=\vec{F}\left(t\right)+\frac{1}{2}\vec{F}^{'}\left(t\right)\Delta t+\frac{1}{2}\vec{F}^{'}\left(t+\Delta t\right)\Delta t-\frac{1}{12}\vec{F}^{'''}\left(t\right)\Delta t^{3}+O\left(\Delta t^{4}\right).$$

Solve for $\vec{F}^{'''}\left(t\right)$

$$\vec{F}'''(t) = \frac{12}{\Delta t^3}\left(\vec{F}(t) - \vec{F}(t+\Delta t) + \frac{1}{2}\vec{F}'(t)\Delta t + \frac{1}{2}\vec{F}'(t+\Delta t)\Delta t + O\left(\Delta t^4\right)\right),$$

$$\vec{F}'''(t) = \frac{6}{\Delta t^3}\left(2\left(\vec{F}(t) - \vec{F}(t+\Delta t)\right) + \left(\vec{F}'(t) + \vec{F}'(t+\Delta t)\right)\Delta t\right) + O(\Delta t).$$

Substitute this expression into the first expression for $F(t+\Delta t)$ and solve for $F''(\Delta t)$

$$\begin{aligned}\vec{F}(t+\Delta t) &= \vec{F}(t) + \vec{F}'(t)\Delta t + \frac{1}{2}\vec{F}''(t)\Delta t^2 \\ &\quad + \left(2\left(\vec{F}(t) - \vec{F}(t+\Delta t)\right) + \left(\vec{F}'(t) + \vec{F}'(t+\Delta t)\right)\Delta t\right) + O\left(\Delta t^4\right),\end{aligned}$$

$$3\vec{F}(t+\Delta t) = 3\vec{F}(t) + 2\vec{F}'(t)\Delta t + \frac{1}{2}\vec{F}''(t)\Delta t^2 + \vec{F}'(t+\Delta t)\Delta t + O\left(\Delta t^4\right),$$

$$\vec{F}''(t) = \frac{2}{\Delta t^2}\left(-3\vec{F}(t) - 2\vec{F}'(t)\Delta t - \vec{F}'(t+\Delta t)\Delta t + 3\vec{F}(t+\Delta t)\right) + O\left(\Delta t^2\right).$$

Now consider the Taylor series expansion for $\vec{r}(t+\Delta t)$ and $\vec{v}(t+\Delta t)$

$$\begin{aligned}\vec{r}(t+\Delta t) &= \vec{r}(t)+\vec{r}^{\,\prime}(t)\,\Delta t+\frac{1}{2}\vec{r}^{\,\prime\prime}(t)\,\Delta t^2+\frac{1}{6}\vec{r}^{\,\prime\prime\prime}(t)\,\Delta t^3+\frac{1}{24}\vec{r}^{\,\prime\prime\prime\prime}(t)\,\Delta t^4\\ &\quad+\frac{1}{120}\vec{r}^{\,\prime\prime\prime\prime\prime}(t)\,\Delta t^5+O\left(\Delta t^6\right)\end{aligned}$$

$$\begin{aligned}\vec{v}\,(t+\Delta t) &= \vec{v}\,(t)+\vec{v}^{\,\prime}(t)\,\Delta t+\frac{1}{2}\vec{v}^{\,\prime\prime}(t)\,\Delta t^2+\frac{1}{6}\vec{v}^{\,\prime\prime\prime}(t)\,\Delta t^3+\\ &\quad\frac{1}{24}\vec{v}^{\,\prime\prime\prime\prime}(t)\,\Delta t^4+O\left(\Delta t^5\right)\end{aligned}$$

Now unify the notation by substituting:

- $\vec{r}^{\,\prime\prime}=\frac{\vec{F}}{m}$,
- $\vec{r}^{\,\prime\prime\prime}=\frac{\vec{F}^{\prime}}{m}$,
- $\vec{r}^{\,\prime\prime\prime\prime}=\frac{\vec{F}^{;\prime}}{m}$,
- $\vec{r}^{\,\prime\prime\prime\prime\prime}=\frac{\vec{F}^{\prime\prime\prime}}{m}$,
- $\vec{v}^{\,\prime}=\frac{\vec{F}}{m}$,
- $\vec{v}^{\,\prime\prime}=\frac{\vec{F}^{\prime}}{m}$,
- $\vec{v}^{\,\prime\prime\prime}=\frac{\vec{F}^{\prime\prime}}{m}$,
- $\vec{v}^{\,\prime\prime\prime\prime}=\frac{\vec{F}^{\prime\prime\prime}}{m}$.

$$\begin{aligned}r\,(t+\Delta t) &= \vec{r}(t)+\vec{r}^{\,\prime}(t)\,\Delta t+\frac{1}{2m}\vec{F}\,(t)\,\Delta t^2+\frac{1}{6m}\vec{F}^{\prime}(t)\,\Delta t^3\\ &\quad+\frac{1}{24m}\vec{F}^{\prime\prime}(t)\,\Delta t^4+\frac{1}{120m}\vec{F}^{\prime\prime\prime}(t)\,\Delta t^5+O\left(\Delta t^6\right),\end{aligned}$$

$$\begin{aligned} v\left(t+\Delta t\right) &= \vec{v}\left(t\right)+\frac{1}{m}\vec{F}\left(t\right)\Delta t+\frac{1}{2m}\vec{F}'\left(t\right)\Delta t^{2}+\frac{1}{6m}\vec{F}''\left(t\right)\Delta t^{3} \\ &\quad +\frac{1}{24m}\vec{F}'''\left(t\right)\Delta t^{4}+O\left(\Delta t^{5}\right). \end{aligned}$$

We know $\vec{F}$ and $\vec{F}'$ as explicit functions of the position and we previously derived expressions for $\vec{F}''$ and $\vec{F}'''$. So substitute those expressions and simplify. For r we get

$$\begin{aligned} \vec{r}\left(t+\Delta t\right) &= \vec{r}\left(t\right)+\vec{r}'\left(t\right)\Delta t+\frac{1}{2m}\vec{F}\left(t\right)\Delta t^{2}+\frac{1}{6m}\vec{F}'\left(t\right)\Delta t^{3} \\ &\quad +\frac{1}{12m}\left(3\left(\vec{F}\left(t+\Delta t\right)-\vec{F}\left(t\right)\right)-\left(2\vec{F}'\left(t\right)+\vec{F}'\left(t+\Delta t\right)\right)\Delta t\right)\Delta t^{2} \\ &\quad +\frac{1}{20m}\left(2\left(\vec{F}\left(t\right)-\vec{F}\left(t+\Delta t\right)\right)+\left(\vec{F}'\left(t\right)+\vec{F}'\left(t+\Delta t\right)\right)\Delta t\right)\Delta t^{2} \\ &\quad +O\left(\Delta t^{6}\right). \end{aligned}$$

Similarly in v we have

$$\begin{aligned} \vec{v}\left(t+\Delta t\right) &= \vec{v}\left(t\right)+\frac{1}{m}\vec{F}\left(t\right)\Delta t+\frac{1}{2m}\vec{F}'\left(t\right)\Delta t^{2} \\ &\quad +\frac{1}{3m}\left(3\left(\vec{F}\left(t+\Delta t\right)-\vec{F}\left(t\right)\right)-\left(2\vec{F}'\left(t\right)+\vec{F}'\left(t+\Delta t\right)\right)\Delta t\right)\Delta t \\ &\quad +\frac{1}{4m}\left(2\left(\vec{F}\left(t\right)-\vec{F}\left(t+\Delta t\right)\right)+\left(\vec{F}'\left(t\right)+\vec{F}'\left(t+\Delta t\right)\right)\Delta t\right)\Delta t \\ &\quad +O\left(\Delta t^{5}\right). \end{aligned}$$

Since we do not know $\vec{F}\left(t+\Delta t\right)$ or $\vec{F}'\left(t+\Delta t\right)$ we cannot use this method exactly. Instead we will do a two step method.

$$\vec{r}\left(t+\Delta t\right)=\vec{r}_t+\overrightarrow{\Delta r}.$$

$$\vec{v}\left(t+\Delta t\right)=\vec{v}_t+\overrightarrow{\Delta v}.$$

Where $\vec{r}_t$ and $\vec{v}_t$ are calculated via Taylor3 as:

$$\vec{r}_t=\vec{r}\left(t\right)+\vec{r}^{'}\left(t\right)\Delta t+\frac{1}{2m}\vec{F}\left(t\right)\Delta t^2+\frac{1}{6m}\vec{F}^{'}\left(t\right)\Delta t^3,$$

$$\vec{v}_t=\vec{v}\left(t\right)+\frac{1}{m}\vec{F}\left(t\right)\Delta t+\frac{1}{2m}\vec{F}^{'}\left(t\right)\Delta t^2,$$

and $\overrightarrow{\Delta r}$ and $\overrightarrow{\Delta v}$ are calculated as

$$\begin{aligned}\overrightarrow{\Delta r} &= \frac{1}{12m}\left(3\left(\vec{F}|_{r_t}-F\left(t\right)\right)-\left(2\vec{F}^{'}\left(t\right)+\vec{F}^{'}|_{r_t}\right)\Delta t\right)\Delta t^2\\ &+\frac{1}{20m}\left(2\left(\vec{F}\left(t\right)-\vec{F}|_{r_t}\right)+\left(\vec{F}^{'}\left(t\right)+\vec{F}^{'}|_{r_t}\right)\Delta t\right)\Delta t^2,\end{aligned}$$

$$\begin{aligned}\overrightarrow{\Delta v} &= \frac{1}{3m}\left(3\left(\vec{F}|_{r_t}-\vec{F}\left(t\right)\right)-\left(2\vec{F}^{'}\left(t\right)+\vec{F}^{'}|_{r_t}\right)\Delta t\right)\Delta t\\ &+\frac{1}{4m}\left(2\left(\vec{F}\left(t\right)-\vec{F}|_{r_t}\right)+\left(\vec{F}^{'}\left(t\right)+\vec{F}^{'}|_{r_t}\right)\Delta t\right)\Delta t.\end{aligned}$$

Which simplify to

$$\overrightarrow{\Delta r}=\Delta t^2\left[-\frac{-9\vec{F}|_{r_t}+9\vec{F}\left(t\right)+\left(7\vec{F}^{'}\left(t\right)+2\vec{F}^{'}|_{r_t}\right)\Delta t}{60m}\right],$$

$$\overrightarrow{\Delta v} = \Delta t \left[-\frac{-6\vec{F}|_{r_t} + 6\vec{F}(t) + \left\{ 5\vec{F}'(t) + \vec{F}'|_{r_t} \right\} \Delta t}{12m} \right].$$

A.3 RK4

The fourth order Runge Kutta method is a member of a family of Runge Kutta methods. The derivation of each member of the family is similar to the others. In that light, we will derive the second order method and then simply state the fourth. This follow the approach of Koonin & Meredith (1990). To begin we will discuss an arbitrary first order ODE of the form

$$\frac{\partial y}{\partial x} = f(x, y(x)).$$

We will attempt to solve this ODE with the integration of the general form

$$y_{n+1} = y_n + \int_{x_n}^{x_n + \Delta x} f(x, y(x))\, dx, \tag{A.2}$$

where $y_n = y(x)$ and $y_{n+1} = y(x + \Delta x)$.

Now approximate f by its Taylor expansion about the midpoint $z(x) = x - \left(x_n + \frac{\Delta x}{2}\right)$.

$$f(z, y(z)) = f(0, y(0)) + f'(0, y(0))\, z + O\left(z^2\right),$$

where

$$f' = \frac{\partial f}{\partial z}$$

$$f^{'}\frac{\partial z}{\partial x}=\frac{\partial f}{\partial z}\frac{\partial z}{\partial x}$$

$$f^{'}\frac{\partial z}{\partial x}=\frac{\partial f}{\partial x}$$

$$f^{'}=\frac{\partial f}{\partial x}$$

since $\frac{\partial z}{\partial x}=1$.

The integral from A.2 evaluated in z is

$$\begin{aligned}
\int_{x_n}^{x_n+\Delta x} f\left(x,y\left(x\right)\right)dx &= \int_{x_n}^{x_n+\Delta x} f\left(z,y\left(z\right)\right)dz \\
&= \int_{z-\frac{\Delta x}{2}}^{z+\frac{\Delta x}{2}} f\left(0,y\left(0\right)\right)+f^{'}\left(0,y\left(0\right)\right)z+O\left(z^2\right)dz \\
&= f\left(0,y\left(0\right)\right)z+\frac{1}{2}f^{'}\left(0,y\left(0\right)\right)z^2+O\left(z^3\right)\Big]_{x_n}^{x_n+\Delta x} \\
&= f\left(x_{n+\frac{1}{2}},y_{n+\frac{1}{2}}\right)\left(z\left(x_n+\Delta x\right)-z\left(x_n\right)\right) \\
&\quad +\frac{1}{2}f^{'}\left(x_{n+\frac{1}{2}},y_{n+\frac{1}{2}}\right)\left(z\left(x_n+\Delta x\right)^2-z\left(x_n\right)^2\right)+O\left(z^3\right) \\
&= f\left(x_{n+\frac{1}{2}},y_{n+\frac{1}{2}}\right)\left(x_n+\Delta x-x_n-\frac{1}{2}\Delta x-x_n+x_n+\frac{1}{2}\Delta x\right) \\
&= f\left(x_{n+\frac{1}{2}},y_{n+\frac{1}{2}}\right)\Delta x \\
&\quad +\frac{1}{2}f^{'}\left(x_{n+\frac{1}{2}},y_{n+\frac{1}{2}}\right)\left(\left(-\frac{1}{2}\Delta x\right)^2-\left(-\frac{1}{2}\Delta x\right)^2\right) \\
&\quad +O\left(\Delta x^3\right)
\end{aligned}$$

$$\int_{x_n}^{x_n+\Delta x} f\left(x,y\left(x\right)\right)dx=f\left(x_{n+\frac{1}{2}},y_{n+\frac{1}{2}}\right)\Delta x+O\left(\Delta x^3\right).$$

Now insert this into A.2 and get

$$y_{n+1} = y_n + f\left(x_{n+\frac{1}{2}}, y_{n+\frac{1}{2}}\right)\Delta x + O\left(\Delta x^3\right). \tag{A.3}$$

Equation A.2 is difficult in the fact that we do not know $y_{n+\frac{1}{2}}$. However, since the error term is $O\left(\Delta x^3\right)$, an approximation to $y_{n+\frac{1}{2}}$ which is $O\left(\Delta x^2\right)$ will be sufficient. An Euler approach will provide just that. So if we define

$$k = f\left(x_n, y_n\right)\Delta x.$$

Then, equation A.3 simplifies to

$$y_{n+1} = y_n + f\left(x_n + \frac{1}{2}\Delta x, y_n + \frac{1}{2}k\right)\Delta x + O\left(\Delta x^3\right). \tag{A.4}$$

Which is the second order Runge Kutta method for a first order ODE. Higher order methods can be derived by applying quadrature techniques to equation A.2, and proceeding similarly.

To apply this method to a second order ODE all that is required is to separate the equation into a form

$$\frac{\partial y}{\partial x} = v\left(x, y\right)$$

$$\frac{\partial v}{\partial x} = f\left(x, y\right)$$

and apply equation A.4 to both first order ODEs.

A.4 Verlet

The Verlet algorithm can be derived from two Taylor series.

$$\vec{r}(t+\Delta t) = \vec{r}(t) + \vec{r}^{\,\prime}(t)\,\Delta t + \frac{1}{2}\vec{r}^{\,\prime\prime}(t)\,\Delta t^2 + \frac{1}{6}\vec{r}^{\,\prime\prime\prime}(t)\,\Delta t^3 + O(\Delta t^4)$$

$$\vec{r}(t-\Delta t) = \vec{r}(t) - \vec{r}^{\,\prime}(t)\,\Delta t + \frac{1}{2}\vec{r}^{\,\prime\prime}(t)\,\Delta t^2 - \frac{1}{6}\vec{r}^{\,\prime\prime\prime}(t)\,\Delta t^3 + O(\Delta t^4)$$

Add these two equations together and get

$$\vec{r}(t-\Delta t) + \vec{r}(t+\Delta t) = 2\vec{r}(t) + \vec{r}^{\,\prime\prime}(t)\,\Delta t^2 + O(\Delta t^4).$$

Finally solve for $\vec{r}(t+\Delta t)$ as

$$\vec{r}(t+\Delta t) = 2\vec{r}(t) - \vec{r}(t-\Delta t) + \vec{r}^{\,\prime\prime}(t)\,\Delta t^2 + O(\Delta t^4),$$

which is the Verlet equation.

A.4.1 Velocity Approximation

Verlet type algorithms do not provide an explicit value of the velocity. So, it is necessary to approximate the velocity from the position. This can be done with the use of Lagrange interpolating polynomials. An $(n-1)$th order Lagrange interpolating polynomial can be generated from

$$P(x) = \Sigma_{j=1}^{n} P_j(x).$$

Where

$$P_j(x) = y_j \Pi_{k \neq j}^{n} \frac{x - x_j}{x_j - x_j},$$

and (x_j, y_j) are n known positions.

