Planetesimal Dynamics

by

Derek C. Richardson
Clare Hall
Institute of Astronomy

Dissertation submitted for the degree of
Doctor of Philosophy at the
University of Cambridge

September 1993

PREFACE

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration. Any sources from which information is derived are noted in the text and summarized in the References section. I declare that this dissertation is not substantially the same as any that I have submitted for a degree or diploma or other qualification at any other University. I further state that no part of my dissertation has already been or is being concurrently submitted for any such degree, diploma, or other qualification. This dissertation does not exceed 60 000 words in length (with the computer source code in Appendix B omitted).

The source listings for `box_tree` as well as all other subsidiary programs written for use with `box_tree` are available by request to the author (e-mail `dcr@mail.ast.cam.ac.uk`) or by anonymous `ftp` from `ftp.ast.cam.ac.uk` in the directory `/pub/dcr`. The source code in its entirety is Copyright © 1993 by Derek C. Richardson and is distributed under the terms of the GNU General Public License so that it may be freely used and modified. A copy of the License is included in the source distribution.

Derek C. Richardson
September 28, 1993

ACKNOWLEDGMENTS

I wish to thank the following people for their assistance during the course of my research project: my supervisor Dr. Sverre J. Aarseth, for his guidance, enthusiasm, and expertise; D. N. C. Lin, S. L. W. McMillan, and S. Tremaine for valuable ideas and suggestions; L. Hernquist, H. Salo, C. F. Gammie, and F. Governato for technical comments; and the staff and students at the Institute of Astronomy, in particular G. F. Lewis, P. Tribble, H. Ferguson, R. C. Thomson, and D. J. D. Earn. I am indebted to the Institute of Astronomy for providing excellent computing facilities for this project, and for providing support in many other ways as well. Lastly, I wish to thank the Association of Commonwealth Universities for giving me the opportunity to study abroad at the University of Cambridge.

This document was prepared using LaTeX. Figures were produced using `sm`, `xfig`, and `xv`. Source code listings were generated with `c++2latex`.

SUMMARY

A new tree code method for simulation of planetesimal dynamics is presented. A self-similarity argument is used to restrict the problem to a small patch of a ring of particles. The code incorporates a sliding box model with periodic boundary conditions and surrounding ghost particles. The tree is self-repairing and exploits the flattened nature of Keplerian discs to maximize efficiency. The code uses a fourth-order force polynomial integration algorithm with individual particle time-steps. Collisions and mergers, which play an important role in planetesimal evolution, are treated in a comprehensive manner. The collision equations include provisions for particle spin. In order to take advantage of facilities available, the code was written in C in a Unix workstation environment. The unique aspects of the code are discussed in detail and the results of a number of performance tests are presented. Timing tests show that the CPU time as a function of particle number varies in a way consistent with an $\mathcal{O}(N \log N)$ algorithm. The average relative force error incurred in typical runs is less than 0.2% in magnitude. Simulations of early solar system planetesimal evolution and the equilibrium properties of Saturn's B ring are discussed. With the most realistic simulations to date, it is found that particle aggregates and gravitational wakes readily form in Saturn's B ring, which may account in part for observed non-uniformities in Saturn's outer rings. Illustrations of the applicability of the code to other areas of research are given. Possible enhancements and extensions to the code are also discussed. Appendices containing a user manual and full source listings are provided. The source distribution, which includes supporting programs and macros, is available on request.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Dictionaries may variously define a “planetesimal” as a minute planetary body (from “planet”, meaning wanderer, and “infinitesimal”), or a small body that existed during an early stage in the development of the solar system. The first definition is preferred, as it is more general, but the more specific definition connected with the history of the solar system has come to be the one adopted by astronomers. This is unfortunate because it restricts the discussion of planetesimals to the past, yet present-day asteroids, comets, and meteoroids may be remnants of this primordial population. Indeed, if a planetesimal is an infinitesimally small wandering body (small, say, with respect to a typical terrestrial planet), then there is no reason why the particles that constitute planetary rings could not be called planetesimals as well. It is this definition of planetesimal, namely *any* small body in the solar system, that is used in this thesis. Moreover, it is the study of the dynamics of such bodies, modelled with a new computer code, that forms the broad subject of the research presented here. Although attention is focused mainly on the evolution of planetary rings and early planetesimals, much of the discussion may apply equally well to other planetesimal populations.

1.1 Historical Motivation

It has been recognized for some time that techniques used in numerical simulations of solar system formation can often be applied to studies of planetary ring dynamics, and vice versa (e.g. Ward 1984). The most important feature common to both systems is a flattened Keplerian disk which describes the mean motion of the constituent particles. These particles interact with each other, causing the system to evolve with time to some equilibrium state, or through a series of quasi-equilibrium states. Processes important to both regimes include energy dissipation due to collisions and velocity randomization due to gravitational interactions (Petit & Hénon 1987; Aarseth, Lin & Palmer 1993, hereafter ALP). Other mechanisms that play a role in these systems include resonance trapping, merging, fragmentation, and gas drag (Beaugé, Aarseth & Ferraz-Mello, in preparation), and, in the case of diffuse planetary rings, electromagnetic effects as well (e.g. Burns, Showalter & Morfill 1984).

Both analytical and numerical methods (and, in the case of planetary rings, direct observation) have been used to study planetesimal dynamics. In general, successive investigations employ more accurate or more efficient techniques, or include more of the key processes and fewer assumptions. Much of the groundwork for modern research into early solar system problems was laid down by Safronov (1969). Goldreich & Ward (1973) were able to show that initial gravitational instabilities in the primordial solar nebula could lead to the formation of kilometre-sized planetesimals. Early numerical work by Greenberg et al. (1978) used statistical methods based on heuristic scattering and coalescence cross sections to study the growth of planetesimals. Their method included the possibility of planetesimal fragmentation as well. A review of early numerical and theoretical work can be found in Wetherill (1980). Nakagawa, Hayashi & Nagazawa (1983) added gas drag to their numerical simulations, but still relied on a statistical method due to the prohibitive cost of direct force calculations. Wetherill & Stewart (1989) established a theoretical basis for the cross sections used in statistical methods, and found that runaway accretion can occur for certain configurations. Recent work by ALP used a direct method to show that conditions for runaway are favourable in the early solar system. Emori, Ida & Nakazawa (1993) have developed a new method that separates the dominant solar force from the planetesimal perturbations, reducing numerical errors and improving efficiency. Many other important contributions to this field have been made over the past years.