Also Mathematica will generate these polynomials with the function Interpolating Polynomial.

We wish to generate an approximation to the velocity that is good to fourth order. So we will need a fourth order polynomial and five known positions. The fourth order Lagrange interpolating polynomial is really quite ugly and can easily be generated by the reader. If you take the derivative of that polynomial and then simplify with the assumptions:

- $x_1 = x_5 - 4\Delta t$,
- $x_2 = x_5 - 3\Delta t$,
- $x_3 = x_5 - \Delta t$,
- $x_4 = x_5 - \Delta t$,
- and $x = x_5$

you arrive at the simple result of

$$x'[x_5] = \frac{3y_1 - 16y_2 + 36y_3 - 48y_4 + 25y_5}{12\Delta t}$$

which is what is listed as equation 2.12.

A.5 Verlet4

Begin with the Taylor series expansion for the position at $(t + \Delta t)$ and $(t - \Delta t)$

$$\begin{aligned}\vec{r}\,(t+\Delta t) &= \vec{r}\,(t)+\vec{r}^{\,\prime}\,(t)\,\Delta t+\frac{1}{2}\vec{r}^{\,\prime\prime}\,(t)\,\Delta t^2+\frac{1}{6}\vec{r}^{\,\prime\prime\prime}\,(t)\,\Delta t^3+\frac{1}{24}\vec{r}^{\,\prime\prime\prime\prime}\,(t)\,\Delta t^4\\ &\quad+\frac{1}{5!}\vec{r}^{\,\prime\prime\prime\prime\prime}\,(t)\,\Delta t^5+O(\Delta t^6).\end{aligned}$$

$$\begin{aligned}\vec{r}\,(t-\Delta t) &= \vec{r}\,(t)-\vec{r}^{\,\prime}\,(t)\,\Delta t+\frac{1}{2}\vec{r}^{\,\prime\prime}\,(t)\,\Delta t^2-\frac{1}{6}\vec{r}^{\,\prime\prime\prime}\,(t)\,\Delta t^3+\frac{1}{24}\vec{r}^{\,\prime\prime\prime\prime}\,(t)\,\Delta t^4\\ &\quad-\frac{1}{5!}\vec{r}^{\,\prime\prime\prime\prime\prime}\,(t)\,\Delta t^5+O(\Delta t^6).\end{aligned}$$

Now add them together and simplify to get

$$\vec{r}\,(t+\Delta t)=2\vec{r}\,(t)-\vec{r}\,(t-\Delta t)+\vec{r}\,(t)\,\Delta t^2+\frac{1}{12}\vec{r}^{\,\prime\prime\prime\prime}\,(t)\,\Delta t^4+O(\Delta t^6).$$

Define $2\vec{r}\,(t)-\vec{r}\,(t-\Delta t)+\vec{r}\,(t)\,\Delta t^2=r_V$.

$$\vec{r}\,(t+\Delta t)=r_V+\frac{1}{12}\vec{r}^{\,\prime\prime\prime\prime}\,(t)\,\Delta t^4+O(\Delta t^6) \tag{A.5}$$

However we do not know $r^{\prime\prime\prime\prime}\,(t)$ explicitly. To get $r^{\prime\prime\prime\prime}\,(t)$ first notice that $r^{\prime\prime\prime\prime}\,(t)=F^{\prime\prime}\,(t)$ where F is the force. Time for another pair of Taylor series.

$$\vec{F}\,(t+\Delta t)=\vec{F}\,(t)+\vec{F}^{\prime}\,(t)\,\Delta t+\frac{1}{2}\vec{F}^{\prime\prime}\,(t)\,\Delta t^2+\frac{1}{6}\vec{F}^{\prime\prime\prime}\,(t)\,\Delta t^3+O(\Delta t^4).$$

$$\vec{F}\left(t-\Delta t\right)=\vec{F}\left(t\right)-\vec{F}'\left(t\right)\Delta t+\frac{1}{2}\vec{F}''\left(t\right)\Delta t^{2}-\frac{1}{6}\vec{F}'''\left(t\right)\Delta t^{3}+O(\Delta t^{4}).$$

Add them together to get the Verlet equation for $\vec{F}$ instead of $\vec{r}$

$$\vec{F}\left(t+\Delta t\right)=2\vec{F}\left(t\right)+\vec{F}''\left(t\right)\Delta t^{2}-\vec{F}\left(t-\Delta t\right)+O(\Delta t^{4}).$$

Solve for $\vec{F}''$

$$-\vec{F}''\left(t\right)\Delta t^{2}=2\vec{F}\left(t\right)-\vec{F}\left(t-\Delta t\right)-\vec{F}\left(t+\Delta t\right)+O(\Delta t^{4}),$$

$$\vec{F}''\left(t\right)=\frac{\vec{F}\left(t+\Delta t\right)-2\vec{F}\left(t\right)+\vec{F}\left(t-\Delta t\right)}{\Delta t^{2}}+O(\Delta t^{2}).$$

Now substitute this into equation A.5 and simplify.

$$\vec{r}\left(t+\Delta t\right)=r_{V}+\left(\frac{\vec{F}\left(t+\Delta t\right)-2\vec{F}\left(t\right)+\vec{F}\left(t-\Delta t\right)}{\Delta t^{2}}+O(\Delta t^{2})\right)\Delta t^{4}+O(\Delta t^{6}),$$

$$\vec{r}\left(t+\Delta t\right)=r_{V}+\left(\vec{F}\left(t+\Delta t\right)-2\vec{F}\left(t\right)+\vec{F}\left(t-\Delta t\right)\right)\Delta t^{2}+O(\Delta t^{6}).$$

Since we don't know $\vec{F}\left(t+\Delta t\right)$, we make the best possible guess, that is $\vec{F}\left(t+\Delta t\right)=0$ and compute

$$\vec{r_{t1}}=\overrightarrow{r}_{V}+\frac{1}{12}\left\{\frac{1}{m}\vec{F}\left(\vec{r}(t-\Delta t)\right)-\frac{1}{m}\vec{F}\left(\vec{r}(t)\right)\right\}\Delta t^{2}.$$

Now use $\vec{r_{t1}}$ to compute

$$\vec{r_{t2}} = \overrightarrow{r}_V + \frac{1}{12}\left\{\frac{1}{m}\vec{F}\left(\vec{r}_{t1}\right) + \frac{1}{m}\vec{F}\left(\vec{r}\left(t-\Delta t\right)\right) - 2\frac{1}{m}\vec{F}\left(\vec{r}\left(t\right)\right)\right\}\Delta t^2.$$

Finally use $\vec{r_{t2}}$ to compute

$$\vec{r}\left(t+\Delta t\right) = \overrightarrow{r}_V + \frac{1}{12}\left\{\frac{1}{m}\vec{F}\left(\vec{r}_{t2}\right) + \frac{1}{m}\vec{F}\left(\vec{r}\left(t-\Delta t\right)\right) - 2\frac{1}{m}\vec{F}\left(\vec{r}\left(t\right)\right)\right\}\Delta t^2$$

which is the final Verlet4 equation.

APPENDIX B

NEWTON'S METHOD

Newton's method is an iterative numerical method of solving the equation $F(x) = 0$. As stated by Cheney (1999) as

$$x_{n+1} = x_n + \frac{F(x)}{F'(x)}.$$

As this is an iterative method it is necessary to define a starting point. The method is quite sensitive to that initial guess, so some care must be taken. Also it is necessary to define the limit where the answer is good enough. Typically that criteria is set in the form of an ϵ such that $F(x) < \epsilon$ is an acceptable approximation to $F(x) = 0$. Also sometimes this method can oscillate, never quite achieving $F(x) < \epsilon$, so it is necessary to set a maximum number of iterations after which you give up and use what you have.

Pseudo code for Newton's method is:

x=x_0
$error = 1$
$maxError = 1E - 3$
$maxIts = 20$
$i = 0$
$while((i < maxIts)And(errormax > error))$
{
$x = x - F(x)/FPrime(x)$
$error = abs(F(x))$
$i = i + 1$
}