Work on planetary rings has followed a similar evolutionary path. Goldreich & Tremaine (1978) used an analytical model consisting of identical non-self-gravitating particles to establish the importance of collisions in governing the equilibrium velocity dispersion in Saturn's rings. An important review of post-Voyager observational data as well as analytical and numerical results to date can be found in Greenberg & Brahic (1984). Work by Wisdom & Tremaine (1988, hereafter WT) on planetary rings will be discussed in detail in this thesis. Recent numerical work by Araki (1991) used a kinetic theory of dense gases and included spin effects to determine various equilibrium properties of Saturn's rings. Still more recent work by Salo (1991; 1992a; 1992b) expanded on the WT model by incorporating interparticle gravity and initial size distributions.

Despite advances in our understanding of the processes governing planetesimal dynamics, it is acknowledged that analytical treatments still only give a very rough picture of the true behaviour. It is rapidly becoming more feasible to rely on a numerical approach to the problem, where at least some of the more fundamental complexities—gravitational interactions between particles, physical collisions, and particle spin—can be taken into account. However, in order to address the most central issues in planetesimal dynamics, numerical simulations of systems with large dynamical range and high spatial resolution are required (Salo 1991; Palmer, Lin & Aarseth 1993, hereafter PLA; ALP). This is the motivation for the numerical code presented in this thesis.

1.2 Numerical Simulations

In order to minimize the computational expense arising from the large number of particles needed to generate realistic planetesimal simulations, two important techniques have been devised. The first of these methods, hereafter termed "box code", was introduced by WT to simplify numerical studies of the Saturnian ring system. The box code is based on a self-similarity assumption that the dynamical interaction over the entire extent of a given ring of planetesimals may be studied by the detailed analysis of a local representative patch of the ring. Hence a small-N system can be used to model a large-N system.

The second, more widely-known technique is the "tree code", presented by Barnes & Hut (1986, hereafter BH) as an efficient tool for fast and reasonably accurate force calculations. Although the original motivation for the development of tree code stemmed from modelling galactic dynamics, the method has been implemented successfully in a variety of contexts. A simple angular size rule is used to decide whether to add force contributions collectively by cell through multipole expansions, or individually by particle through direct summation. For sufficiently large N, the tree code reduces the computational requirements of a standard N-body simulation from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$.

These components have been combined together with a standard N-body integrator (Aarseth 1985) to form a new code for planetesimal dynamics simulations, called `box_tree`. The purpose of the code, much like any other gravitational N-body simulator, is to integrate the equations of motion of a given set of particles from a given set of initial conditions until some user-specified time has elapsed. In the case of `box_tree`, events such as particle collisions and boundary crossings are resolved along the way. This aspect adds an extra layer of complexity over typical N-body codes, since the accurate prediction and resolution of such events may become as important as the integration itself. As far as the author is aware, `box_tree` is the first implementation of tree code for the planetesimal problem and incorporates most of the key features of previous codes (currently with the exception of fragmentation), providing the most realistic simulations to date.

The `box_tree` code has evolved considerably since its inception, both in form and efficiency. Originally, `box_tree` was designed to simulate conditions in the early solar system at 1 AU from the Sun using particles of size $\sim$ 10–100 km. Early results from this work have been published in Richardson (1993a). Later, the code was generalized to perform simulations of Saturn's B ring using centimetre- to metre-sized particles (Richardson 1993b). Much of the content of this thesis is taken from these publications. The `box_tree` code has been used in one form or another in other applications as well. As a result, `box_tree` has evolved into a much more general simulation package, capable of handling a variety of numerical tasks.

1.3 Thesis Layout

This thesis is divided into seven major chapters and two appendices. Chapter 2 is essentially a continuation of this introduction, describing the box code and tree code in greater depth. The technical details of the new scheme are given in Chapter 3. Various performance tests are presented and discussed in Chapter 4. The largest chapter, Chapter 5, contains detailed results of the major simulations performed using `box_tree`. Other applications and future directions for the code are presented in Chapter 6. The last chapter, Chapter 7, contains a summary and further comments. Appendix A is essentially a user manual for `box_tree`, but does make frequent reference to material discussed in the earlier chapters. Appendix B contains full source code listings for `box_tree`.

Chapter 2

Fundamentals

Before discussing the inner workings of the code, it is helpful to examine in more detail the two main components that have been fused together to form `box_tree`, without considering the actual implementation too closely. The box code (§2.1) provides the model for the planetesimal simulations, specifying a coordinate system, the linearized equations of motion, and a prescription for boundary conditions. The tree code (§2.2) provides the means for fast calculation of the interparticle forces that are added as perturbations to the equations of motion. Together these components form the basic tool for describing planetesimal dynamics.

2.1 The Box Code

2.1.1 Coordinate System

WT demonstrated that a typical planetary ring can be divided into self-similar patches or boxes which are dynamically independent, provided that the unit patch is larger than the radial mean free path and much smaller than the distance to the planet centre. Hence the dynamical evolution within a single patch can be used to represent the behaviour of the ring as a whole. A further simplification is obtained by referring patch particle coordinates to the centre of a comoving Cartesian coordinate system superimposed on the unit cell. The system follows a Keplerian orbit in the $z = 0$ plane with its y-axis always pointing in the direction of motion and its x-axis pointing radially away from the planet (Fig. 2.1). Under these conditions it is possible to linearize the equations of motion for the particles (§2.1.2). The WT method can be applied equally well to the case of planetesimals orbiting the Sun, although in this case patch sizes tend to be large, typically a few hundredths of the distance from the Sun. For Saturn's rings there is no danger of violating the model assumptions, since typical particle sizes are in the 1 cm–1 m range (compared to an average orbital distance of $\sim 10^8$ m) and the particles are densely packed (optical depth ~ 1), so the largest practical box sizes are only a few km on a side and the radial mean free path is small.

2.1.2 Linearized Equations of Motion

To derive the linearized equations of motion, it is necessary to transform the particle accelerations to the rotating frame. Let $\mathcal{R}_s$ denote the position a particle relative to the space frame origin (the heliocentre for example), as seen in the space frame. The relative position vector of the particle as seen in the rotating frame is the same at any instant, i.e.

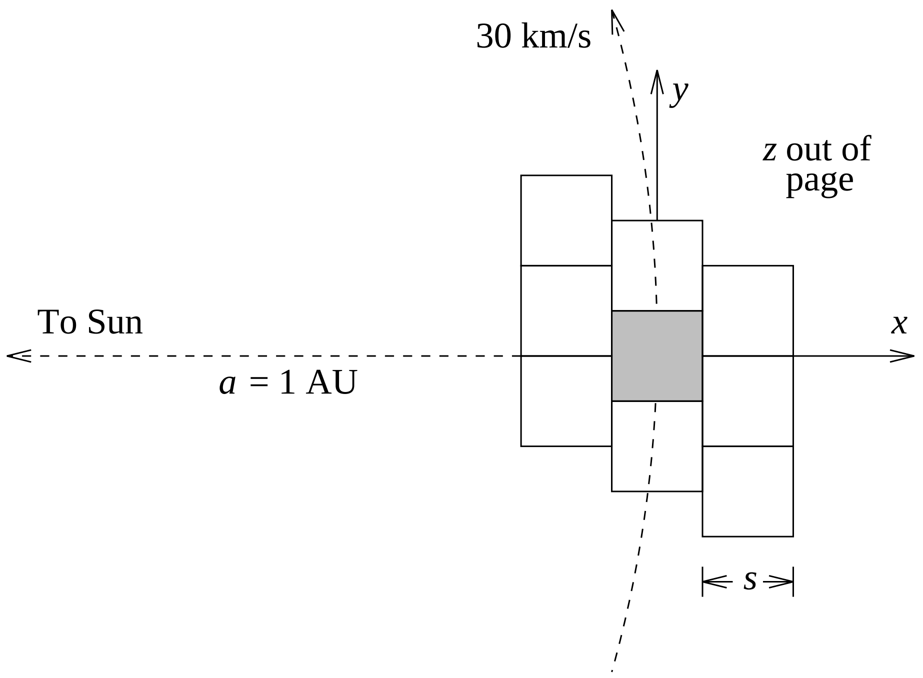


Figure 2.1: Rotating coordinate system used in `box_tree`. Note that $a \gg s \gg R$, where R is the average particle radius. The central box (shaded region) is surrounded by eight ghost boxes, some of which are shifted in the y-direction to illustrate Keplerian shear.

$\boldsymbol{\mathcal{R}}_r = \boldsymbol{\mathcal{R}}_s = \boldsymbol{\mathcal{R}}$. However, the particle experiences a net acceleration in the space frame given by:

$$\ddot{\boldsymbol{\mathcal{R}}}_s = -\frac{GM}{\mathcal{R}^3}\boldsymbol{\mathcal{R}} - \boldsymbol{\nabla}\phi, \tag{2.1}$$

where M is the mass of the central body (the Sun in this case), and $-\boldsymbol{\nabla}\phi$ is the sum of the contributions of all other particles in the orbiting patch. The acceleration will appear to be different in the rotating (non-inertial) frame. To derive $\ddot{\boldsymbol{\mathcal{R}}}_r$, it is convenient to apply an operator used by Goldstein (1980) which relates the time rate of change of a quantity in the space frame to its counterpart in the rotating frame:

$$\left(\frac{d}{dt}\right)_s = \left(\frac{d}{dt}\right)_r + \boldsymbol{\Omega}\times \tag{2.2}$$

where $\boldsymbol{\Omega}$ is the instantaneous angular velocity of the rotating frame. For the current model, $\boldsymbol{\Omega} = \Omega\hat{\boldsymbol{z}}$ is constant and has magnitude:

$$\Omega = \sqrt{\frac{GM}{a^3}}, \tag{2.3}$$

where a is the (constant) distance separating the origin of the rotating frame and the central body (say 1 AU). Note that in this model the origin of the rotating frame and the location of the rotation axis itself do not coincide. Apply the operator given by equation (2.2) to $\boldsymbol{\mathcal{R}}$:

$$\dot{\boldsymbol{\mathcal{R}}}_s = \dot{\boldsymbol{\mathcal{R}}}_r + \boldsymbol{\Omega}\times\boldsymbol{\mathcal{R}}. \tag{2.4}$$

Now apply the operator to $\dot{\boldsymbol{\mathcal{R}}}_s$:

$$\ddot{\boldsymbol{\mathcal{R}}}_s = \left(\frac{d}{dt}\dot{\boldsymbol{\mathcal{R}}}_s\right)_r + \boldsymbol{\Omega}\times\dot{\boldsymbol{\mathcal{R}}}_s. \tag{2.5}$$