APPENDIX C

SOURCE CODE

```
//Vector3d.cpp used throughout the project

#include <math.h>

// Local Includes
#include "Vector3d.h"

// Static Initialization
const Vector3d Vector3d::ms_Null(0.0,0.0,0.0);

//============================================================================
// Size Functions

// Size
double Vector3d::Size() const
{
// Return Euclidean Norm
return sqrt(SizeSquared());
}

//============================================================================
// Rotation Functions

// Rotate X
Vector3d Vector3d::RotateX(double dA) const
{
// Compute Cosine & Sine
double dC=cos(dA);
double dS=sin(dA);
// Compute Rotation
return Vector3d(X,dC*Y-dS*Z,dS*Y+dC*Z);
}

// Set Rotate X
Vector3d &Vector3d::SetRotateX(double dA)
{
// Compute Cosine & Sine
double dC=cos(dA);
double dS=sin(dA);
// Compute Rotation
Set(X,dC*Y-dS*Z,dS*Y+dC*Z);
return *this;
}

// Rotate Y
Vector3d Vector3d::RotateY(double dA) const
{
// Compute Cosine & Sine
double dC=cos(dA);
double dS=sin(dA);
// Compute Rotation
return Vector3d(dC*X+dS*Z,Y,-dS*X+dC*Z);
}

// Set Rotate Y
Vector3d &Vector3d::SetRotateY(double dA)
{
// Compute Cosine & Sine
double dC=cos(dA);
double dS=sin(dA);
// Compute Rotation
Set(dC*X+dS*Z,Y,-dS*X+dC*Z);
return *this;
}
```

```
// Rotate Z
Vector3d Vector3d::RotateZ(double dA) const
{
// Compute Cosine & Sine
double dC=cos(dA);
double dS=sin(dA);
// Compute Rotation
return Vector3d(dC*X-dS*Y,dS*X+dC*Y,Z);
}

// Set Rotate Z
Vector3d &Vector3d::SetRotateZ(double dA)
{
// Compute Cosine & Sine
double dC=cos(dA);
double dS=sin(dA);
// Compute Rotation
Set(dC*X-dS*Y,dS*X+dC*Y,Z);
return *this;
}

//Rotate R
Vector3d Vector3d::RotateN(const double dAngle,const Vector3d vN) const
{
double dC=cos(dAngle);
double dS=sin(dAngle);

return *this*dC+vN*vN.Dot(*this)*(1-dC) + vN.Cross(*this)*dS;
}
/*Vector3d.h
Written by Mike Williams 10-29-02
re worked to use only double valuses as an attempt to explain wierd behaviour noted in notes today
*/
#ifndef _vector3d_h
#define _vector3d_h

// Vector3d Class
class Vector3d
{
public:
double X; // Components
double Y;
double Z;
double W; // Padding

private:
const static Vector3d ms_Null; // Null Instance

public:
// Constructors
Vector3d() : X(0.0),Y(0.0),Z(0.0),W(0.0) {}
Vector3d(double dX,double dY) : X(dX),Y(dY),Z(0.0),W(0.0) {}
Vector3d(double dX,double dY,double dZ) : X(dX),Y(dY),Z(dZ),W(0.0) {}
Vector3d(const Vector3d &v) : X(v.X),Y(v.Y),Z(v.Z),W(0.0) {}
Vector3d(const Vector3d &A,const Vector3d &B) : X(B.X-A.X),Y(B.Y-A.Y),Z(B.Z-A.Z),W(0.0) {}

// Null Functions
const static Vector3d &GetNull() {return ms_Null;}

// Set Functions
void SetZero() {X=Y=Z=W=0.0;}
void Set(double dX,double dY,double dZ) {X=dX; Y=dY; Z=dZ;}
void SetW(double dW) {W=dW;}
void Negate() {X=-X; Y=-Y; Z=-Z;}

// Operators
Vector3d &operator=(const Vector3d &v) {X=v.X; Y=v.Y; Z=v.Z; return *this;}
Vector3d &operator+=(const Vector3d &v) {X+=v.X; Y+=v.Y; Z+=v.Z; return *this;}
Vector3d &operator-=(const Vector3d &v) {X-=v.X; Y-=v.Y; Z-=v.Z; return *this;}
Vector3d &operator*=(const double d) {X*=d; Y*=d; Z*=d; return *this;}
Vector3d &operator/=(const double d) {return operator*=(1.0/d);}

Vector3d operator+(const Vector3d &v) const {return Vector3d(X+v.X,Y+v.Y,Z+v.Z);}
Vector3d operator-(const Vector3d &v) const {return Vector3d(X-v.X,Y-v.Y,Z-v.Z);}
Vector3d operator*(const double d) const {return Vector3d(X*d,Y*d,Z*d);}
double operator*(const Vector3d &v) const {return (X*v.X+Y*v.Y+Z*v.Z);}
Vector3d operator/(const double d) const {return operator*(1.0/d);}

int operator==(const Vector3d &v) const {return ((X==v.X && Y==v.Y && Z==v.Z)?1:0);}
int operator!=(const Vector3d &v) const {return ((X!=v.X || Y!=v.Y || Z!=v.Z)?1:0);}

// Products
double Dot(const Vector3d &v) const {return (X*v.X+Y*v.Y+Z*v.Z);}
```

```
Vector3d Cross(const Vector3d &v) const {return Vector3d(Y*v.Z-v.Y*Z,-X*v.Z+v.X*Z,X*v.Y-v.X*Y);}

// Angle Functions
double Cosine(const Vector3d &v) {return Dot(v)/(Size()*v.Size());}

// Size Functions
double Size() const;
double SizeSquared() const {return (X*X+Y*Y+Z*Z);}
Vector3d Normal() const {return operator/(Size());}
Vector3d &Normalize() {return operator/=(Size());}

// Project Functions
double ProjectedDistance(const Vector3d &v) const {return Dot(v)/v.Size();}
Vector3d Project(const Vector3d &v) const {double d=Dot(v)/v.SizeSquared(); return Vector3d(v.X*d,v.Y*d,v.Z*d);}
void SetProject(const Vector3d &v) {double d=Dot(v)/v.SizeSquared(); X=v.X*d; Y=v.Y*d; Z=v.Z*d;}
Vector3d ProjectNormal(const Vector3d &v) const {double d=Dot(v)/v.SizeSquared(); return Vector3d(X-v.X*d,Y-v.Y*d,Z-v.Z*d);}
void SetProjectNormal(const Vector3d &v) {double d=Dot(v)/v.SizeSquared(); X-=v.X*d; Y-=v.Y*d; Z-=v.Z*d;}

// Rotation Functions
Vector3d RotateX(const double dAngle) const;
Vector3d &SetRotateX(const double dAngle);
Vector3d RotateY(const double dAngle) const;
Vector3d &SetRotateY(const double dAngle);
Vector3d RotateZ(const double dAngle) const;
Vector3d &SetRotateZ(const double dAngle);
Vector3d RotateN(const double dAngle,const Vector3d vN)const;
};


#endif
//adaptFunctions.cpp Mike Williams 4/05
#include"adaptFunctions.h"
#include"constants.h"
#include"star6_0.h"

void copyStars(star *starData[],star *starTarget[])
{
for(int i=0;i<g_c_iStars;i++)
{
starTarget[i]->m_vPosition=starData[i]->m_vPosition;
starTarget[i]->m_vVelocity=starData[i]->m_vVelocity;
starTarget[i]->m_dMass=starData[i]->m_dMass;
}
}
double errorPosition(star *stars1[],star *starsh[])
{
double dAnswer;

dAnswer=0;
for(int i=0;i<g_c_iStars;i++)
{
dAnswer+=(starsh[i]->m_vPosition - stars1[i]->m_vPosition).Size();
}
return dAnswer;
}
//adaptFunctions.h Mike Williams 4/05
#ifndef _adapt_h
#define _adapt_h

#include"star6_0.h"
#include"constants.h"

void copyStars(star *starData[],star *starTarget[]);
double errorPosition(star *stars1[],star *starsh[]);

#endif
//binaryCount.cpp
//written by Mike Williams 02-25-05

#include"binaryCount.h"
#include"util.h"

int binaryCount(star *stars[], double Rs[g_c_iStars][g_c_iStars])
{
double dMu,dE,dRMin,dR,dRTemp;
Vector3d vV;
int iRight,iLeft,iBinary;


iBinary=0;
for(int i=0;i<g_c_iStars;i++)
{
for(int j=i+1;j<g_c_iStars;j++)
```

```
{
dMu=stars[i]->m_dMass*stars[j]->m_dMass/(stars[i]->m_dMass+stars[j]->m_dMass);
vV=stars[i]->m_vVelocity - stars[j]->m_vVelocity;
dR=sqrt(Rs[i][j]);
dE=0.5*dMu*vV.SizeSquared() - stars[i]->m_dMass*stars[j]->m_dMass/dR;
if(dE<0)//could be a binary check for local stars
{
dRMin=100*dR;
for(int k=0;k<g_c_iStars;k++)
{
if((k!=i)&&(k!=j))
{
iRight=max(i,k);
iLeft=min(i,k);
dRTemp=sqrt(Rs[iLeft][iRight]);
if(dRTemp<dRMin)
dRMin=dRTemp;

iRight=max(j,k);
iLeft=min(j,k);
dRTemp=sqrt(Rs[iLeft][iRight]);
if(dRTemp<dRMin)
dRMin=dRTemp;
}
}

if(dRMin>g_c_iRMultipy*dR)
{
iBinary++;
}
}//end energy if
}//end for j
}//end for i

return iBinary;
}//end function

int binaryCount(star *stars[])
{
double dMu,dE,dRMin,dR,dRTemp;
Vector3d vV,vR;
int iRight,iLeft,iBinary;


iBinary=0;
for(int i=0;i<g_c_iStars;i++)
{
for(int j=i+1;j<g_c_iStars;j++)
{
dMu=stars[i]->m_dMass*stars[j]->m_dMass/(stars[i]->m_dMass+stars[j]->m_dMass);
vV=stars[i]->m_vVelocity - stars[j]->m_vVelocity;
vR=stars[i]->m_vPosition-stars[j]->m_vPosition;
dR=vR.Size();
dE=0.5*dMu*vV.SizeSquared() - stars[i]->m_dMass*stars[j]->m_dMass/dR;
if(dE<0)//could be a binary check for local stars
{
dRMin=100*dR;
for(int k=0;k<g_c_iStars;k++)
{
if((k!=i)&&(k!=j))
{
dRTemp=(stars[i]->m_vPosition-stars[k]->m_vPosition).Size();
if(dRTemp<dRMin)
dRMin=dRTemp;

dRTemp=(stars[j]->m_vPosition-stars[k]->m_vPosition).Size();
if(dRTemp<dRMin)
dRMin=dRTemp;
}
}

if(dRMin>g_c_iRMultipy*dR)
{
iBinary++;
}
}//end energy if
}//end for j
}//end for i

return iBinary;
}//end function

int binaryCountIndex(star *stars[],int iFirst[],int iSecond[], double dRBinarys[])
```

```
{
double dMu,dE,dRMin,dR,dRTemp;
Vector3d vV,vR;
int iRight,iLeft,iBinary;


iBinary=0;
for(int i=0;i<g_c_iStars;i++)
{
for(int j=i+1;j<g_c_iStars;j++)
{
dMu=stars[i]->m_dMass*stars[j]->m_dMass/(stars[i]->m_dMass+stars[j]->m_dMass);
vV=stars[i]->m_vVelocity - stars[j]->m_vVelocity;
vR=stars[i]->m_vPosition-stars[j]->m_vPosition;
dR=vR.Size();
dE=0.5*dMu*vV.SizeSquared() - stars[i]->m_dMass*stars[j]->m_dMass/dR;
if(dE<0)//could be a binary check for local stars
{
dRMin=100*dR;
for(int k=0;k<g_c_iStars;k++)
{
if((k!=i)&&(k!=j))
{
dRTemp=(stars[i]->m_vPosition-stars[k]->m_vPosition).Size();
if(dRTemp<dRMin)
dRMin=dRTemp;

dRTemp=(stars[j]->m_vPosition-stars[k]->m_vPosition).Size();
if(dRTemp<dRMin)
dRMin=dRTemp;
}
}

if(dRMin>g_c_iRMultipy*dR)
{
iFirst[iBinary]=i;
iSecond[iBinary]=j;
dRBinarys[iBinary]=dR;
iBinary++;
}
}//end energy if
}//end for j
}//end for i

return iBinary;
}//end function
//binaryCount.h
//written by Mike Williams 02-25-05

#ifndef binaryCount_h
#define binaryCount_h

#include"constants.h"
#include"star6_0.h"

int binaryCount(star *stars[], double Rs[g_c_iStars][g_c_iStars]);
int binaryCount(star *stars[]);
int binaryCountIndex(star *stars[],int iFirst[],int iSecond[], double dRBinarys[]);
#endif
//binarytracker.cpp
//written by Mike Williams 05-18-05

#include"binarytracker.h"

binaryTracker::binaryTracker()
{
m_dAveR=0;
m_dAveT=0;
m_iBinaries=0;
m_dRSum=0;
m_iRMeasures=0;
m_dTSum=0;
m_iTMeasures=0;
m_dPMSum=0;
m_dSMSum=0;
m_iMMeasures=0;
m_dAvePM=0;
m_dAveSM=0;
}
binaryTracker::~binaryTracker()
{
//empty
}
```

```
void binaryTracker::addBinary(const int &i,const int &j,const double &dR,const double &dM1,const double &dM2,const double &dT)
{
bool bFound;

bFound=false;


//check if this is a new binary
for(int iPair=0;iPair<m_iBinaries;iPair++)
{
if(((m_iIndicies[iPair][0]==i)&&(m_iIndicies[iPair][1]==j))||((m_iIndicies[iPair][0]==j)&&(m_iIndicies[iPair][1]==i)))
{
m_bCurrent[iPair]=bFound=true;
break;
}
}
if(!bFound)
{
m_iIndicies[m_iBinaries][0]=i;
m_iIndicies[m_iBinaries][1]=j;
m_dTimes[m_iBinaries]=dT;
m_bCurrent[m_iBinaries]=true;
m_iBinaries++;
//update average r stuff
m_dRSum+=dR;
m_iRMeasures++;
m_dAveR=m_dRSum/m_iRMeasures;

//add masses
if(dM1>dM2)
{
m_dPMSum+=dM1;
m_dSMSum+=dM2;
}
else
{
m_dPMSum+=dM2;
m_dSMSum+=dM1;
}

m_iMMeasures++;
m_dAvePM=m_dPMSum/m_iMMeasures;
m_dAveSM=m_dSMSum/m_iMMeasures;
}
}

void binaryTracker::checkTerm(const double &dT)
{
bool bSwitch;
bSwitch=true;
//make sure the last pair is current
if(m_iBinaries>0)
{
while(bSwitch)
{
if(!m_bCurrent[m_iBinaries-1])
{
m_dTSum+=(dT-m_dTimes[m_iBinaries-1]);
m_iTMeasures++;
m_dAveT=m_dTSum/m_iTMeasures;
m_iBinaries--;
}
else
{
bSwitch=false;
}
}
}
//now that I know the last pair is current i can check the rest
if(m_iBinaries>1)
{
for(int iPair=0;iPair<m_iBinaries;iPair++)
{
if(!m_bCurrent[iPair])
{
m_dTSum+=(dT-m_dTimes[iPair]);
m_iTMeasures++;
m_dAveT=m_dTSum/m_iTMeasures;
//put the last one here
m_iIndicies[iPair][0]=m_iIndicies[m_iBinaries-1][0];
m_iIndicies[iPair][1]=m_iIndicies[m_iBinaries-1][1];
m_dTimes[iPair]=m_dTimes[m_iBinaries-1];
```

```
m_iBinaries--;
}
//reset current for next time step
m_bCurrent[iPair]=false;
}
}
else
{
m_bCurrent[0]=false;
}
}
//binarytracker.h
//written by Mike Williams 05-18-05

#ifndef binarytrack_h
#define binarytrack_h

const int g_c_iMaxBin=10;

class binaryTracker
{
public:
//constructors, destructor
binaryTracker();
~binaryTracker();

void addBinary(const int &i,const int &j,const double &dR,const double &dM1,const double &dM2,const double &dT);
void checkTerm(const double &dT);

//public variables
double m_dAveR;
double m_dAveT;
double m_dAvePM;//primary mass average
double m_dAveSM;//secondary mass average

private:
//for binary tracking
int m_iBinaries;
int m_iIndicies[g_c_iMaxBin][2];
double m_dTimes[g_c_iMaxBin];
bool m_bCurrent[g_c_iMaxBin];

//for average r
double m_dRSum;
int m_iRMeasures;

//for average life time T
double m_dTSum;
int m_iTMeasures;

//mass average
double m_dPMSum,m_dSMSum;
int m_iMMeasures;


};

#endif
/* compare2_1,cpp
written by Mike Williams 03-15-05
*/

#include"star6_0.h"
#include"constants.h"
#include"compare2_1.h"
#include"util.h"

bool compareSoftSimple(star *star1, star *star2, Vector3d &vF,double &r,const double &c_dRMin)
{
Vector3d vR;//radious between stars
double dR;

vR=star2->m_vPosition-star1->m_vPosition;
r=(vR).SizeSquared();
dR=sqrt(r);
vF=vR.Normal()*star1->m_dMass*star2->m_dMass/((dR+c_dRMin)*(dR+c_dRMin));
return true;
}



bool compare3Simple(star *star1, star *star2, Vector3d &vF,const double &rMin)
```

```
{
Vector3d vR;//radious between stars

vR=star2->m_vPosition-star1->m_vPosition;
if(vR.SizeSquared()>rMin)//fix this
{
//calc force and acceleration
//vF=vR.Normal()*g_c_dSG*star1->m_dMass*star2->m_dMass/vR.SizeSquared();
vF=vR.Normal()*star1->m_dMass*star2->m_dMass/vR.SizeSquared();
return true;
}
return false;
}

bool compare3Simple2(star *star1, star *star2, Vector3d &vF,double &r, const double &rMin)
{
Vector3d vR;//radious between stars

vR=star2->m_vPosition-star1->m_vPosition;
r=vR.SizeSquared();
if(r>rMin)//fix this
{
//calc force and acceleration
//vF=vR.Normal()*g_c_dSG*star1->m_dMass*star2->m_dMass/vR.SizeSquared();