Substitute equation (2.4) into equation (2.5) and rearrange to get:

$$\ddot{\boldsymbol{\mathcal{R}}}_r = \ddot{\boldsymbol{\mathcal{R}}}_s - 2\left(\boldsymbol{\Omega}\times\dot{\boldsymbol{\mathcal{R}}}_r\right) - \boldsymbol{\Omega}\times\left(\boldsymbol{\Omega}\times\boldsymbol{\mathcal{R}}\right), \tag{2.6}$$

which is the acceleration of the particle as seen in the rotating frame. It is not convenient, however, to use $\boldsymbol{\mathcal{R}}$ to denote particle positions. Instead, write $\boldsymbol{\mathcal{R}} = \boldsymbol{a} + \boldsymbol{r}$, where $\boldsymbol{a}$ is the position vector of the origin of the rotating frame and $\boldsymbol{r}$ is the particle position with respect to this origin. As seen from the rotating frame, the time derivative of $\boldsymbol{a}$ is zero, hence $\dot{\boldsymbol{\mathcal{R}}}_r = \dot{\boldsymbol{r}}$ and $\ddot{\boldsymbol{\mathcal{R}}}_r = \ddot{\boldsymbol{r}}$, so equation (2.6) becomes:

$$\ddot{\boldsymbol{r}} = -2\left(\boldsymbol{\Omega}\times\dot{\boldsymbol{r}}\right) - \boldsymbol{\Omega}\times\left[\boldsymbol{\Omega}\times\left(\boldsymbol{a}+\boldsymbol{r}\right)\right] - \frac{GM\left(\boldsymbol{a}+\boldsymbol{r}\right)}{\left|\boldsymbol{a}+\boldsymbol{r}\right|^3} - \boldsymbol{\nabla}\phi,$$

where $\ddot{\boldsymbol{\mathcal{R}}}_s$ has been substituted using equation (2.1). Consider the x-component of the expression for $\ddot{\boldsymbol{r}}$:

$$\ddot{x} = 2\Omega\dot{y} - \Omega^2\left(a+x\right) - \frac{GM\left(a+x\right)}{\left|\boldsymbol{a}+\boldsymbol{r}\right|^3} - \frac{\partial\phi}{\partial x}.$$

The denominator in the third term on the right-hand side can be written:

$$\begin{aligned}\left|\boldsymbol{a}+\boldsymbol{r}\right|^{-3} &= a^{-3}\left(1+\frac{2x}{a}+\frac{r^2}{a^2}\right)^{-\frac{3}{2}} \\ &\simeq a^{-3}\left(1-\frac{3x}{a}\right),\end{aligned}$$

where, by virtue of x being much smaller than a, all terms in x/a of second order and higher have been dropped. From equation (2.3), $a^{-3} = \Omega^2/GM$, hence:

$$\begin{aligned}\ddot{x} &\simeq 2\Omega\dot{y} + \frac{3x}{a}\left(a+x\right)\Omega^2 - \frac{\partial\phi}{\partial x} \\ &\simeq 2\Omega\dot{y} + 3\Omega^2 x - \frac{\partial\phi}{\partial x}.\end{aligned}$$

The y- and z-components follow in similar fashion to give the complete set of linearized equations of motion:

$$\begin{aligned}\ddot{x} &= \mathcal{F}_x + 3\Omega^2 x + 2\Omega\dot{y}, \\ \ddot{y} &= \mathcal{F}_y - 2\Omega\dot{x}, \\ \ddot{z} &= \mathcal{F}_z - \Omega^2 z,\end{aligned} \tag{2.7}$$

where $\boldsymbol{\mathcal{F}} = -\boldsymbol{\nabla}\phi$ is the sum of the gravitational forces per unit mass due to the other ring particles.

2.1.3 Boundary Conditions

The particles in a unit patch of a ring each have a mean Keplerian velocity that is a function of the particle orbit radius plus some dispersion about the mean due to heating effects. This mean velocity will generally differ from the orbital velocity of the reference frame (e.g. $\Omega a \sim 30$ km/s at 1 AU from the Sun), being faster for particles closer to the origin of the central force and slower for those further away. In the rotating frame this phenomenon manifests itself as a uniform shear across the patch. As a result, particles inside the central box will quickly leave the system unless some kind of periodic boundary conditions are imposed. However, if the model assumptions are satisfied, the unit patch is only one of many making up the ring, each with the same relative particle distribution as the central box. Thus as a particle leaves the central box, it is replaced by another particle entering from a neighbouring patch. In this sense, periodic boundary conditions are a natural consequence of the patch model.

There are two other consequences of the patch model that must be considered before quantifying the boundary conditions. First, since the particles have physical size, collisions between particles in neighbouring boxes must be allowed for to prevent overlapping. Second, and equally important, gravitational perturbations due to particles in neighbouring patches must be taken into account, otherwise the box particles would have a tendency to squeeze together toward the centre. The most consistent way to deal with these problems, and at the same time provide a convenient means of handling actual boundary crossings, is to include contributions from the eight neighbouring patches (or "ghost boxes") of the central box in the simulation. It is only necessary to keep track of the boxes, not the ghost particles themselves, because the relative placement of particles in each ghost box is the same as in the central box.

The positions and velocities of the ghost boxes can be determined by noting that, in the absence of interparticle gravitational forces, equation (2.7) is invariant under the transformation:

$$(x, y, z) \to (x + \delta x, y + \delta y, z), \quad (\dot{x}, \dot{y}, \dot{z}) \to \left(\dot{x}, \dot{y} - \frac{3}{2}\Omega\,\delta x, \dot{z}\right), \tag{2.8}$$

where δx and δy are arbitrary (this justifies the assertion that the relative particle positions in each patch can be the same). Thus if the central box has sides of length s, the eight nearest ghost boxes can be chosen with centres at $(i_x s, -\frac{3}{2} i_x s \Omega t + i_y s, 0)$, where $i_x, i_y \in \{-1, 0, +1\}$ (not both zero). Boxes with $i_x = \pm 1$ experience a transverse shear of magnitude $\frac{3}{2}\Omega s$ due to the differential rotation of the ring (Fig. 2.1). Similarly, each particle in the central box experiences a shear of magnitude $\frac{3}{2}\Omega x$. Note that as t increases, it is necessary to subtract multiples of s from the shearing distance of the ghost boxes to ensure that they remain close to the central box.

In summary, if a particle leaves the central box, it is replaced by its corresponding image entering *from* a ghost box. Note that a particle leaving in the $\pm x$-direction may also undergo a change in y due to the shear of the ghost boxes. Such a particle also experiences a jump in its angular momentum (see §4.4.2 for a discussion). There are no boundary conditions in the z-direction. Perturbations on central particles are calculated by summing over contributions due to the $N - 1$ other central particles and the $8N$ ghost particles (§2.3). Collisions on the boundary are detected during the force calculation by tagging the closest particle, which may be a ghost (§3.5.2).