vF=vR.Normal()*star1->m_dMass*star2->m_dMass/vR.SizeSquared();
return true;
}
return false;
}
//compare2_1.h Mike Williams 3/05
#ifndef _compare_h
#define _compare_h

#include"star6_0.h"
#include"constants.h"


bool compare3Simple(star *star1, star *star2, Vector3d &vF,const double &rMin);
bool compare3Simple2(star *star1, star *star2, Vector3d &vF,double &r, const double &rMin);
bool compareSoftSimple(star *star1, star *star2, Vector3d &vF,double &r,const double &c_dRMin);
#endif
/*constants.h
programed by Mike Williams 10-02-02
Contains constants that are needed in multiple files of the simulation.
*/
#ifndef constants_h
#define constants_h
#include<math.h>


//physicly meaningfull constants
const float g_c_fG = (float)6.67e-11;//Gravitational constant
const double g_c_dG = 6.67e-11;//Nm^2/kg^2 Gravitational consant
const double g_c_dMSun=1.99E30;//kg
const double g_c_dRSun=6.955E8;//m

//simulation scale units, in standard units.
const double g_c_dR0=(9.46E15)*1.5;//m is my default measure of lenght actualy it is about a light year


const double g_c_dL=g_c_dR0;
const double g_c_dM=g_c_dMSun;
const double g_c_dT=sqrt(g_c_dL*g_c_dL*g_c_dL/(g_c_dG*g_c_dM));

//const double g_c_dT=31536000;//s = 1 year

//Units to help me think
//devide a number by one of these to get that number of (name) in scaled units
//or multiply a scaled quantity by these numbers to get them in days or years or whatever
const double g_c_dDayConvert=g_c_dT/(3600*24);
const double g_c_dYearConvert=g_c_dT/(3600*24*365.25);


//scaled gravitational constant
const double g_c_dSG=g_c_dG*g_c_dM*g_c_dT*g_c_dT/(g_c_dL*g_c_dL*g_c_dL);//G*M*T^2/L^3 should be Force*lenght^2/Mas^2
const double g_c_dEnergyConverter=g_c_dT*g_c_dT/(g_c_dL*g_c_dL*g_c_dM);//T^2/(L^2*M) Jules = Energy/EnergyConverter

const int g_c_iStars = 100;
const double g_c_dPairs = (g_c_iStars*g_c_iStars-g_c_iStars)/2.0;
const double g_c_dDtMin=1E-7;
const double g_c_dDtMax=.001;
const double g_c_dClusterTimes=3;//1.0;//0.1;
const double g_c_dRSystem=1000;
```

```
//const double g_c_dRMinM=9.46E10;//m
const double g_c_dRMinM=g_c_dRSun*10;
const double g_c_dRMin=g_c_dRMinM/g_c_dL;
const double g_c_dRMin2=g_c_dRMin*g_c_dRMin;
const int g_c_iRMultipy=2;
const int g_c_iSmallTimeSteps=10;
const int g_c_iDtDevider=1000;
const double g_c_dDtSmall=.1;//how much smaller must he the local dt before I care
const int g_c_iNTimeStepsToBeBinary=0;
const double g_c_dTSystem =0;//= g_c_iNTimeStepsToBeBinary*g_c_dDt;//minimum time to be considered a binary in scaled units
const double g_c_dAdaptPower=0.2;
const double g_c_dError0m=9.46E2;//m
const double g_c_dError0=g_c_dError0m/g_c_dL;//ensure that my error is unitless

//other constants
const double g_c_dPi=3.14159265;
const double g_c_dSK=1;//Scailed bolzman's constant. I don't think it matters so I am setting it to one

#endif
//distroIO.cpp
//Mike Williams 02-05-05

#include"distroIO1_2.h"
#include"star6_0.h"
#include"constants.h"
#include<fstream>
#include<iostream>

void saveDistro(star *stars[],std::ofstream &disOut,double a_dTime,double a_dTimeStep)
{
double dMt,dXt,dYt,dZt,dVxt,dVyt,dVzt;
disOut<<g_c_iStars<<std::endl;
//save star data
for(int i = 0;i<g_c_iStars;i++)
{
dMt=stars[i]->m_dMass;
dXt=stars[i]->m_vPosition.X;
dYt=stars[i]->m_vPosition.Y;
dZt=stars[i]->m_vPosition.Z;

dVxt=stars[i]->m_vVelocity.X;
dVyt=stars[i]->m_vVelocity.Y;
dVzt=stars[i]->m_vVelocity.Z;
disOut<<dMt<<" "<<dXt<<" "<<dYt<<" "<<dZt<<" "<<dVxt<<" "<<dVyt<<" "<<dVzt<<std::endl;
}
}

void loadDistro(star *stars[],std::ifstream &disIn)
{
double dMt,dXt,dYt,dZt,dVxt,dVyt,dVzt;
int iNStars;

disIn>>iNStars;
if(iNStars!=g_c_iStars)
{
std::cout<<"Error stars in file "<<iNStars<<" g_c_iStars "<<g_c_iStars<<std::endl;
exit(1);
}
//load star data
for(int i = 0;i<g_c_iStars;i++)
{
disIn>>dMt>>dXt>>dYt>>dZt>>dVxt>>dVyt>>dVzt;
stars[i]->initialize(Vector3d(dXt,dYt,dZt),Vector3d(dVxt,dVyt,dVzt),dMt);
}
}

void saveDistroRealUnits(star *stars[],std::ofstream &disOut,const double &c_dL, const double &c_dT)
{
double dMt,dXt,dYt,dZt,dVxt,dVyt,dVzt;

disOut<<g_c_iStars<<std::endl;
//save star data
for(int i = 0;i<g_c_iStars;i++)
{
dMt=stars[i]->m_dMass*g_c_dM;
dXt=stars[i]->m_vPosition.X*c_dL;
dYt=stars[i]->m_vPosition.Y*c_dL;
dZt=stars[i]->m_vPosition.Z*c_dL;

dVxt=stars[i]->m_vVelocity.X*c_dL/c_dT;
dVyt=stars[i]->m_vVelocity.Y*c_dL/c_dT;
dVzt=stars[i]->m_vVelocity.Z*c_dL/c_dT;
disOut<<dMt<<" "<<dXt<<" "<<dYt<<" "<<dZt<<" "<<dVxt<<" "<<dVyt<<" "<<dVzt<<std::endl;
```

```
}
}
void loadDistroRealUnits(star *stars[],std::ifstream &disIn,const double &c_dL, const double &c_dT)
{
double dMt,dXt,dYt,dZt,dVxt,dVyt,dVzt;
int iNStars;

disIn>>iNStars;
if(iNStars!=g_c_iStars)
{
std::cout<<"Error stars in file "<<iNStars<<" g_c_iStars "<<g_c_iStars<<std::endl;
exit(1);
}
//load star data
for(int i = 0;i<g_c_iStars;i++)
{
disIn>>dMt>>dXt>>dYt>>dZt>>dVxt>>dVyt>>dVzt;
dMt/=g_c_dM;
dXt/=c_dL;
dYt/=c_dL;
dZt/=c_dL;
dVxt*=c_dT/c_dL;
dVyt*=c_dT/c_dL;
dVzt*=c_dT/c_dL;
stars[i]->initialize(Vector3d(dXt,dYt,dZt),Vector3d(dVxt,dVyt,dVzt),dMt);
}
}

void saveConst(std::ofstream &constOut,char sAdapt[],const double &c_dR0, const double &c_dL)
{
//save every possible parameter

//simulation scale units, in standard units.
constOut<<"Sim Constants"<<std::endl;
constOut<<"dR0 =\t\t\t"<<c_dR0<<std::endl;

constOut<<"dL =\t\t\t"<<c_dL<<std::endl;
constOut<<"g_c_dM =\t\t\t"<<g_c_dM<<std::endl;
constOut<<"g_c_iStars =\t\t\t"<<g_c_iStars<<std::endl;
constOut<<"g_c_dDtMin =\t\t\t"<<g_c_dDtMin<<std::endl;
constOut<<"g_c_dDtMax=\t\t\t"<<g_c_dDtMax<<std::endl;
constOut<<"g_c_dClusterTimes =\t\t"<<g_c_dClusterTimes<<std::endl;
constOut<<"g_c_dRSystem =\t\t\t"<<g_c_dRSystem<<std::endl;
constOut<<"g_c_dRMinM =\t\t\t"<<g_c_dRMinM<<std::endl;
constOut<<"g_c_iRMultipy =\t\t\t"<<g_c_iRMultipy<<std::endl;
constOut<<"g_c_iSmallTimeSteps =\t\t"<<g_c_iSmallTimeSteps<<std::endl;
constOut<<"g_c_iDtDevider =\t\t"<<g_c_iDtDevider<<std::endl;
constOut<<"g_c_dDtSmall =\t\t\t"<<g_c_dDtSmall<<std::endl;
constOut<<"g_c_iNTimeStepsToBeBinary =\t"<<g_c_iNTimeStepsToBeBinary<<std::endl;
constOut<<"g_c_dTSystem =\t\t\t"<<g_c_dTSystem<<std::endl;
constOut<<"g_c_dAdaptPower =\t\t"<<g_c_dAdaptPower<<std::endl;
constOut<<"g_c_dError0m =\t\t\t"<<g_c_dError0m<<std::endl;

constOut<<"Adapt description\t\t"<<sAdapt<<std::endl;
}
//distroIO1_2.h
//by Mike Williams 05-16-05
//load and save star distrobutions
//simplere than 1_0 which tried to save units and such
//uses command line arguments

#ifndef distroIO_h
#define distroIO_h

#include"star6_0.h"
#include<fstream>

void saveDistro(star *stars[],std::ofstream &disOut, double a_dTime, double a_dTimeStep);
void loadDistro(star *stars[],std::ifstream &disIn);

void saveDistroRealUnits(star *stars[],std::ofstream &disOut,const double &c_dL, const double &c_dT);
void loadDistroRealUnits(star *stars[],std::ifstream &disIn,const double &c_dL, const double &c_dT);

void saveConst(std::ofstream &constOut,char sAdapt[],const double &c_dR0, const double &c_dL);

#endif
//distroStats.cpp
//by Mike Williams 04-25-05

#include"distroStats.h"
#include"star6_0.h"
#include"constants.h"
```

```
void RVDRrms(star *stars[],double &dRrms,double &dVrms,double &dDRrms)
{
dRrms=0;
dVrms=0;
dDRrms=0;

for(int i=0;i<g_c_iStars;i++)
{
for(int j=i+1;j<g_c_iStars;j++)
{
dDRrms+=(stars[i]->m_vPosition-stars[j]->m_vPosition).SizeSquared();
}
dRrms+=stars[i]->m_vPosition.SizeSquared();
dVrms+=stars[i]->m_vVelocity.SizeSquared();
}

dRrms=sqrt(dRrms/(double)g_c_iStars);
dVrms=sqrt(dVrms/(double)g_c_iStars);
dDRrms=sqrt(dDRrms/g_c_dPairs);
}
void RVDRrmsL(star *stars[],double &dRrms,double &dVrms,double &dDRrms,Vector3d &vL)
{
dRrms=0;
dVrms=0;
dDRrms=0;
vL.SetZero();

for(int i=0;i<g_c_iStars;i++)
{
for(int j=i+1;j<g_c_iStars;j++)
{
dDRrms+=(stars[i]->m_vPosition-stars[j]->m_vPosition).SizeSquared();
}
dRrms+=stars[i]->m_vPosition.SizeSquared();
dVrms+=stars[i]->m_vVelocity.SizeSquared();
vL+=stars[i]->m_vPosition.Cross(stars[i]->m_vVelocity)*stars[i]->m_dMass;
}

dRrms=sqrt(dRrms/(double)g_c_iStars);
dVrms=sqrt(dVrms/(double)g_c_iStars);
dDRrms=sqrt(dDRrms/g_c_dPairs);
}
double AveMass(star *stars[])
{
double dM;
dM=0;
for(int i=0;i<g_c_iStars;i++)
{
dM+=stars[i]->m_dMass;
}
return dM/g_c_iStars;
}
//distroStats.h
//by Mike Williams 04-25-05
//Calculate thins ling Rrms Vrms DelRrms

#ifndef distroStats_h
#define distroStats_h

#include"star6_0.h"
#include"Vector3d.h"

void RVDRrms(star *stars[],double &dRrms,double &dVrms,double &dDRrms);
void RVDRrmsL(star *stars[],double &dRrms,double &dVrms,double &dDRrms,Vector3d &vL);

double AveMass(star *stars[]);

#endif
//distrobution.h Mike Williams 2/05
#include"distrobution.h"
#include"constants.h"
#include"star6_0.h"
#include"energy1_0.h"
#include"rand.h"
#include<math.h>
#include<iostream>
#include<fstream>
#include"util.h"

//spatial functions
double ro(double a_dR,double a_dR0)
{
return g_c_iStars/a_dR0*exp(-a_dR/a_dR0);
```

```
}

double rMid(int a_iI,double a_dR0)
{
return (a_iI*rOne(a_dR0)/g_c_iNBins - (a_iI-1)*rOne(a_dR0)/g_c_iNBins)/2 + (a_iI-1)*rOne(a_dR0)/g_c_iNBins;
}

int nStarsInBinR(double a_dR, double a_dR0)
{
return round(rOne(a_dR0)/g_c_iNBins*ro(a_dR,a_dR0));
}

double rOne(double a_dR0)
{
return a_dR0*-log(1.0/g_c_iStars);
}

void uniformMassDistribute(star *stars[])
{
for(int i=0;i<g_c_iStars;i++)
{
stars[i]->m_dMass=g_c_dMSun/g_c_dM;
}
}

double boltzmann(double a_dE,double a_dT)
{
return g_c_iStars/g_c_dSK/a_dT*exp(-a_dE/g_c_dSK/a_dT);
}

double boltzmannV2(double a_dE,double a_dT)
{
return g_c_iStars/(2*g_c_dSK*g_c_dSK*g_c_dSK*a_dT*a_dT*a_dT)*a_dE*a_dE*exp(-a_dE/g_c_dSK/a_dT);
}

double eMid(int a_iI,double a_dT)
{
return (a_iI*eOne(a_dT)/g_c_iNBins - (a_iI-1)*eOne(a_dT)/g_c_iNBins)/2 + (a_iI-1)*eOne(a_dT)/g_c_iNBins;
}

int nStarsInBinE(double a_dE, double a_dT)
{
return round(eOne(a_dT)/g_c_iNBins*boltzmann(a_dE,a_dT));
}
int nStarsInBinEV2(double a_dE,double a_dT,double a_dE1)
{
double dDE=a_dE1/g_c_iNBins;

return round(dDE*boltzmannV2(a_dE,a_dT));
}

double eOne(double a_dT)
{
return -g_c_dSK*a_dT*log(1.0/g_c_iStars);
}

double eMid(int a_iI,double a_dT,double a_dE1)
{
double dDE=a_dE1/g_c_iNBins;
return dDE/2 + (a_iI-1)*dDE;
}

double tailV2(double a_dE,double a_dT)
{
double dK2=g_c_dSK*g_c_dSK;
double dT2=a_dT*a_dT;
return exp(-1*a_dE/(g_c_dSK*a_dT))*g_c_iStars*(a_dE*a_dE+2*a_dE*a_dT*g_c_dSK+2*dK2*dT2)/(2*dK2*dT2);
}

double tailV2p(double a_dE, double a_dT)
{
double dK2=g_c_dSK*g_c_dSK;
double dT2=a_dT*a_dT;

double dTerm1,dTerm2;

dTerm1=exp(-1*a_dE/(g_c_dSK*a_dT))*g_c_iStars*(2*a_dE+2*g_c_dSK*a_dT)/(2*dK2*dT2);

dTerm2=exp(-1*a_dE/(g_c_dSK*a_dT))*g_c_iStars*(a_dE*a_dE+2*a_dE*a_dT*g_c_dSK+2*dK2*dT2)/(2*dK2*dT2*g_c_dSK*a_dT);

return dTerm1-dTerm2;

}
```

```
double eOneV2(double a_dT,double a_dE0)
{
const double c_dErrorTollerence=1E-10;//error convergence tollerance
const int c_iNIterations = 100;//maximum iterations to try befroe giving up
double dError=1;
double dE,dNE;

dE=a_dE0;

for(int i=0;(i<c_iNIterations&&dError>c_dErrorTollerence);i++)
{

dNE=tailV2(dE,a_dT);
dError = abs(dNE-1);
dE=dE - (dNE-1)/tailV2p(dE,a_dT);
}
return dE;
}
void spatialyDistributeMid(star *stars[],const double a_dR0)
{
int iNInit=0;
int iStarsInBin,iIndex;
double dR;
bool bInit[g_c_iStars];
//std::ofstream spOut;

// spOut.open("spaceOut.csv");

//spOut<<"i,x,y,z"<<std::endl;

for(int i=0;i<g_c_iStars;i++)
{
bInit[i]=false;
}

for(int i=1;i<=g_c_iNBins;i++)
{
dR=rMid(i,a_dR0);
iStarsInBin=nStarsInBinR(dR,a_dR0);
for(int j=0;j<iStarsInBin;j++)
{
iIndex=Random(g_c_iStars-1);
while(bInit[iIndex])
{
iIndex=Random(g_c_iStars-1);
}
bInit[iIndex]=true;
stars[iIndex]->m_vPosition=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*dR;
iNInit++;
}
}
if(iNInit<g_c_iStars)
{
//put one at rOne
//hopefully there arn't to many so rather than randomize i will be systematic
//so the first uninitiated star I find I initalize to be at rOne
//spOut<<",,,,ROne"<<std::endl;
for(int i=0;i<g_c_iStars;i++)
{
if(!bInit[i])
{
stars[i]->m_vPosition=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*rOne(a_dR0);
bInit[i]=true;
iNInit++;
//kill the for
break;
}
}
//it could happen that there are sill a couple who didn't get placed...
//so place any unpaced stars completely randomly
//spOut<<",,,,Extras"<<std::endl;
for(int i=0;i<g_c_iStars;i++)
{
if(!bInit[i])
{
stars[i]->m_vPosition=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*fRandom(rOne(a_dR0));
bInit[i]=true;
iNInit++;
}
}
}
// spOut.close();
```

```
}
void energyDistribute(star *stars[],double &a_dT)
{
int iNInit=0;
int iStarsInBin,iIndex;
double dE,dPE,dKE,dV;
bool bInit[g_c_iStars];

dPE=energyP(stars);
dKE=-dPE/2.0;
a_dT=dKE/g_c_iStars/g_c_dSK;

for(int i=0;i<g_c_iStars;i++)
{
bInit[i]=false;
}

for(int i=1;i<=g_c_iNBins;i++)
{
dE=eMid(i,a_dT);
iStarsInBin=nStarsInBinE(dE,a_dT);
for(int j=0;j<iStarsInBin;j++)
{
iIndex=Random(g_c_iStars-1);
while(bInit[iIndex])
{
iIndex=Random(g_c_iStars-1);
}
bInit[iIndex]=true;
dV=sqrt(2*dE/stars[iIndex]->m_dMass);
stars[iIndex]->m_vVelocity=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*dV;
iNInit++;
}
}
if(iNInit<g_c_iStars)
{
//put one at rOne
//hopefully there arn't to many so rather than randomize i will be systematic
//so the first uninitiated star I find I initalize to be at rOne
for(int i=0;i<g_c_iStars;i++)
{
if(!bInit[i])
{
dE=eOne(a_dT);
dV=sqrt(2*dE/stars[i]->m_dMass);
stars[i]->m_vVelocity=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*dV;
bInit[i]=true;
iNInit++;
//kill the for
break;
}
}
//it could happen that there are sill a couple who didn't get placed...
//so place any unpaced stars completely randomly
for(int i=0;i<g_c_iStars;i++)
{
if(!bInit[i])
{
dE=fRandom(eOne(a_dT));
dV=sqrt(2*dE/stars[i]->m_dMass);
stars[i]->m_vVelocity=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*dV;
bInit[i]=true;
iNInit++;
}
}
}
}

void energyDistributeV2(star *stars[],double &a_dT,double &a_dE1)
{
int iNInit=0;
int iStarsInBin,iIndex;
double dE,dPE,dKE,dV,dE1;
bool bInit[g_c_iStars];

dPE=energyP(stars);
dKE=-dPE/2.0;
a_dT=dKE/(g_c_iStars*g_c_dSK*3);

dE1=eOneV2(a_dT,dKE/g_c_iStars);
a_dE1=dE1;

for(int i=0;i<g_c_iStars;i++)
```

```
{
bInit[i]=false;
}

for(int i=1;i<=g_c_iNBins;i++)
{
dE=eMid(i,a_dT,dE1);
iStarsInBin=nStarsInBinEV2(dE,a_dT,dE1);
for(int j=0;j<iStarsInBin;j++)
{
iIndex=Random(g_c_iStars-1);
while(bInit[iIndex])
{
iIndex=Random(g_c_iStars-1);
}
bInit[iIndex]=true;
dV=sqrt(2*dE/stars[iIndex]->m_dMass);
stars[iIndex]->m_vVelocity=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*dV;
iNInit++;
}
}
if(iNInit<g_c_iStars)
{
//put one at eOne
//hopefully there arn't to many so rather than randomize i will be systematic
//so the first uninitiated star I find I initalize to be at eOne
for(int i=0;i<g_c_iStars;i++)
{
if(!bInit[i])
{
dE=dE1;
dV=sqrt(2*dE/stars[i]->m_dMass);
stars[i]->m_vVelocity=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*dV;
bInit[i]=true;
iNInit++;
//kill the for
break;
}
}
//it could happen that there are sill a couple who didn't get placed...
//so place any unpaced stars completely randomly
for(int i=0;i<g_c_iStars;i++)
{
if(!bInit[i])
{
dE=fRandom(dE1);
dV=sqrt(2*dE/stars[i]->m_dMass);
stars[i]->m_vVelocity=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*dV;
bInit[i]=true;
iNInit++;
}
}
}

}

void energyDistributeV2(star *stars[])
{
int iNInit=0;
int iStarsInBin,iIndex;
double dE,dPE,dKE,dV,dE1,dT;
bool bInit[g_c_iStars];

dPE=energyP(stars);
dKE=-dPE/2.0;
dT=dKE/(g_c_iStars*g_c_dSK*3);

dE1=eOneV2(dT,dKE/g_c_iStars);

for(int i=0;i<g_c_iStars;i++)
{
bInit[i]=false;
}

for(int i=1;i<=g_c_iNBins;i++)
{
dE=eMid(i,dT,dE1);
iStarsInBin=nStarsInBinEV2(dE,dT,dE1);
for(int j=0;j<iStarsInBin;j++)
{
iIndex=Random(g_c_iStars-1);
while(bInit[iIndex])
{
```

```
iIndex=Random(g_c_iStars-1);
}
bInit[iIndex]=true;
dV=sqrt(2*dE/stars[iIndex]->m_dMass);
stars[iIndex]->m_vVelocity=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*dV;
iNInit++;
}
}
if(iNInit<g_c_iStars)
{
//put one at rOne
//hopefully there arn't to many so rather than randomize i will be systematic
//so the first uninitiated star I find I initalize to be at rOne
for(int i=0;i<g_c_iStars;i++)
{
if(!bInit[i])
{
dE=dE1;
dV=sqrt(2*dE/stars[i]->m_dMass);
stars[i]->m_vVelocity=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*dV;
bInit[i]=true;
iNInit++;
//kill the for
break;
}
}
//it could happen that there are sill a couple who didn't get placed...
//so place any unpaced stars completely randomly
for(int i=0;i<g_c_iStars;i++)
{
if(!bInit[i])
{
dE=fRandom(dE1);
dV=sqrt(2*dE/stars[i]->m_dMass);
stars[i]->m_vVelocity=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*dV;
bInit[i]=true;
iNInit++;
}
}
}

}

void energyDistribute(star *stars[])
{
int iNInit=0;
int iStarsInBin,iIndex;
double dE,dPE,dKE,dV,dT;
bool bInit[g_c_iStars];
//std::ofstream spOut;

// spOut.open("velocityOut.csv");

//spOut<<"i,Vx,Vy,Vz"<<std::endl;

dPE=energyP(stars);
dKE=-dPE/2.0;
dT=dKE/g_c_iStars/g_c_dSK;

for(int i=0;i<g_c_iStars;i++)
{
bInit[i]=false;
}

for(int i=1;i<=g_c_iNBins;i++)
{
dE=eMid(i,dT);
iStarsInBin=nStarsInBinE(dE,dT);
for(int j=0;j<iStarsInBin;j++)
{
iIndex=Random(g_c_iStars-1);
while(bInit[iIndex])
{
iIndex=Random(g_c_iStars-1);
}
bInit[iIndex]=true;
dV=sqrt(2*dE/stars[iIndex]->m_dMass);
stars[iIndex]->m_vVelocity=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*dV;
iNInit++;
}
}
if(iNInit<g_c_iStars)
{
```

```
//spOut<<",,,,eOne"<<std::endl;
//put one at rOne
//hopefully there arn't to many so rather than randomize i will be systematic
//so the first uninitiated star I find I initalize to be at eOne
for(int i=0;i<g_c_iStars;i++)
{
if(!bInit[i])
{
dE=eOne(dT);
dV=sqrt(2*dE/stars[i]->m_dMass);
stars[i]->m_vVelocity=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*dV;
bInit[i]=true;
iNInit++;
//kill the for
break;
}
}
//spOut<<",,,,Extras"<<std::endl;
//it could happen that there are sill a couple who didn't get placed...
//so place any unpaced stars completely randomly
for(int i=0;i<g_c_iStars;i++)
{
if(!bInit[i])
{
dE=dRandom(eOne(dT));
dV=sqrt(2*dE/stars[i]->m_dMass);
stars[i]->m_vVelocity=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*dV;
bInit[i]=true;
iNInit++;
}
}
}

}

//mass realted functions

double massPower(double a_dM,double a_dMMin, double a_dMMax)
{
return 0.35*pow(a_dM,-1.35)*g_c_iStars/(1.0/pow(a_dMMin,0.35)-1.0/pow(a_dMMax,0.35));
}
double mMid(int a_iI,double a_dMMin, double a_dMMax)
{
double dDM=(a_dMMax-a_dMMin)/g_c_iNBins;
return a_dMMin + dDM/2.0 + (a_iI-1)*dDM;
}
int nStarsInBinM(double a_dM,double a_dMMin, double a_dMMax)
{
double dDM=(a_dMMax-a_dMMin)/g_c_iNBins;
//return round(dDM*massPower(a_dM,a_dMMin,a_dMMax));
double dA=a_dM-dDM/2.0;//integrating between a and b
double dB=a_dM+dDM/2.0;

double dAnswer;

dAnswer=round((1.0/pow(dA,0.35)-1.0/pow(dB,0.35))*g_c_iStars/(1.0/pow(a_dMMin,0.35)-1.0/pow(a_dMMax,0.35)));

return dAnswer;

}

void powerMassDistribute(star *stars[],double a_dMMin, double a_dMMax)
{
int iNInit=0;
int iStarsInBin,iIndex;
double dM;
bool bInit[g_c_iStars];
double dDM=(a_dMMax-a_dMMin)/g_c_iNBins;
int iTotalStars=0;

//test massPower
for(int i=1;i<=(g_c_iNBins);i++)
{
dM=mMid(i,a_dMMin,a_dMMax);
iTotalStars+=nStarsInBinM(dM,a_dMMin,a_dMMax);
}
if(iTotalStars>g_c_iStars)
{
std::cout<<"Error to course a mass distrobution"<<std::endl;
std::cout<<"Total stars initialzed "<<iTotalStars<<" Total stars "<<g_c_iStars<<std::endl;
}

for(int i=0;i<g_c_iStars;i++)
```

```
{
bInit[i]=false;
}

for(int i=1;i<=(g_c_iNBins);i++)
{
dM=mMid(i,a_dMMin,a_dMMax);
iStarsInBin=nStarsInBinM(dM,a_dMMin,a_dMMax);
for(int j=0;j<iStarsInBin;j++)
{
iIndex=Random(g_c_iStars-1);
while(bInit[iIndex])
{
iIndex=Random(g_c_iStars-1);
}
bInit[iIndex]=true;
stars[iIndex]->m_dMass=dM;
iNInit++;
}
}
if(iNInit<g_c_iStars)
{
//it could happen that there are sill a couple who didn't get placed...
//so place any unpaced stars completely randomly
//spOut<<",,,,Extras"<<std::endl;
for(int i=0;i<g_c_iStars;i++)
{
if(!bInit[i])
{
stars[i]->m_dMass=fRandom(a_dMMin,a_dMMax);
bInit[i]=true;
iNInit++;
}
}
}
}

//R2 spatial functions
double roR2(double a_dR,double a_dR0)
{
return g_c_iStars*pow(a_dR,2)/(2*pow(a_dR0,3))*exp(-1*a_dR/a_dR0);
}

double tailR2p(double a_dR,double a_dR0)
{
double dTerm1,dTerm2;

dTerm1=exp(-1*a_dR/a_dR0)*g_c_iStars*2*(a_dR0+a_dR)/(2*pow(a_dR0,2));
dTerm2=exp(-1*a_dR/a_dR0)*g_c_iStars*(2*pow(a_dR0,2)+2*a_dR0*a_dR+pow(a_dR,2))/(2*pow(a_dR0,3));

return dTerm1 - dTerm2;
}

double tailR2(double a_dR,double a_dR0)
{
return exp(-1*a_dR/a_dR0)*g_c_iStars*(2*pow(a_dR0,2) + 2*a_dR0*a_dR + pow(a_dR,2))/(2*pow(a_dR0,2));
}

double rMidR2(int a_iI,double a_dR0,double a_dROne)
{
double dDR=a_dROne/g_c_iNBins;

return  dDR/2 + (a_iI-1)*dDR;
}

int nStarsInBinRR2(double a_dR,double a_dR0,double a_dROne)
{
double dDR=a_dROne/g_c_iNBins;
return round(roR2(a_dR,a_dR0)*dDR);
}

double rOneR2(double a_dR0,double a_dRInit)
{
const double c_dErrorTollerence=1E-10;//error convergence tollerance
const int c_iNIterations = 100;//maximum iterations to try befroe giving up
double dError=1;
double dR,dNR;

dR=a_dRInit;

for(int i=0;(i<c_iNIterations&&dError>c_dErrorTollerence);i++)
{
```

```
dNR=tailR2(dR,a_dR0);
dError = abs(dNR-1);
dR=dR - (dNR-1)/tailR2p(dR,a_dR0);
}
return dR;
}

void spatialyDistributeMidR2(star *stars[],const double a_dR0, double &a_dROne)
{
int iNInit=0;
int iStarsInBin,iIndex;
double dR,dR1;
bool bInit[g_c_iStars];

dR1=rOneR2(a_dR0,a_dR0);
a_dROne=dR1;

for(int i=0;i<g_c_iStars;i++)
{
bInit[i]=false;
}

for(int i=1;i<=g_c_iNBins;i++)
{
dR=rMidR2(i,a_dR0,dR1);
iStarsInBin=nStarsInBinRR2(dR,a_dR0,dR1);
for(int j=0;j<iStarsInBin;j++)
{
iIndex=Random(g_c_iStars-1);
while(bInit[iIndex])
{
iIndex=Random(g_c_iStars-1);
}
bInit[iIndex]=true;
stars[iIndex]->m_vPosition=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*dR;
iNInit++;
}
}
if(iNInit<g_c_iStars)
{
//put one at eOne
//hopefully there arn't to many so rather than randomize i will be systematic
//so the first uninitiated star I find I initalize to be at eOne
for(int i=0;i<g_c_iStars;i++)
{
if(!bInit[i])
{
dR=dR1;
stars[i]->m_vPosition=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*dR;
bInit[i]=true;
iNInit++;
//kill the for
break;
}
}
//it could happen that there are sill a couple who didn't get placed...
//so place any unpaced stars completely randomly
for(int i=0;i<g_c_iStars;i++)
{
if(!bInit[i])
{
dR=fRandom(dR1);
stars[i]->m_vPosition=Vector3d(fRandom(-1,1),fRandom(-1,1),fRandom(-1,1)).Normal()*dR;
bInit[i]=true;
iNInit++;
}
}
}
}
//distrobution.h
//by Mike Williams 2-2-05

#ifndef distrobution_h
#define distrobution_h

#include<math.h>
#include"constants.h"
#include"star6_0.h"

//spatial distrobution constants
const int g_c_iNBins=15;
```

```
//spatial functions
double ro(double a_dR,double a_dR0);
double rMid(int a_iI,double a_dR0);
int nStarsInBinR(double a_dR,double a_dR0);
double rOne(double a_dR0);

//more appropriate boltzmann shaped spatial functions
double roR2(double a_dR,double a_dR0);
double rMidR2(int a_iI,double a_dR0,double a_dROne);
int nStarsInBinRR2(double a_dR,double a_dR0,double a_dROne);
double rOneR2(double a_dR0,double a_dRInit);
double tailR2p(double a_dR,double a_dR0);
double tailR2(double a_dR,double a_dR0);

//energy functions
double boltzmann(double a_dE,double a_dT);
double eMid(int a_iI,double a_dT);
int nStarsInBinE(double a_dE,double a_dT);
double eOne(double a_dT);

//energy functions for v^2Exp[-E/kt]
double boltzmannV2(double a_dE,double a_dT);
double eMid(int a_iI,double a_dT,double a_dE1);
int nStarsInBinEV2(double a_dE,double ad_T,double a_dE1);
double eOneV2(double a_dT,double a_dE0);
double tailV2p(double a_dE, double a_dT);
double tailV2(double a_dE,double a_dT);

//mass functions
double massPower(double a_dM,double a_dMMin, double a_dMMax);
double mMid(int a_iI,double a_dMMin, double a_dMMax);
int nStarsInBinM(double a_dM,double a_dMMin, double a_dMMax);

void uniformMassDistribute(star *stars[]);
void powerMassDistribute(star *stars[],double a_dMMin, double a_dMMax);

void spatialyDistributeMid(star *stars[],const double a_dR0);
void spatialyDistributeMidR2(star *stars[],const double a_dR0, double &a_dROne);
void energyDistribute(star *stars[],double &a_dT);
void energyDistributeV2(star *stars[],double &a_dT,double &a_dE1);
void energyDistribute(star *stars[]);
void energyDistributeV2(star *stars[]);

#endif
//energy2_0.cpp Mike Williams 2/05
#include"star6_0.h"
#include"constants.h"
#include"Vector3d.h"
#include<math.h>
#include"energy1_0.h"

double energyNotVerlet(star *stars[],const int &n,double &PE, double &KE)
{
PE=0;
KE=0;
for(int i = 0; i<n;i++)
{
for(int j =i+1; j<n; j++)
{
PE+=-stars[i]->m_dMass*stars[j]->m_dMass/(stars[i]->m_vPosition-stars[j]->m_vPosition).Size();
}//end iner for
KE+=.5*stars[i]->m_dMass*stars[i]->m_vVelocity.SizeSquared();
}//end outer for
return KE+PE;
}

double energyNotVerlet(star *stars[],double Rs[g_c_iStars][g_c_iStars],const int &n)
//computes the energy for a non verlet integrator (assumes the velocity is current)
{
double dPE=0;
double dKE=0;
for(int i = 0; i<n;i++)
{
for(int j =i+1; j<n; j++)
{
dPE+=-stars[i]->m_dMass*stars[j]->m_dMass/sqrt(Rs[i][j]);
}//end iner for
dKE+=.5*stars[i]->m_dMass*stars[i]->m_vVelocity.SizeSquared();
}//end outer for
return dKE+dPE;
}

double energyNotVerlet(star *stars[],double Rs[g_c_iStars][g_c_iStars],const int &n,double &PE, double &KE)
```

```
//computes the energy for a non verlet integrator (assumes the velocity is current)
{
PE=0;
KE=0;
for(int i = 0; i<n;i++)
{
for(int j =i+1; j<n; j++)
{
PE+=-stars[i]->m_dMass*stars[j]->m_dMass/sqrt(Rs[i][j]);
}//end iner for
KE+=.5*stars[i]->m_dMass*stars[i]->m_vVelocity.SizeSquared();
}//end outer for
return KE+PE;
}

double energyP(star *stars[])
{
double dPE;
Vector3d vR;
dPE=0;

for(int i = 0; i<g_c_iStars;i++)
{
for(int j =i+1; j<g_c_iStars; j++)
{
vR=stars[i]->m_vPosition - stars[j]->m_vPosition;
dPE+=-g_c_dSG*stars[i]->m_dMass*stars[j]->m_dMass/vR.Size();
}//end iner for
}
return dPE;
}
double energy(star *stars[])
{
double dPE,dKE;
Vector3d vR;
dPE=0;
dKE=0;