2.2 The Tree Code

The CPU time required to run a typical N-body simulation using a standard direct method scales as $\mathcal{O}(N^2)$ since, in repeated intervals typically much shorter than the dynamical time, $\mathcal{O}(\mathcal{N})$ force calculations must be performed, each involving a sum over contributions from $\mathcal{O}(\mathcal{N})$ particles (ignoring ghosts). In the present context, reasonably large N ($\sim 10^2$–10^4) is required to examine the evolution of systems with large dynamic range, making numerical computation with a standard direct method unrealistic. BH put forward a hierarchical algorithm that reduces the expense to $\mathcal{O}(N \log N)$ for sufficiently large N, but which introduces a small error in the calculated force, on average $\sim 1\%$ (somewhat larger than the intrinsic error in direct summation techniques). The idea is to place particles in a tree-like hierarchy of boxes or cells and replace the direct force with a multipole expansion about the centre of mass of those cells small enough or far

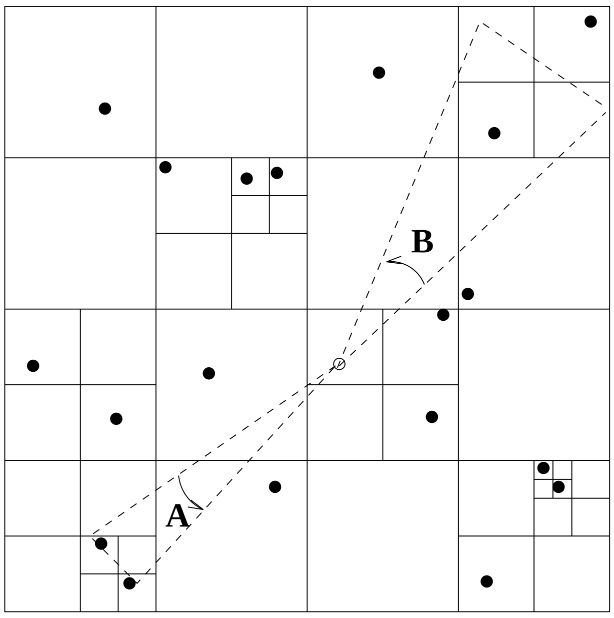


Figure 2.2: Twenty particles in a 2D tree. Two possible expansions are shown for one particle (open circle): in case (A), the opening angle may be small enough for a multipole expansion, but in case (B) the angle is probably too large so the force contribution of the two particles in the upper-right would be added individually.

enough away from the test point. To load the tree, particles are placed into a cell large enough to accommodate the entire system: if two particles occupy the same cell, the cell is subdivided into 2^n equal-sized boxes, where n is the tree dimension, until the particles occupy separate cells (Fig. 2.2). Note that other forms of tree code exist, such as the binary or mutually nearest neighbour tree (see Hernquist 1987 for a review), but the quadrant/octant BH form is best suited to the box model used here.

2.2.1 Components of the Tree

Before explaining the details of how the force is calculated, it is helpful to define some of the terms that describe the components of the tree. A cell that contains more than one particle (and therefore at least four sub-boxes in 2D) is called a *branch* or *node*, and in some contexts an *ancestor* or *parent*. Parents have *children* or *siblings* or *daughters* or *descendants*, which may be *leaves* in the case of isolated particles, or smaller branches (sub-boxes with more than one particle). The largest cell is the *root* branch and is the ancestor of everything in the tree. The root cell has no parent. Finally, a tree is often divided into *generations* or *levels*: the zeroth level consists of the root cell, its immediate children form the first level, and so on. The maximum number of nodes on level ℓ of an n-dimensional tree is given by $2^{n\ell}$.

2.2.2 Multipole Expansions

To calculate the force at a point, each branch is considered in turn, starting with the root: if the angle $\theta = s_{\rm cell}/r$ subtended by the cell at its centre of mass is smaller than a specified opening-angle parameter (usually $\theta_{\rm C} \lesssim 1$ rad), a multipole expansion is performed about the centre of mass of the cell (see Fig. 2.2). Otherwise, its children are examined: if a leaf is found, its force contribution is calculated directly; if a branch is found, the procedure continues recursively with a calculation of the angle subtended by the descendant cell, then its child branches if necessary, and so on. The number of poles used in the expansion determines the accuracy of the force approximation (see §4.3). The monopole term (where all the particles in an expanded cell are replaced by one large particle at the centre of mass), though easy to calculate, is generally an insufficient approximation of the force. With the expansion being about the centre of mass, the dipole term vanishes, leaving the more complex quadrupole term as the next contribution in the series. Most implementations stop at the quadrupole, but a few include the octupole (e.g. McMillan & Aarseth 1993, hereafter MA). With just the monopole and quadrupole components, the force per unit mass at a position $\boldsymbol{r}$ relative to the cell's centre of mass is given by (e.g. Marion & Heald 1980):

$$\boldsymbol{\mathcal{F}} = -\frac{M}{r^3}\boldsymbol{r} + \frac{\mathbf{Q}\cdot\boldsymbol{r}}{r^5} - \frac{5}{2}\frac{(\boldsymbol{r}\cdot\mathbf{Q}\cdot\boldsymbol{r})\boldsymbol{r}}{r^7}, \tag{2.9}$$

where the gravitational constant G has been defined such that $GM^\star \equiv 1$, where $M^\star$ is the mass unit (see §5.1). The quantity M in equation (2.9) is the total mass of the cell particles (in mass units), and $\mathbf{Q}$ is the quadrupole moment tensor of the cell given by:

$$Q_{jk} = \sum_i m_i(3x_{i,j}x_{i,k} - r_i^2\delta_{jk}), \tag{2.10}$$

where $\boldsymbol{r}_i = (x_{i,1}, x_{i,2}, x_{i,3})$ is the position of particle i relative to the cell's centre of mass. Note that $\mathbf{Q}$ is symmetric, and, in 3D, traceless, so that only $2n - 1$ elements need be stored in memory for each matrix. Hernquist (1987) gives a useful recursion relation for calculating the quadrupole moment of a cell from the quadrupole moments of its children (cf. Shift Theorem):

$$\mathbf{Q} = \sum_i^{N_{\text{cells}}} \mathbf{Q}_i + \sum_i^{N_{\text{cells}}} m_i\left[3\boldsymbol{r}_i'\boldsymbol{r}_i' - (r_i')^2\mathbf{I}\right], \tag{2.11}$$

where $\boldsymbol{r}_i' = \boldsymbol{r}_{g,i} - \boldsymbol{r}_g$ is the displacement vector between the centre of mass of sub-box i and the centre of mass of the parent cell, and $\mathbf{I}$ is the unit matrix.

BH suggested that due to its highly recursive nature, their version of the tree code was well suited to an implementation in C. Other versions, where the recursion has been "unwound", have been programmed in vectorized form—usually in FORTRAN—for improved performance on supercomputers (e.g. Hernquist 1990, Makino 1990; also see §6.4.4). Tree code in one form or another is now used widely and has proved to be a very successful method for approximating $1/r^2$ interaction laws, especially in collisionless systems where close encounters need not be represented accurately.

2.3 The `box_tree` Code

Since "boxes" are a fundamental concept in both box code and tree code, it seems natural to combine the techniques to provide a fast method for simulating planetesimal evolution. The best configuration is obtained by constructing a tree for the central box alone, which can then be mapped without modification onto each ghost box since the relative position of any two particles is preserved under equation (2.8). The total perturbation on a particle is obtained by summing contributions from particles in the central box and each ghost box in turn according to the standard tree code algorithm described above (§2.2). Note that the force due to a particle's own ghosts is included in the summation, since it is too costly to remove the contribution from the appropriate cell moments in the case of a force expansion. However, ghost particles are distributed symmetrically around the central particle, so the net force from a particle's own ghosts is zero anyway.

These ideas were put together and the resulting code is a `C` program called `box_tree`, developed in a Unix workstation environment. A number of special and in some cases unique considerations went into developing the code. All of these aspects are presented in detail in the following chapter, along with a general description of the more basic components of the code.

Chapter 3

Code Details

The technical details of `box_tree` are presented in this chapter. An overview of the logical structure of the program is given first to provide a framework for the more detailed discussion. This is followed by a short description of the layout and use of the major data structures. The rest of the chapter is devoted to describing certain features of the code in detail, namely the options for initial conditions, the special tree considerations, and particle collision handling. Figure 3.1 shows a schematic outline of the main integration loop for reference.

This chapter is not intended to be a user manual for `box_tree`. Many details pertaining to the actual running of the code have been omitted. Instead, the focus here is on the major aspects that make the code unique as an N-body simulator. Extensive information on how to compile, run, and modify `box_tree` including a complete source listing, can be found in the Appendix.

3.1 Overview

The `box_tree` integrator is based on a standard fourth-order polynomial N-body code with individual time-steps (Aarseth 1985). Particles are given initial positions and velocities at time $t = 0$ and the force and first three derivatives acting on the particles are calculated explicitly. Each particle is assigned a time-step Δt_i according to a user-specified formula (§3.5.1). Divided differences are introduced by converting from the Taylor series derivatives. The times $t_i + \Delta t_i$ (where t_i is the last update time, initially zero) are sorted chronologically into a list and integration proceeds one particle at a time starting at the top of the list. When it is time for a particle to be updated, its position and velocity are first predicted to high order ($\mathcal{F}^{(3)}$) using the stored derivatives. Next, the force acting on the particle in its new position is calculated by predicting the positions of all other particles to low order ($\mathcal{F}^{(1)}$) and summing the contributions to the force (including ghosts). Then new derivatives/differences are calculated and a fourth-order semi-iteration is performed to further improve the accuracy of the position and velocity of the current

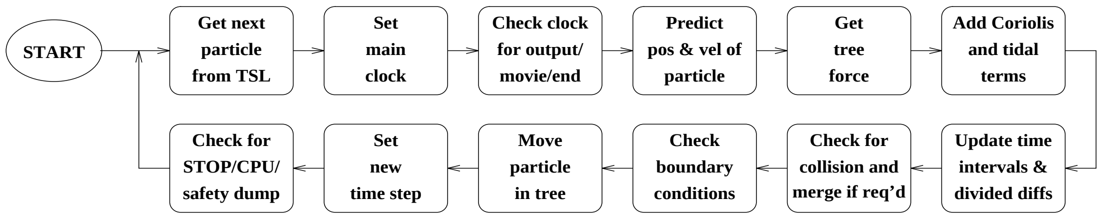

START
Get next particle from TSL
Set main clock
Check clock for output/ movie/end
Predict pos & vel of particle
Get tree force
Add Coriolis and tidal terms
Update time intervals & divided diffs
Check for collision and merge if req'd
Check boundary conditions
Move particle in tree
Set new time step
Check for STOP/CPU/ safety dump

Figure 3.1: A schematic of the program flow in the main `box_tree` integration routine.

particle. After a new time-step has been determined, the particle is placed back on the time-step list (TSL), and integration continues with the next particle.

Many modifications to this scheme had to be made in order to incorporate the box code and tree code. However, with the exception of the force calculation, these modifications consist largely of statements inserted into the main integration routine, without replacing major parts of the existing algorithm. The most important changes are: (1) construction of the initial tree at time $t = 0$, including a calculation of all the node moments; (2) replacement of the force calculation with a "tree walk" over nodes in the central and ghost boxes by performing multipole expansions over nodes that are sufficiently small or far away, or adding contributions from individual particles; (3) addition of Coriolis and tidal terms to the force [cf. equation (2.7)]; (4) checks for collisions once the new position has been determined; (5) application of boundary conditions if the particle has moved outside the central box; and (6) repair of the tree to account for the change in position of the particle.