for(int i = 0; i<g_c_iStars;i++)
{
for(int j =i+1; j<g_c_iStars; j++)
{
vR=stars[i]->m_vPosition - stars[j]->m_vPosition;
dPE+=-g_c_dSG*stars[i]->m_dMass*stars[j]->m_dMass/vR.Size();
}//end iner for
dKE+=.5*stars[i]->m_dMass*stars[i]->m_vVelocity.SizeSquared();
}
return dPE+dKE;
}
double energy(star *stars[],double &dPE,double &dKE)
{
Vector3d vR;
dPE=0;
dKE=0;

for(int i = 0; i<g_c_iStars;i++)
{
for(int j =i+1; j<g_c_iStars; j++)
{
vR=stars[i]->m_vPosition - stars[j]->m_vPosition;
dPE+=-g_c_dSG*stars[i]->m_dMass*stars[j]->m_dMass/vR.Size();
}//end iner for
dKE+=.5*stars[i]->m_dMass*stars[i]->m_vVelocity.SizeSquared();
}
return dPE+dKE;
}
//energy2_0.h Mike Williams 2/05
#ifndef _energy_h
#define _energy_h

#include"star6_0.h"
#include"constants.h"

double energyNotVerlet(star *stars[],double Rs[g_c_iStars][g_c_iStars],const int &n);
double energyNotVerlet(star *stars[],double Rs[g_c_iStars][g_c_iStars],const int &n,double &PE, double &KE);
double energyNotVerlet(star *stars[],const int &n,double &PE, double &KE);
double energyP(star *stars[]);
double energy(star *stars[]);
double energy(star *stars[],double &dPE,double &dKE);
//#include"energy1_0.cpp"

#endif
/*group1_0.cpp
```

```
 *Written by Mike Williams 07-16-03
 has getMembers function
*/

#include"group1_0.h"
#include<iostream>

group::group()
{//Constructor
clear();
}

group::~group()
{//destructor
//do nothing
}

bool group::isMember(const int &index)
{//search m_iMembers[] return true if index is present
for(int i = 0; i<m_iNumber; i++)
{
if(m_iMembers[i]==index)
return true;
}
return false;
}

void group::addMember(const int &index)
{//add index to list
if(!isMember(index))
{
m_iMembers[m_iNumber]=index;
m_iNumber++;
}
}

void group::removeMember(const int &index)
{
for(int i = 0; i<m_iNumber-1; i++)
{
if(m_iMembers[i]==index)
{
m_iMembers[i]=m_iMembers[m_iNumber];
m_iMembers[m_iNumber]=UNUSED_INDEX;
m_iNumber--;
}
}
if(m_iMembers[m_iNumber]==index)
{
m_iMembers[m_iNumber]=UNUSED_INDEX;
m_iNumber--;
}
}

void group::getMembers(int members[],int &n)
{
for(int i = 0; i<m_iNumber;i++)
{
members[i]=m_iMembers[i];
}
n=m_iNumber;
}

void group::clear()
{
m_iNumber = 0;
for(int i = 0; i<g_c_iMaxSystem; i++)
{
m_iMembers[i]=UNUSED_INDEX;
}
m_dDt=1E10;//set dt arbitrarily large so that the first suggestion is taken
}

int group::numberOfMembers()
{
return m_iNumber;
}

void group::suggestDt(const double &a_dDt)
{
if(a_dDt<m_dDt)//ensure that the smallest dt of the system is used
{
m_dDt=a_dDt;
```

```
}
}
/*group1_0.h
 *Written by Mike Williams 07-16-04
 *was system renamed to group since system is a reserved word in this version of c++
 *Added a remove member function


*/

#ifndef group_h
#define group_h
//#include"star5_1.h"
#include"analytical.h"

const int g_c_iMaxSystem = 10;//maxiumum stars to expect in a system
const int UNUSED_INDEX = -1;//a tag to identify that an aray element contains no data

class group
{
public:
//constructors, destructor
group();
~group();

//functions
bool isMember(const int &index);//search m_iMembers[] return true if index is present
void addMember(const int &index);//add index to list
void removeMember(const int &index);//remove index from the list
void clear();
void getMembers(int members[],int &n);//tell me who is in the system
int numberOfMembers();//return m_iNumber
void suggestDt(const double &a_dDt);

//public variables
double m_dDt;

private:
//Variables
int m_iMembers[g_c_iMaxSystem];//aray to store member indicies
int m_iNumber;//number of stars in system
};


#endif
/*main4_0_RealDistroSwaper.cpp
Written by Mike Williams 05-16-05
Built to exchange real distrobutions with others
Step by steps
Generate a distro and ouptut it in real units.
Send it to your friends
Load the same distro
convert it to code units
run simulation
compare to friends.
Implements classic RK4 adaption
Takes arguments to determin distro size
*/
#include<iostream>
#include<fstream>
#include<math.h>
#include"star6_0.h"
#include"group1_0.h"
#include"constants.h"
#include"Vector3d.h"
#include"rand.h"
#include<sys/time.h>
#include"rk2_1.h"
#include"energy1_1.h"
#include"distrobution.h"
#include"distroIO1_2.h"
#include"util.h"
#include"binaryCount.h"
#include"outputManager1_1.h"
#include"adaptFunctions.h"
#include"distroStats.h"



//so stars is an aray of pointers to stars to be updated, n is the number of stars in the aray,
//t is the total time to go through, and dt is the time step
```

```
int main(int argc, char *argv[])
{
/*
****************************************************************************************************
Declare variables
****************************************************************************************************
*/

char *cEnd;
//units
double dR0,dL,dT,dRMin,dError0;

//pars input
dR0=strtod(argv[1],&cEnd);
std::cout<<dR0<<std::endl;

dL=dR0;
dT=sqrt(dL*dL*dL/(g_c_dG*g_c_dM));
dRMin=g_c_dRMinM/dL;
dError0=g_c_dError0m/dL;


//Sim constants
//using namespace std;
const int c_iNDataPtsS=1000;
const int c_iNDataPtsF=1000;
const double c_dChangePerOutputS=1/(double)c_iNDataPtsS;
const double c_dChangePerOutputF=1/(double)c_iNDataPtsF;
const int c_iNSteps=100000;
const double c_dSimTime =g_c_dClusterTimes*sqrt(dR0*dR0*dR0/(g_c_dG*g_c_dM))/dT;
double dLastTPS,dLastTPF;//last time percent screen, file
double dDt = 1E-5;
double dDtNew;

char sAdaptDesc[13]="2StepRk4Soft";

double dErrorP;


outputManager allOutput;
//const double c_dSimTime =2.418;//Total time for cecil example
//loading related variables
const bool c_bLoad = false;//to load the
const bool c_bSave = true;//to save or not to save

//Sim variables
double dTime = 0;
double dDtMin=dDt;
double dStopTime = c_dSimTime;
star *stars[g_c_iStars],*starsp[g_c_iStars],*stars1[g_c_iStars];
double dEffort=0;//a number representing the effort the code has done so far. So if you are in
//the main loop it is time steps*Total stars sqyared. however if you are
//in the small group processing it is time steps*the number of stars in that small group squared

//Output variables and temps

//Energy related
double dPE=0;//total potential engergy
double dKE=0;//total kenetic energy
double dEnergy;
double dMinR;
double dROne;

//Noodle related

//Binary related
double dRs[g_c_iStars][g_c_iStars];
int iNGroups,iNMembers;
int iMembers[g_c_iMaxSystem];
// double dR;// 0 to c_dRdist during init, in sim radious of an interaction
int iNaries[g_c_iMaxSystem];
int iNOrbits=0;
int iBinary;
bool bSplody=false;

Vector3d vVCm,vCm;
double dTotalMass;


std::ofstream ruOut,constOut,massOut;
std::ifstream saveIn;

//Output controles
```

```
int iCounter = 0;//used to controle number of outputs
int c_iOutDevide = c_dSimTime/dDt/c_iNDataPtsS;//output will only happen every c_iOutDevide time

c_iOutDevide = c_dSimTime/dDt/c_iNDataPtsS;//output will only happen every c_iOutDevide time
dTime=0;
iCounter=0;
dEffort=0;

/*
**************************************************************************************************
Initialize variables
**************************************************************************************************
*/



dTime=0;
allOutput.openOpt();

for(int i = 0; i<g_c_iStars; i++)
{
stars[i]=new star;
stars1[i]=new star;
starsp[i]=new star;
}

if(c_bLoad)//get the initial conditions from file
{
//saveIn.open("./codeIo/input/clusterDistro.txt");
saveIn.open("./codeIo/input/50VMass4-22.txt");
//saveIn.open("./codeIo/input/50VariableMass.txt");

//saveIn.open("./codeIo/input/4BODY.TXT");
loadDistroRealUnits(stars,saveIn,dL,dT);
saveIn.close();
}
else//generate the initial conditions yourself
{
    srand( (unsigned)time( NULL ) );
//set up stars
//uniformMassDistribute(stars);
powerMassDistribute(stars,.5,10);
spatialyDistributeMidR2(stars,dR0/dL,dROne);
//energyDistribute(stars);
double dJunk;
//energyDistributeV2(stars,dJunk,dJunk);
energyDistributeV2(stars);
//store the distrobution for repetition
}
//Initialize output files
ruOut.open("./codeIo/clusterDistro.txt");
saveDistroRealUnits(stars,ruOut,dL,dT);
ruOut.close();

constOut.open("./codeIo/const.txt");
saveConst(constOut,sAdaptDesc,dR0,dL);
constOut.close();

//allOutput.doCmOut(stars,dTime);
massOut.open("./codeIo/averageMass.txt");
massOut<<"Average Mass "<<AveMass(stars)<<std::endl;


vVCm.SetZero();
vCm.SetZero();
dTotalMass=0;
for(int i=0;i<g_c_iStars;i++)
{
dTotalMass+=stars[i]->m_dMass;
vVCm+=stars[i]->m_vVelocity*stars[i]->m_dMass;
vCm+=stars[i]->m_vPosition*stars[i]->m_dMass;
}
vVCm/=dTotalMass;
vCm/=dTotalMass;

for(int i=0;i<g_c_iStars;i++)
{
stars[i]->m_vPosition-=vCm;
stars[i]->m_vVelocity-=vVCm;
}

allOutput.doDistroOut(stars,dTime,dDt);
allOutput.doInitialOutOpt(stars,dTime);
```

```
//allOutput.doCmOut(stars,dTime);
//do 5 integration steps
for(int k=0;k<5;k++)
{
for(int c=0; c<g_c_iStars;c++)
{
//reset the stars for the next step
stars[c]->m_vAcceleration.SetZero();
stars[c]->m_vTmpAcceleration.SetZero();
stars[c]->m_vTmpAcceleration2.SetZero();
stars[c]->m_vTmpAcceleration3.SetZero();
stars[c]->m_bClose=false;
}

rk42StepSoft(stars,starsp,stars1,dRs,dDt,dDtNew,dEffort,dError0,dRMin);

dTime+=dDt;
dDt=dDtNew;
dLastTPS=dTime/dStopTime;
dLastTPF=dTime/dStopTime;

// allOutput.doMcNeilOut(stars,dTime,7,47);
//allOutput.doMcNeilOut(stars,dTime,0,1);
allOutput.doFileOutOpt(stars,dTime,dEnergy);
allOutput.doScreenOutOpt(stars,dEnergy,dTime,c_dSimTime,dDt);


}
for(int c=0; c<g_c_iStars;c++)
{
//reset the stars for the next step
stars[c]->m_vAcceleration.SetZero();
stars[c]->m_vTmpAcceleration.SetZero();
stars[c]->m_vTmpAcceleration2.SetZero();
stars[c]->m_vTmpAcceleration3.SetZero();
stars[c]->m_bClose=false;
}

/*
*****************************************************************************************************
Main Loop
*****************************************************************************************************
*/
dKE=dPE=0;
for(;dTime<dStopTime;dKE=0,dPE=0)//main loop, controles when the sim stops
//for(int iSteps=0;iSteps<c_iNSteps;iSteps++,dTime+=dDt,dKE=0,dPE=0)
{
//eliminate old forces
for(int c=0; c<g_c_iStars;c++)
{
//reset the stars for the next step
stars[c]->m_vAcceleration.SetZero();
stars[c]->m_vTmpAcceleration.SetZero();
stars[c]->m_vTmpAcceleration2.SetZero();
stars[c]->m_vTmpAcceleration3.SetZero();
stars[c]->m_bClose=false;
if(stars[c]->m_vPosition.X !=stars[c]->m_vPosition.X)
{
std::cout<<"Warning star "<<c<<" is broken at time "<<dTime<<std::endl;
for(int i=c+1;i<g_c_iStars;i++)
{
if(stars[i]->m_vPosition.X !=stars[c]->m_vPosition.X)
{
std::cout<<"Warning star "<<i<<" is broken"<<std::endl;
}
}
exit(1);
}
}

rk42StepSoft(stars,starsp,stars1,dRs,dDt,dDtNew,dEffort,dError0,dRMin);
dTime+=dDt;
dDt=dDtNew;

//controle output
if((dTime/dStopTime-dLastTPF)>=c_dChangePerOutputF)
{
dLastTPF=dTime/dStopTime;
//allOutput.doMcNeilOut(stars,dTime,22,40);
//allOutput.doMcNeilOut(stars,dTime,7,47);
//allOutput.doMcNeilOut(stars,dTime,0,1);
allOutput.doFileOutOpt(stars,dTime,dEnergy);
}//endif
```

```
if((dTime/dStopTime-dLastTPS)>=c_dChangePerOutputS)
{
dLastTPS=dTime/dStopTime;
//allOutput.doMcNeilOut(stars,dTime,0,1);
//allOutput.doMainScrenOut1(stars,dRs,dTime,c_dSimTime,dDt);
allOutput.doScreenOutOpt(stars,dEnergy,dTime,c_dSimTime,dDt);
}//endif
iCounter++;
}//outermost while

//do one last outputs
//allOutput.doMainOut(stars,dTime,dRs);
allOutput.doFileOutOpt(stars,dTime,dEnergy);


//allOutput.doCmOut(stars,dTime);
//allOutput.close();
allOutput.closeOpt();

allOutput.doFinalScreenOut();
for(int i = 0; i<g_c_iStars; i++)
{
delete stars[i];
delete stars1[i];
delete starsp[i];
}

system("xmms -p /mnt/ROUTER/public/reaserch/sounds/done.wav&");

return 1;
}
/*outputManager1_1.cpp
Written by Mike Williams 04-13-05
*/

#include"outputManager1_1.h"
#include<fstream>
#include<iostream>
#include"group1_0.h"
#include"Vector3d.h"
#include"distroIO.h"
#include"binaryCount.h"
#include"energy1_1.h"
#include"util.h"
#include"distroStats.h"
#include"binarytracker.h"
#include<sys/time.h>

void outputManager::open()
{
system("rm ./codeIo/last/*.csv");
system("rm ./codeIo/last/*.txt");
system("cp ./codeIo/*.* ./codeIo/last/");
system("rm ./codeIo/*.csv");
system("rm ./codeIo/*.txt");

mcOut.open("./codeIo/mcneil.csv");

eOut.open("./codeIo/energy.csv");

noodleOut.open("./codeIo/noodle.csv");

rOut.open("./codeIo/rs.csv");

binaryOut.open("./codeIo/binary.csv");

cmOut.open("./codeIo/cm.csv");

rvOut.open("./codeIo/rvMike.txt");

abOut.open("./codeIo/averageBinary.csv");

dtOut.open("./codeIo/dt.csv");

bIndexOut.open("./codeIo/binaryIndex.csv");


//initialize files
mcOut<<"T,DE,Nb,xi,yi,zi,xj,yj,zj,rij,cm,vcm,Rrms,Vrms,DRrms,Lx,Ly,Lz"<<std::endl;
bIndexOut<<"Time,Binaries,i0,j0,r0,i1,j1,r1"<<std::endl;
dtOut<<"Time,Dt"<<std::endl;
noodleOut<<"Time,";
rOut<<"Time,";
```

```
binaryOut<<"Time,Binaries"<<std::endl;
rvOut<<"T ";
for(int i = 0; i<g_c_iStars;i++)
{
for(int j =i+1; j<g_c_iStars; j++)
{
rOut<<"r"<<i<<"-"<<j<<",";
}
rvOut<<"Rx"<<i<<" Ry"<<i<<" Rz"<<i<<" Vx"<<i<<" Vy"<<i<<" Vz"<<i<<" ";
noodleOut<<"X"<<i<<","<<"Y"<<i<<","<<"Z"<<i<<",";
}
rvOut<<std::endl;
rOut<<std::endl;