Double precision is used throughout `box_tree` to minimize roundoff errors and to conform with standard C math function prototypes. The tree expansion is taken to quadrupole order as a compromise between speed and accuracy (see §4.3). The initial conditions and other program parameters, including termination time and output control, are supplied through a parameter file by the user at run time. Several timers are used to keep track of output schedules. Safety dumps of all variables are performed regularly, and give identical results on restarts within machine precision. The program can be halted cleanly and simply at any time by creating a special file in the run directory. These periodic checks are made at various points inside the main integration loop, which otherwise continues uninterrupted.

3.2 Program and Data Structures

During the development of `box_tree`, an attempt was made to exploit the features of C to the fullest extent possible. Such features include dynamic memory allocation, pointer references, data structures, recursion, preprocessor macros, and so on. Further, code dealing exclusively with tree management was kept as separate as possible from the integration code, both to lend a more logical structure to the program and to aid in debugging.

Each particle in `box_tree` is described by a structure containing data such as the particle mass, radius, position, spin, time-step, tree node, etc., amounting to 440 bytes of information per particle. Memory for these structures is allocated dynamically at run time, and deallocated when particles undergo mergers. Tree nodes have similar structures to store sizes and positions, child information, multipole moments and their derivatives, and various indices and flags (520 bytes per node). These structures are created and destroyed quite frequently as a result of tree repair. There are many other useful global structures, including one for all the program clocks and timers, one for storing closest-particle information, one for the TSL data, and a large structure for storing most of the run parameters. Structures are a convenient means of grouping similar data in a logical and easily interpreted fashion. In the case of the particle data, an array of pointers to the structures is kept globally, so that, for example, the current position of particle i is kept in `Data[i]->pos`. For the tree nodes, only the root address is stored globally; all other nodes are accessed from the children of the root. For example, the position of the first child branch of the root is `Root->child[0].branch->pos` (recall that array indexing in C begins at 0, not 1 as in FORTRAN).

The pointer implementation of node structures allows recursive tree routines to be constructed very easily. The following C function illustrates the principle:

```
void GetNumLeaves(branch, num_leaves)
BRANCH_T *branch;
int *num_leaves;
{
    int i;

    for (i = 0; i < MAX_NUM_CHILDREN; i++)
        switch (branch→child_type[i]) {
            LEAF:
                ++(*num_leaves);
                break;
            BRANCH:
                GetNumLeaves(branch→child[i].branch, num_leaves);
        }
}
```

This routine returns the total number of particles contained in a node and its subnodes (`MAX_NUM_CHILDREN` is defined as 2^n). If `branch` points to the root node, then after the call `num_leaves` will contain the total number of (central) particles in the simulation (note `num_leaves` must be zeroed prior to the first call). This sort of recursive program structure is used repeatedly in `box_tree` and has proved to be very efficient.

Where appropriate, preprocessor macros (via the `#define` directive) have been used to improve both the readability and efficiency of the code. Simple mathematical functions, "fuzzy" comparisons of machine precision limited numbers, logical flags, and some short core functions have been coded in this way. Most `box_tree` options can be configured at run time through the use of a special parameter file. Many options can be changed on restart so that program flow can be altered during the course of a long run. Most of the parameter data are stored in a large global structure for easy access anywhere within the program.

In all, `box_tree` consists of 167 routines (of which 79 are global), grouped into 17 files. There are three header files containing fixed parameters, function declarations, type definitions, and macro definitions, and a "makefile" for compilation on Unix platforms. The source, including extensive comments, is roughly 370 kB in size (just over 12 000 lines). The size of the symbol-stripped executable is just over 200 kB when compiled and optimized using "`gcc -O2`" on a Sparc 10/50 running `SunOS 4.1.3`.

3.3 Initial Conditions

For ease of use, `box_tree` has several built-in algorithms for generating initial conditions. There is also an option to read in external data, so that in principle any initial conditions can be accommodated (e.g. §6.3). Currently only built-in functions for populating a patch of a ring are implemented. The most important of these are described in Chapter 5 in the context of the simulations in which they have been used. Generally, given the number of particles N, the box size, an initial velocity dispersion, and a mass range, these functions generate initial particle positions and velocities, usually adjusted so that there is no net momentum and the centre of mass is at the origin. Initial bound pairs (within a fixed distance) can optionally be rejected. Generation of the initial mass distribution is described in the following section.

3.3.1 Initial Mass Function

The initial mass function (IMF) used to generate particle masses is derived from:

$$n(m) \propto m^{\alpha},$$

where $n(m)\,dm$ is the number of objects with mass in the range $[m, m + dm]$, and α is a dimensionless parameter. Let $N(m)$ be the cumulative distribution, such that $dN/dm = n(m)$. Integrating between $m_{\min}$ and m gives:

$$N(m) = \frac{m_{\min}^{\alpha+1}}{\alpha+1}\left[\left(\frac{m}{m_{\min}}\right)^{\alpha+1} - 1\right], \ \alpha \neq -1.$$

The total number of particles N is simply $N(m_{\max})$. Let $f = N(m)/N$ and solve for m:

$$m = m_{\min}\left\{1 + f\left[\left(\frac{m_{\max}}{m_{\min}}\right)^{\alpha+1} - 1\right]\right\}^{\frac{1}{\alpha+1}}. \tag{3.1}$$

This equation is equivalent to:

$$m = \left[(1-f)m_{\min}^{\alpha+1} + f m_{\max}^{\alpha+1}\right]^{\frac{1}{\alpha+1}},$$

which is less computationally efficient but somewhat more intuitive. Masses are chosen by replacing f with uniform deviates between 0 and 1, or, for a smooth distribution, with values varied monotonically from 0 to 1 in steps of N^{-1}.