noodleOut<<std::endl;
eOut<<"T,Etot,PE,KE,abs(E-.5PE)/(-Eo),abs(E-Eo)/(-Eo)"<<std::endl;

cmOut<<"t,CMx,CMy,CMz,VCMx,VCMy,VCMz"<<std::endl;
abOut<<"t,AverageBinary,=Sum(#Binaries*dt)/dTime"<<std::endl;
m_dAvBinSum=0;
m_dLastT=0;

}
void outputManager::doMainOut(star *stars[],const double &dTime, double dRs[g_c_iStars][g_c_iStars])
{
int iBinary,iNGroups,iNMembers;
int iNaries[g_c_iMaxSystem];
int iMembers[g_c_iMaxSystem];
double dPE,dKE,dEnergy;

iBinary=binaryCount(stars);
binaryOut<<dTime<<","<<iBinary<<std::endl;

m_dAvBinSum+=iBinary*(dTime-m_dLastT);
abOut<<dTime<<","<<m_dAvBinSum/dTime<<std::endl;
m_dLastT=dTime;

for(int i=0;i<g_c_iMaxSystem;i++)
iNaries[i]=0;

noodleOut<<dTime<<",";
rOut<<dTime<<",";
rvOut<<dTime<<" ";
for(int i = 0; i<g_c_iStars; i++)
{
for(int j =i+1; j<g_c_iStars; j++)
{
rOut<<sqrt(dRs[i][j])<<",";
}
rvOut<<stars[i]->m_vPosition.X<<" "<<stars[i]->m_vPosition.Y<<" "<<stars[i]->m_vPosition.Z<<" "
     <<stars[i]->m_vVelocity.X<<" "<<stars[i]->m_vVelocity.Y<<" "<<stars[i]->m_vVelocity.Z<<" ";
noodleOut<<stars[i]->m_vPosition.X<<","<<stars[i]->m_vPosition.Y<<","<<stars[i]->m_vPosition.Z<<",";
}
rvOut<<std::endl;
rOut<<std::endl;
noodleOut<<std::endl;
dEnergy=energyNotVerlet(stars,dRs,g_c_iStars,dPE,dKE);
eOut<<dTime<<","<<dEnergy<<","<<dPE<<","<<dKE<<","<<abs(dEnergy-.5*dPE)/-m_dE0<<","<<abs(dEnergy-m_dE0)/-m_dE0<<std::endl;
}
void outputManager::doMcNeilOut(star *stars[],const double dTime,int i, int j)
{
double dEnergy,dCm,dVCm,dTotalMass,dRrms,dVrms,dDRrms;
Vector3d vVCm,vCm,vL;

dEnergy=energy(stars);
for(int i=0;i<g_c_iStars;i++)
{
dTotalMass+=stars[i]->m_dMass;
vVCm+=stars[i]->m_vVelocity*stars[i]->m_dMass;
vCm+=stars[i]->m_vPosition*stars[i]->m_dMass;
}
vVCm/=dTotalMass;
vCm/=dTotalMass;
dCm=vCm.Size();
dVCm=vVCm.Size();

RVDRrmsL(stars,dRrms,dVrms,dDRrms,vL);

mcOut<<dTime<<","<<abs(dEnergy-m_dE0)/-m_dE0<<","<<binaryCount(stars)<<","<<stars[i]->m_vPosition.X<<","<<stars[i]->m_vPosition.Y
<<","<<stars[i]->m_vPosition.Z<<","<<stars[j]->m_vPosition.X<<","<<stars[j]->m_vPosition.Y<<","<<stars[j]->m_vPosition.Z<<","
<<(stars[i]->m_vPosition - stars[j]->m_vPosition).Size()<<","<<dCm<<","<<dVCm<<","<<dRrms<<","<<dVrms<<","
<<dDRrms<<","<<vL.X<<","<<vL.Y<<","<<vL.Z<<std::endl;
}
```

```
void outputManager::doScreenOutOpt(star *stars[],const double &dE, const double &dTime, const double &dStopTime,const double c_dDt)
{
double dTimeOut;
char sTimeOutTag;

gettimeofday(&m_tEnd, NULL);
dTimeOut=(m_tEnd.tv_sec+((m_tEnd.tv_usec/1000000.0))-(m_tStart.tv_sec+(m_tStart.tv_usec/1000000.0)))/(dTime/dStopTime)-
(m_tEnd.tv_sec+((m_tEnd.tv_usec/1000000.0))-(m_tStart.tv_sec+(m_tStart.tv_usec/1000000.0)));
//convert to better units
if(dTimeOut<60)//time best measured in seconds
{
sTimeOutTag='s';
}
else if(dTimeOut<3600)//time best measured in minutes
{
dTimeOut/=60.0;
sTimeOutTag='m';
}
else if(dTimeOut<86400)
{
dTimeOut/=3600.0;
sTimeOutTag='h';
}
else
{
dTimeOut/=86400.0;
sTimeOutTag='d';
}
std::cout<<dTime/dStopTime<<" "<<dTimeOut<<sTimeOutTag<<" "<<abs(dE-m_dE0)/-m_dE0<<" "<<c_dDt<<std::endl;
}
void outputManager::doFileOutOpt(star *stars[],const double &dTime,double &dEnergy)
{
double dCm,dVCm,dTotalMass,dRrms,dVrms,dDRrms,dPE,dKE;
double dMu,dEP,dRMin,dR,dRTemp,dMinR,dSplodyTime;
Vector3d vVCm,vCm,vL,vR,vV;
int iBinary,iRight,iLeft;
char sSplodyTag;

dTotalMass=0;
dPE=0;
dKE=0;
iBinary=0;
dRrms=0;
dVrms=0;
dDRrms=0;
dMinR=1E100;
vL.SetZero();
for(int i=0;i<g_c_iStars;i++)
{
//calc cCM and CM sums
dTotalMass+=stars[i]->m_dMass;
vVCm+=stars[i]->m_vVelocity*stars[i]->m_dMass;
vCm+=stars[i]->m_vPosition*stars[i]->m_dMass;

for(int j =i+1; j<g_c_iStars; j++)
{
//Binary counting stuff
dMu=stars[i]->m_dMass*stars[j]->m_dMass/(stars[i]->m_dMass+stars[j]->m_dMass);
vV=stars[i]->m_vVelocity - stars[j]->m_vVelocity;
vR=stars[i]->m_vPosition-stars[j]->m_vPosition;
dR=vR.Size();
if(dR<dMinR)
{
dMinR=dR;
}
dEP=0.5*dMu*vV.SizeSquared() - stars[i]->m_dMass*stars[j]->m_dMass/dR;
if(dEP<0)//could be a binary check for local stars
{
dRMin=100*dR;
for(int k=0;k<g_c_iStars;k++)
{
if((k!=i)&&(k!=j))
{
dRTemp=(stars[i]->m_vPosition-stars[k]->m_vPosition).Size();
if(dRTemp<dRMin)
dRMin=dRTemp;

dRTemp=(stars[j]->m_vPosition-stars[k]->m_vPosition).Size();
if(dRTemp<dRMin)
dRMin=dRTemp;
}
}
```

```
if(dRMin>g_c_iRMultipy*dR)
{
iBinary++;
m_binTracker.addBinary(i,j,dR,stars[i]->m_dMass,stars[j]->m_dMass,dTime);
}
}//end energy if

//calc PE
dPE+=-g_c_dSG*stars[i]->m_dMass*stars[j]->m_dMass/vR.Size();

//Average seperation
dDRrms+=vR.SizeSquared();
}//end iner for
//Calc KE
dKE+=.5*stars[i]->m_dMass*stars[i]->m_vVelocity.SizeSquared();
//rms quantities
dRrms+=stars[i]->m_vPosition.SizeSquared();
dVrms+=stars[i]->m_vVelocity.SizeSquared();
vL+=stars[i]->m_vPosition.Cross(stars[i]->m_vVelocity)*stars[i]->m_dMass;
}
//calc cCM and CM
vVCm/=dTotalMass;
vCm/=dTotalMass;
dCm=vCm.Size();
dVCm=vVCm.Size();

//rms finalization
dRrms=sqrt(dRrms/(double)g_c_iStars);
dVrms=sqrt(dVrms/(double)g_c_iStars);
dDRrms=sqrt(dDRrms/g_c_dPairs);

dEnergy=dPE+dKE;

m_dAvBinSum+=iBinary*(dTime-m_dLastT);
m_dLastT=dTime;

m_binTracker.checkTerm(dTime);

optOut<<dTime<<","<<dEnergy<<","<<abs(dEnergy-m_dE0)/-m_dE0<<","<<abs(dEnergy-.5*dPE)/-m_dE0<<","<<iBinary<<","<<m_dAvBinSum/dTime<<","
<<m_binTracker.m_dAveR<<","<<m_binTracker.m_dAveT<<","<<m_binTracker.m_dAvePM<<","<<m_binTracker.m_dAveSM<<","<<dMinR<<","<<dCm<<","
<<dVCm<<","<<dRrms<<","<<dVrms<<","<<dDRrms<<","<<vL.Size()<<std::endl;
if((dEnergy>0&&!m_bSplody)||abs(dEnergy-m_dE0)/-m_dE0>0.01)
{
system("xmms -p /mnt/ROUTER/public/reaserch/sounds/splody.wav&");
m_bSplody=true;
gettimeofday(&m_tEnd, NULL);
dSplodyTime=(m_tEnd.tv_sec+((m_tEnd.tv_usec/1000000.0))-(m_tStart.tv_sec+(m_tStart.tv_usec/1000000.0)));
//convert to better units
if(dSplodyTime<60)//time best measured in seconds
{
sSplodyTag='s';
}
else if(dSplodyTime<3600)//time best measured in minutes
{
dSplodyTime/=60.0;
sSplodyTag='m';
}
else if(dSplodyTime<86400)
{
dSplodyTime/=3600.0;
sSplodyTag='h';
}
else
{
dSplodyTime/=86400.0;
sSplodyTag='d';
}

splodyOut.open("./codeIo/splody.txt");
splodyOut<<"Simulation Time of explosion "<<dTime<<std::endl;
splodyOut<<"Run Time of explosion "<<dSplodyTime<<sSplodyTag<<std::endl;
splodyOut.close();
exit(1);
}

}
void outputManager::openOpt()
{
system("rm ./codeIo/last/*.csv");
system("rm ./codeIo/last/*.txt");
system("cp ./codeIo/*.* ./codeIo/last/");
system("rm ./codeIo/*.csv");
system("rm ./codeIo/*.txt");
```

```
optOut.open("./codeIo/output.csv");
optOut<<"T,E,DE,DV,Nb,AB,ABR,ABLT,ABPM,ABSM,MinR,cm,vcm,Rrms,Vrms,DRrms,L"<<std::endl;
}
void outputManager::doInitialOutOpt(star *stars[],const double &dTime)
{
double dEnergy,dCm,dVCm,dTotalMass,dRrms,dVrms,dDRrms,dPE,dKE;
double dMu,dEP,dRMin,dR,dRTemp,dMinR;
Vector3d vVCm,vCm,vL,vR,vV;
int iBinary,iRight,iLeft;

dTotalMass=0;
dPE=0;
dKE=0;
iBinary=0;
dRrms=0;
dVrms=0;
dDRrms=0;
dMinR=1E100;
vL.SetZero();
for(int i=0;i<g_c_iStars;i++)
{
//calc cCM and CM sums
dTotalMass+=stars[i]->m_dMass;
vVCm+=stars[i]->m_vVelocity*stars[i]->m_dMass;
vCm+=stars[i]->m_vPosition*stars[i]->m_dMass;

for(int j =i+1; j<g_c_iStars; j++)
{
//Binary counting stuff
dMu=stars[i]->m_dMass*stars[j]->m_dMass/(stars[i]->m_dMass+stars[j]->m_dMass);
vV=stars[i]->m_vVelocity - stars[j]->m_vVelocity;
vR=stars[i]->m_vPosition-stars[j]->m_vPosition;
dR=vR.Size();
if(dR<dMinR)
{
dMinR=dR;
}
dEP=0.5*dMu*vV.SizeSquared() - stars[i]->m_dMass*stars[j]->m_dMass/dR;
if(dEP<0)//could be a binary check for local stars
{
dRMin=100*dR;
for(int k=0;k<g_c_iStars;k++)
{
if((k!=i)&&(k!=j))
{
dRTemp=(stars[i]->m_vPosition-stars[k]->m_vPosition).Size();
if(dRTemp<dRMin)
dRMin=dRTemp;

dRTemp=(stars[j]->m_vPosition-stars[k]->m_vPosition).Size();
if(dRTemp<dRMin)
dRMin=dRTemp;
}
}

if(dRMin>g_c_iRMultipy*dR)
{
iBinary++;
m_binTracker.addBinary(i,j,dR,stars[i]->m_dMass,stars[j]->m_dMass,dTime);
}
}//end energy if

//calc PE
dPE+=-g_c_dSG*stars[i]->m_dMass*stars[j]->m_dMass/vR.Size();

//Average seperation
dDRrms+=vR.SizeSquared();
}//end iner for
//Calc KE
dKE+=.5*stars[i]->m_dMass*stars[i]->m_vVelocity.SizeSquared();
//rms quantities
dRrms+=stars[i]->m_vPosition.SizeSquared();
dVrms+=stars[i]->m_vVelocity.SizeSquared();
vL+=stars[i]->m_vPosition.Cross(stars[i]->m_vVelocity)*stars[i]->m_dMass;
}
//calc cCM and CM
vVCm/=dTotalMass;
vCm/=dTotalMass;
dCm=vCm.Size();
dVCm=vVCm.Size();

//rms finalization
```

```
dRrms=sqrt(dRrms/(double)g_c_iStars);
dVrms=sqrt(dVrms/(double)g_c_iStars);
dDRrms=sqrt(dDRrms/g_c_dPairs);

dEnergy=dPE+dKE;
m_dE0=dEnergy;

gettimeofday(&m_tStart, NULL);

m_dAvBinSum=0;
m_dLastT=0;

optOut<<dTime<<","<<dEnergy<<","<<abs(dEnergy-m_dE0)/-m_dE0<<","<<abs(dEnergy-.5*dPE)/-m_dE0<<","<<iBinary<<","
<<iBinary<<","<<m_binTracker.m_dAveR<<","<<m_binTracker.m_dAveT<<","<<m_binTracker.m_dAvePM<<","
<<m_binTracker.m_dAveSM<<","<<dMinR<<","<<dCm<<","<<dVCm<<","<<dRrms<<","<<dVrms<<","<<dDRrms<<","<<vL.Size()<<std::endl;
}
void outputManager::closeOpt()
{
optOut.close();
}
void outputManager::doMainOut1(star *stars[],const double &dTime,
double dRs[g_c_iStars][g_c_iStars],const double c_dDt)
{
int iBinary,iNGroups,iNMembers;
int iNaries[g_c_iMaxSystem];
int iMembers[g_c_iMaxSystem];
int iFirst[g_c_iStars/2];
int iSecond[g_c_iStars/2];
double dRBinary[g_c_iStars/2];
double dPE,dKE,dEnergy;

dtOut<<dTime<<","<<c_dDt<<std::endl;

iBinary=binaryCountIndex(stars,iFirst,iSecond,dRBinary);
binaryOut<<dTime<<","<<iBinary<<std::endl;
if(iBinary>0)
{
bIndexOut<<dTime<<","<<iBinary<<",";
}
for(int i=0;i<iBinary;i++)
{
bIndexOut<<iFirst[i]<<","<<iSecond[i]<<","<<dRBinary[i]<<",";
}
if(iBinary>0)
{
bIndexOut<<std::endl;
}


m_dAvBinSum+=iBinary*(dTime-m_dLastT);
abOut<<dTime<<","<<m_dAvBinSum/dTime<<std::endl;
m_dLastT=dTime;

noodleOut<<dTime<<",";
rOut<<dTime<<",";
rvOut<<dTime<<" ";
for(int i = 0; i<g_c_iStars; i++)
{
if(g_c_iStars<11)
{
for(int j =i+1; j<g_c_iStars; j++)
{
rOut<<sqrt(dRs[i][j])<<",";
}
}
rvOut<<stars[i]->m_vPosition.X<<" "<<stars[i]->m_vPosition.Y<<" "<<stars[i]->m_vPosition.Z<<" "
<<stars[i]->m_vVelocity.X<<" "<<stars[i]->m_vVelocity.Y<<" "<<stars[i]->m_vVelocity.Z<<" ";
noodleOut<<stars[i]->m_vPosition.X<<","<<stars[i]->m_vPosition.Y<<","<<stars[i]->m_vPosition.Z<<",";
}
rvOut<<std::endl;
rOut<<std::endl;
noodleOut<<std::endl;
dEnergy=energy(stars,dPE,dKE);
eOut<<dTime<<","<<dEnergy<<","<<dPE<<","<<dKE<<","<<abs(dEnergy-.5*dPE)/-m_dE0<<","<<abs(dEnergy-m_dE0)/-m_dE0<<std::endl;
}

void outputManager::doCmOut(star *stars[],const double &dTime)
{
Vector3d vVCm,vCm;
double dTotalMass;

vVCm.SetZero();
vCm.SetZero();
```

```
dTotalMass=0;
for(int i=0;i<g_c_iStars;i++)
{
dTotalMass+=stars[i]->m_dMass;
vVCm+=stars[i]->m_vVelocity*stars[i]->m_dMass;
vCm+=stars[i]->m_vPosition*stars[i]->m_dMass;
}
vVCm/=dTotalMass;
vCm/=dTotalMass;

cmOut<<dTime<<","<<vCm.X<<","<<vCm.Y<<","<<vCm.Z<<","<<vVCm.X<<","<<vVCm.Y<<","<<vVCm.Z<<std::endl;

}
void outputManager::doDistroOut(star *stars[],const double &dTime,const double &dDt)
{
saveOut.open("./codeIo/initDistro.txt");
saveDistro(stars,saveOut,dTime,dDt);
saveOut.close();
}
void outputManager::doInitialOut(star *stars[],const double &dTime)
{
int iBinary;
double dEnergy,dPE,dKE;

iBinary=binaryCount(stars);
m_iOldBinary=iBinary;
binaryOut<<dTime<<","<<iBinary<<std::endl;
noodleOut<<dTime*g_c_dYearConvert<<",";
rvOut<<dTime<<" ";
for(int i=0;i<g_c_iStars;i++)
{
noodleOut<<stars[i]->m_vPosition.X<<","<<stars[i]->m_vPosition.Y<<","<<stars[i]->m_vPosition.Z<<",";
rvOut<<stars[i]->m_vPosition.X<<" "<<stars[i]->m_vPosition.Y<<" "<<stars[i]->m_vPosition.Z<<" "
<<stars[i]->m_vVelocity.X<<" "<<stars[i]->m_vVelocity.Y<<" "<<stars[i]->m_vVelocity.Z<<" ";
}
rvOut<<std::endl;
noodleOut<<std::endl;
dEnergy=energyNotVerlet(stars,g_c_iStars,dPE, dKE);
m_dE0=dEnergy;
eOut<<dTime<<","<<dEnergy<<","<<dPE<<","<<dKE<<","<<abs(dEnergy-.5*dPE)/-m_dE0<<","<<abs(dEnergy-m_dE0)/-m_dE0<<std::endl;
gettimeofday(&m_tStart, NULL);
m_bSplody=false;
}
void outputManager::close()
{
eOut.close();
noodleOut.close();
rOut.close();
binaryOut.close();
cmOut.close();
rvOut.close();
dtOut.close();
bIndexOut.close();
mcOut.close();
}
void outputManager::doMainScrenOut(star *stars[],double dRs[g_c_iStars][g_c_iStars], const double &dTime, const double &dStopTime)
{
double dTimeOut,dEnergy,dKE,dPE;
char sTimeOutTag;

gettimeofday(&m_tEnd, NULL);
dTimeOut=(m_tEnd.tv_sec+((m_tEnd.tv_usec/1000000.0))-(m_tStart.tv_sec+(m_tStart.tv_usec/1000000.0)))/(dTime/dStopTime)-
(m_tEnd.tv_sec+((m_tEnd.tv_usec/1000000.0))-(m_tStart.tv_sec+(m_tStart.tv_usec/1000000.0)));
//convert to better units
if(dTimeOut<60)//time best measured in seconds
{
sTimeOutTag='s';
}
else if(dTimeOut<3600)//time best measured in minutes
{
dTimeOut/=60.0;
sTimeOutTag='m';
}
else if(dTimeOut<86400)
{
dTimeOut/=3600.0;
sTimeOutTag='h';
}
else
{
dTimeOut/=86400.0;
sTimeOutTag='d';
}
```

```
dEnergy=energyNotVerlet(stars,dRs,g_c_iStars);
if(dEnergy>0&&!m_bSplody)
{
system("xmms -p /mnt/ROUTER/public/reaserch/sounds/splody.wav&");
m_bSplody=true;
}
dEnergy=energyNotVerlet(stars,dRs,g_c_iStars,dPE,dKE);
std::cout<<dTime/dStopTime<<" "<<dTimeOut<<sTimeOutTag<<" "<<abs(dEnergy-m_dE0)/-m_dE0<<std::endl;
}
void outputManager::doMainScrenOutClose(star *stars[],double dRs[g_c_iStars][g_c_iStars], const double &dTime, const double &dStopTime)
{
double dTimeOut,dEnergy,dKE,dPE;
char sTimeOutTag;

gettimeofday(&m_tEnd, NULL);
dTimeOut=(m_tEnd.tv_sec+((m_tEnd.tv_usec/1000000.0))-(m_tStart.tv_sec+(m_tStart.tv_usec/1000000.0)))/(dTime/dStopTime)-
(m_tEnd.tv_sec+((m_tEnd.tv_usec/1000000.0))-(m_tStart.tv_sec+(m_tStart.tv_usec/1000000.0)));
//convert to better units
if(dTimeOut<60)//time best measured in seconds
{
sTimeOutTag='s';
}
else if(dTimeOut<3600)//time best measured in minutes
{
dTimeOut/=60.0;
sTimeOutTag='m';
}
else if(dTimeOut<86400)
{
dTimeOut/=3600.0;
sTimeOutTag='h';
}
else
{
dTimeOut/=86400.0;
sTimeOutTag='d';
}
dEnergy=energyNotVerlet(stars,dRs,g_c_iStars);
if(dEnergy>0&&!m_bSplody)
{
system("xmms -p /mnt/ROUTER/public/reaserch/sounds/splody.wav&");
m_bSplody=true;
}
dEnergy=energyNotVerlet(stars,dRs,g_c_iStars,dPE,dKE);
std::cout<<dTime/dStopTime<<" "<<dTimeOut<<sTimeOutTag<<" "<<abs(dEnergy-m_dE0)/-m_dE0<<" "<<dRs[0][1]<<std::endl;
}
void outputManager::doMainScrenOut1(star *stars[],double dRs[g_c_iStars][g_c_iStars],
const double &dTime, const double &dStopTime,const double c_dDt)
{
double dTimeOut,dEnergy,dKE,dPE;
char sTimeOutTag;

gettimeofday(&m_tEnd, NULL);
dTimeOut=(m_tEnd.tv_sec+((m_tEnd.tv_usec/1000000.0))-(m_tStart.tv_sec+(m_tStart.tv_usec/1000000.0)))/(dTime/dStopTime)-
(m_tEnd.tv_sec+((m_tEnd.tv_usec/1000000.0))-(m_tStart.tv_sec+(m_tStart.tv_usec/1000000.0)));
//convert to better units
if(dTimeOut<60)//time best measured in seconds
{
sTimeOutTag='s';
}
else if(dTimeOut<3600)//time best measured in minutes
{
dTimeOut/=60.0;
sTimeOutTag='m';
}
else if(dTimeOut<86400)
{
dTimeOut/=3600.0;
sTimeOutTag='h';
}
else
{
dTimeOut/=86400.0;
sTimeOutTag='d';
}
dEnergy=energyNotVerlet(stars,dRs,g_c_iStars);
if(dEnergy>0&&!m_bSplody)
{
system("xmms -p /mnt/ROUTER/public/reaserch/sounds/splody.wav&");
m_bSplody=true;
}
dEnergy=energy(stars,dPE,dKE);
std::cout<<dTime/dStopTime<<" "<<dTimeOut<<sTimeOutTag<<" "<<abs(dEnergy-m_dE0)/-m_dE0<<" "<<c_dDt<<std::endl;
```

```
}
void outputManager::doFinalScreenOut()
{
double dTimeOut;
char sTimeOutTag;
std::ofstream timeOut;

//errorOut<<g_c_dDt*365<<","<<dMaxErrorE<<","<<dMaxErrorR<<std::endl;
gettimeofday(&m_tEnd, NULL);
dTimeOut=(m_tEnd.tv_sec+((m_tEnd.tv_usec/1000000.0))-(m_tStart.tv_sec+(m_tStart.tv_usec/1000000.0)));
//convert to better units
if(dTimeOut<60)//time best measured in seconds
{
sTimeOutTag='s';
}
else if(dTimeOut<3600)//time best measured in minutes
{
dTimeOut/=60.0;
sTimeOutTag='m';
}
else if(dTimeOut<86400)
{
dTimeOut/=3600.0;
sTimeOutTag='h';
}
else
{
dTimeOut/=86400.0;
sTimeOutTag='d';
}

std::cout<<"Time to complete = "<<dTimeOut<<sTimeOutTag<<std::endl;

timeOut.open("./codeIo/runtime.txt");
timeOut<<"Time to complete = "<<dTimeOut<<sTimeOutTag<<std::endl;
timeOut.close();
}
/*outputManager1_1.h
Written by Mike Williams 04-13-05
*/

#ifndef outputManager_h
#define outputManager_h

#include<fstream>
#include<sys/time.h>
#include"constants.h"
#include"star6_0.h"
#include"binarytracker.h"

// Vector3d Class
class outputManager
{
public:
void open();
void openOpt();
void doMainOut(star *stars[],const double &dTime, double dRs[g_c_iStars][g_c_iStars]);
void doMainOut1(star *stars[],const double &dTime, double dRs[g_c_iStars][g_c_iStars],const double c_dDt);
void doCmOut(star *stars[],const double &dTime);
void doDistroOut(star *stars[],const double &dTime,const double &dDt);
void doInitialOut(star *stars[],const double &dTime);
void doMainScrenOut(star *stars[],double dRs[g_c_iStars][g_c_iStars], const double &dTime, const double &dStopTime);
void doMainScrenOutClose(star *stars[],double dRs[g_c_iStars][g_c_iStars], const double &dTime, const double &dStopTime);
void doMainScrenOut1(star *stars[],double dRs[g_c_iStars][g_c_iStars], const double &dTime, const double &dStopTime,const double c_dDt);
void doFinalScreenOut();
void doMcNeilOut(star *stars[],const double dTime,int i, int j);
void doScreenOutOpt(star *stars[], const double &dE, const double &dTime, const double &dStopTime,const double c_dDt);
void doFileOutOpt(star *stars[],const double &dTime,double &dEnergy);
void doInitialOutOpt(star *stars[],const double &dTime);
void close();
void closeOpt();

private:
std::ofstream noodleOut,saveOut,eOut,rOut,binaryOut,cmOut,rvOut,abOut,dtOut,bIndexOut,mcOut,optOut,splodyOut;
double m_dE0,m_dAvBinSum,m_dLastT;
int m_iOldBinary;
timeval m_tStart, m_tEnd;
bool m_bSplody;
binaryTracker m_binTracker;
};

#endif
// rand.cpp
```

```
// System Includes
#include <stdlib.h>
#include "rand.h"

// Constants
static float g_fRandMaxReciprocal=1.0f/(float) RAND_MAX;
static float g_dRandMaxReciprocal=1.0/(double) RAND_MAX;

// Random (0 <= i <= RAND_MAX)
int Random()
{
return rand();
}

// Random (0 <= i <= nRange)
int Random(int nRange)
{
return rand()%(nRange+1);
}

// Random Range (nStart <= i <= nEnd)
int RandomRange(int nStart,int nEnd)
{
return ((nStart<nEnd)?(nStart+rand()%(nEnd-nStart+1)):(nEnd+rand()%(nStart-nEnd+1)));
}

// fRandom (0.0f <= f <= 1.0f)
float fRandom()
{
return g_fRandMaxReciprocal*rand();
}

// fRandom (0.0f <= f <= fRange)
float fRandom(float fRange)
{
return fRange*g_fRandMaxReciprocal*rand();
}

// fRandom (fStart <= f <= fEnd)
float fRandom(float fStart,float fEnd)
{
return fStart+(fEnd-fStart)*g_fRandMaxReciprocal*rand();
}

double dRandom(double dRange)
{
return dRange*g_dRandMaxReciprocal*rand();
}
//rand.h
//Random Integer functions
int Random();
int Random(int nRange);
int RandomRange(int nStart,int nEnd);

//Random Float Functions
float fRandom();
float fRandom(float fRange);
float fRandom(float fStart,float fEnd);

//Random double Functions
double dRandom(double dRange);
//rk2_1,cpp Mike Williams 2-05
#include"star6_0.h"
#include"constants.h"
#include"compare2_1.h"
#include"rk2_1.h"
#include<iostream>
#include<fstream>
#include"util.h"
#include"adaptFunctions.h"



void rk42StepSoft(star *stars[],star *starsp[],star *stars1[], double dRs[g_c_iStars][g_c_iStars],  const double &dDt,double &dDtNew,double &dEffort,
const double &c_dError0, const double &c_dRMin)//just calls rk4SoftNC in a 2 step way
{
double dErrorP;
copyStars(stars,starsp);

rk4SoftNC(stars,dRs,g_c_iStars,dDt,dEffort,c_dRMin);
copyStars(stars,stars1);//stor where we are after 1 whole step
copyStars(starsp,stars);//reset to initial conditions
```

```
for(int c=0; c<g_c_iStars;c++)
{
//reset the stars for the next step
stars[c]->m_vAcceleration.SetZero();
stars[c]->m_vTmpAcceleration.SetZero();
stars[c]->m_vTmpAcceleration2.SetZero();
stars[c]->m_vTmpAcceleration3.SetZero();
stars[c]->m_bClose=false;
}

rk4SoftNC(stars,dRs,g_c_iStars,dDt/2.0,dEffort,c_dRMin);
for(int c=0; c<g_c_iStars;c++)
{
//reset the stars for the next step
stars[c]->m_vAcceleration.SetZero();
stars[c]->m_vTmpAcceleration.SetZero();
stars[c]->m_vTmpAcceleration2.SetZero();
stars[c]->m_vTmpAcceleration3.SetZero();
stars[c]->m_bClose=false;
}

rk4SoftNC(stars,dRs,g_c_iStars,dDt/2.0,dEffort,c_dRMin);

dErrorP=errorPosition(stars1,stars);
//copyStars(stars1,stars);//really we want to use the time step suggested last time
dDtNew=dDt/pow(dErrorP/c_dError0,g_c_dAdaptPower);//set the dt for next time step
if(dDtNew<g_c_dDtMin)
{
dDtNew=g_c_dDtMin;
}
else if(dDtNew>g_c_dDtMax)
{
dDtNew=g_c_dDtMax;
}
}


void rk4SoftNC(star *stars[], double Rs[g_c_iStars][g_c_iStars], const int &n,
const double &dt, double &Effort,const double &c_dRMin)
{
double dR;
Vector3d vF;

//compute first r
for(int i = 0; i<n;i++)
{
for(int j =i+1; j<n; j++)
{
compareSoftSimple(stars[j],stars[i], vF,dR,c_dRMin);
stars[j]->m_vAcceleration+=vF/stars[j]->m_dMass;
stars[i]->m_vAcceleration-=vF/stars[i]->m_dMass;


Rs[i][j]=dR;
Effort++;

//system counting Calculations
//trackSystems.setTime(i,j,dR,dt);

}//end inner for
/*******************************************
Perform actual integration
*******************************************/
stars[i]->m_vPositionT=stars[i]->m_vPosition;

stars[i]->m_vPosition1=stars[i]->m_vVelocity*dt;
stars[i]->m_vVelocity1=stars[i]->m_vAcceleration*dt;

stars[i]->m_vPosition2 = (stars[i]->m_vVelocity + stars[i]->m_vVelocity1*0.5)*dt;
stars[i]->m_vPosition=stars[i]->m_vPositionT + stars[i]->m_vPosition1*0.5;
}//end outer for

for(int i = 0; i<n;i++)
{
for(int j =i+1; j<n; j++)
{
compareSoftSimple(stars[j],stars[i], vF,dR,c_dRMin);
//PE+=dPEtemp;
stars[j]->m_vTmpAcceleration+=vF/stars[j]->m_dMass;//use TmpAcceleration to save processor time of clearing acceleration
stars[i]->m_vTmpAcceleration-=vF/stars[i]->m_dMass;

Effort++;
}//end inner for
```

```
/*********************************************
Perform actual integration
*********************************************/
stars[i]->m_vVelocity2 = stars[i]->m_vTmpAcceleration*dt;
stars[i]->m_vPosition3 = (stars[i]->m_vVelocity + stars[i]->m_vVelocity2*0.5)*dt;

stars[i]->m_vPosition = stars[i]->m_vPositionT + stars[i]->m_vPosition2*0.5;
}//end outer for

for(int i = 0; i<n;i++)
{
for(int j =i+1; j<n; j++)
{
compareSoftSimple(stars[j],stars[i], vF,dR,c_dRMin);
//PE+=dPEtemp;
stars[j]->m_vTmpAcceleration2+=vF/stars[j]->m_dMass;
stars[i]->m_vTmpAcceleration2-=vF/stars[i]->m_dMass;

Effort++;

}//end inner for
/*********************************************
Perform actual integration
*********************************************/
stars[i]->m_vVelocity3=stars[i]->m_vTmpAcceleration2*dt;
stars[i]->m_vPosition4=(stars[i]->m_vVelocity + stars[i]->m_vVelocity3)*dt;

stars[i]->m_vPosition = stars[i]->m_vPositionT + stars[i]->m_vPosition3;

}//end outer for

for(int i = 0; i<n;i++)
{
for(int j =i+1; j<n; j++)
{
compareSoftSimple(stars[j],stars[i], vF,dR,c_dRMin);
//PE+=dPEtemp;
stars[j]->m_vTmpAcceleration3+=vF/stars[j]->m_dMass;
stars[i]->m_vTmpAcceleration3-=vF/stars[i]->m_dMass;

Effort++;
}//end inner for
/*********************************************
Perform actual integration
*********************************************/
stars[i]->m_vVelocity4=stars[i]->m_vTmpAcceleration3*dt;

stars[i]->m_vPosition=stars[i]->m_vPositionT + (stars[i]->m_vPosition1 +
(stars[i]->m_vPosition2 + stars[i]->m_vPosition3)*2.0 + stars[i]->m_vPosition4)/6.0;
stars[i]->m_vVelocity=stars[i]->m_vVelocity + (stars[i]->m_vVelocity1 +
(stars[i]->m_vVelocity2 + stars[i]->m_vVelocity3)*2.0 + stars[i]->m_vVelocity4)/6.0;

}//end outer for

}
//rk2_1,cpp Mike Williams 2-05
#ifndef _rk_h
#define _rk_h

#include"star6_0.h"
#include"constants.h"
#include"analytical.h"

void rk4SoftNC(star *stars[], double Rs[g_c_iStars][g_c_iStars], const int &n,  const double &dt, double &Effort, const double &c_dRMin);
void rk42StepSoft(star *stars[],star *starsp[],star *stars1[], double dRs[g_c_iStars][g_c_iStars],  const double &dDt,double &dDtNew,
double &dEffort,const double &c_dError0, const double &c_dRMin);//just calls rk4SoftNC in a 2 step way

#endif
/*star6_0.cpp
written by Mike Williams 10-05-04
uses scailed units
Tracks previous position for Verlet type integration
Implements Jerk for Hermete integration
Stors intermediate force and jerk for hermete integration
Stors 4 velocity and position vectors for RK4
*/
#include"star6_0.h"
#include "Vector3d.h"
#include "constants.h"
#include <math.h>

star::star()
{
```

```
}

star::~star()
{
//Null
}

void star::initialize(Vector3d p, Vector3d v,double m)
{
m_vPosition = p;
m_vVelocity = v;
m_dMass = m;
m_bClose=false;
}
/*star6_0.h
written by Mike Williams 04-1-05
basic point particle.
Uses Vector3d.h
A diferent aproach then previous versons. uses public variables and limited functions.
Tracks previous position for verlet integration
Tracks jerk for hermete integration
Tracks intermediate force and jerk for hermete integration
Stors 4 position and velocity vectors for rk4
keeps 5 old positions for legrange interpolating polynomials.
*/
#ifndef _star_h
#define _star_h

#include "Vector3d.h"

class star
{
public:
//Functions
star();
~star();
void initialize(Vector3d p, Vector3d v, double m);

//p=position of the star, v= velocity of the star, m=mass of the star,
//min=minimum size of radious squared for comparison to be calculated

//public data
Vector3d m_vPosition;
Vector3d m_vPosition1;
Vector3d m_vPosition2;
Vector3d m_vPosition3;
Vector3d m_vPosition4;
Vector3d m_vPositionT;
Vector3d m_vVelocity;
Vector3d m_vVelocity1;
Vector3d m_vVelocity2;
Vector3d m_vVelocity3;
Vector3d m_vVelocity4;
Vector3d m_vAcceleration;
Vector3d m_vTmpAcceleration;
Vector3d m_vTmpAcceleration2;
Vector3d m_vTmpAcceleration3;
double m_dMass;
bool m_bClose;
};

#endif
//util,cpp Mike Williams 2-05
#include"util.h"

double abs(double x)
{
if(x<0)
return -1*x;
else
return x;
}

double max(double a, double b)
{
if(a>b)
return a;
return b;
}

double min(double a, double b)
{
```

```
if(a<b)
return a;
return b;
}
//util,h Mike Williams 2-05
#ifndef _util_h
#define _util_h

double abs(double x);
double max(double a, double b);
double min(double a, double b);

#endif
```