To obtain a size (radius) distribution in place of a mass distribution, write:

$$dN \propto R^{\alpha^{\star}}dR.$$

Assuming constant particle density ($m \propto R^3$), a substitution of variables gives:

$$dN \propto m^{\frac{\alpha^{\star}-2}{3}}dm.$$

Hence the correct size distribution can be obtained by setting:

$$\alpha = \frac{\alpha^{\star}-2}{3}$$

in equation (3.1).

For large N, global properties of the mass distribution (e.g. total mass, mean mass, mean radius, etc.) can be well approximated by integrating appropriate powers of m in equation (3.1). For example, the mean mass can be estimated by setting $f = x/N$ and integrating the IMF in the range $[0, N]$ in x to give:

$$\overline{m} \approx \left(\frac{\alpha+1}{\alpha+2}\right)\left(\frac{m_{\max}^{\alpha+2} - m_{\min}^{\alpha+2}}{m_{\max}^{\alpha+1} - m_{\min}^{\alpha+1}}\right), \ \alpha \neq -1, -2.$$

For N as low as 50, this estimate is still good to about 10%.

3.4 Tree Considerations

There are a number of difficulties that arise from attempting to impose tree code on the planetesimal problem. These problems and their solutions are discussed in this section.

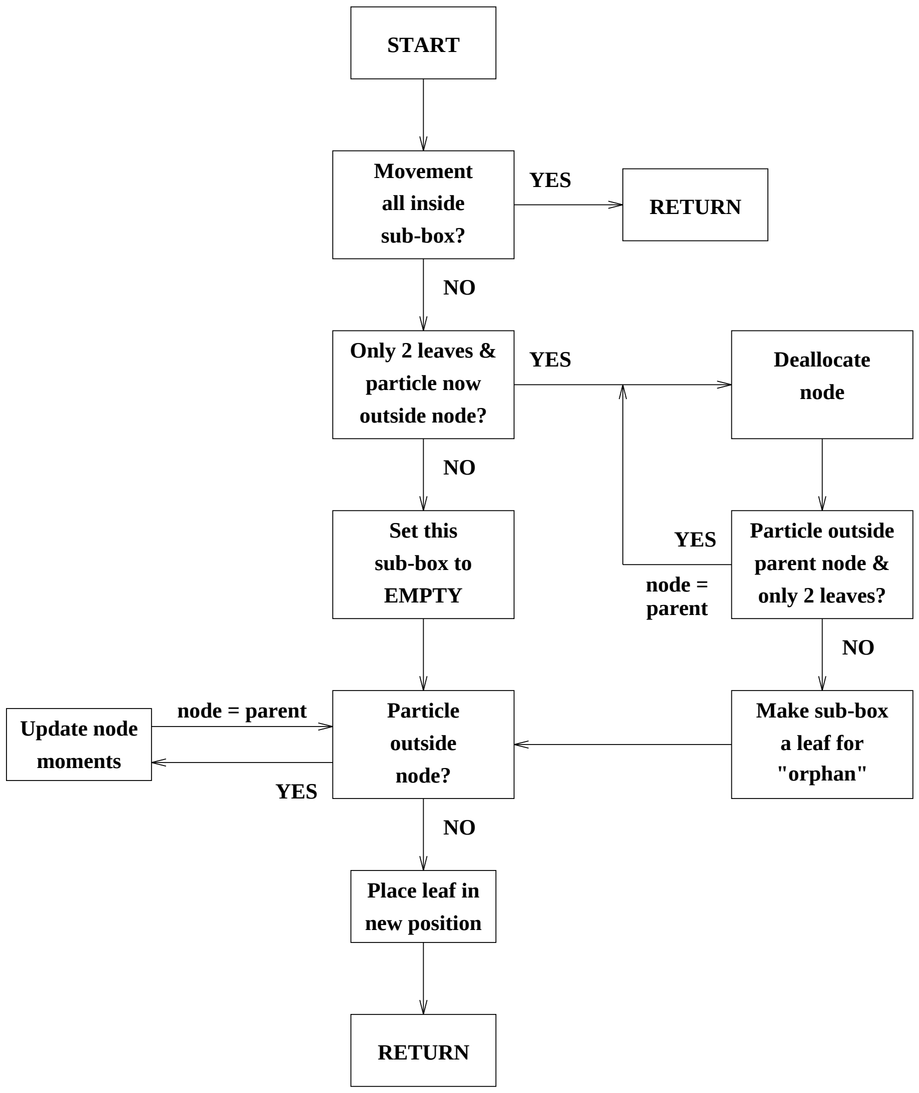

START
Movement all inside sub-box?
YES
RETURN
NO
Only 2 leaves & particle now outside node?
YES
Deallocate node
NO
Set this sub-box to EMPTY
YES
Particle outside parent node & only 2 leaves?
node = parent
NO
Update node moments
node = parent
Particle outside node?
YES
Make sub-box a leaf for "orphan"
NO
Place leaf in new position
RETURN

Figure 3.2: Schematic representation of the tree repair algorithm.

3.4.1 Tree Repair

Since any N-body system is dynamic by its very nature, it is clear that a static tree will cease to represent the correct mass distribution after a fairly small number of time-steps. BH, who employed a common step for all particles, suggested that the tree should be rebuilt after every time-step (since this is a fairly fast procedure for collisionless systems), while other authors assign a time-step to the entire tree (or parts of the tree using hierarchical or "block" steps, e.g. MA) based on the minimum of various timescales, such as the minimum cell crossing time. However, these methods are unsuitable for the current project: rebuilding the tree every time-step is far too expensive since a high collision frequency leads to very short time-steps; but assigning an effective tree-step is difficult because particle velocities vary widely across the central box as a result of the Keplerian shear. Ideally, the tree should only be updated in places where it would actually change as a result of particle motion from step to step. Although *every* particle moves between steps, it is sufficient to consider only the single particle being integrated at each step, since the other particles are only predicted to low order (see §3.4.2 and §3.4.5, however).

Figure 3.2 is a schematic representation of the tree repair algorithm used in `box_tree`. When a particle is to be moved in the tree, a check is made first to see whether the new position is still within its original sub-box (one of the 2^n boxes of its parent). If so, no

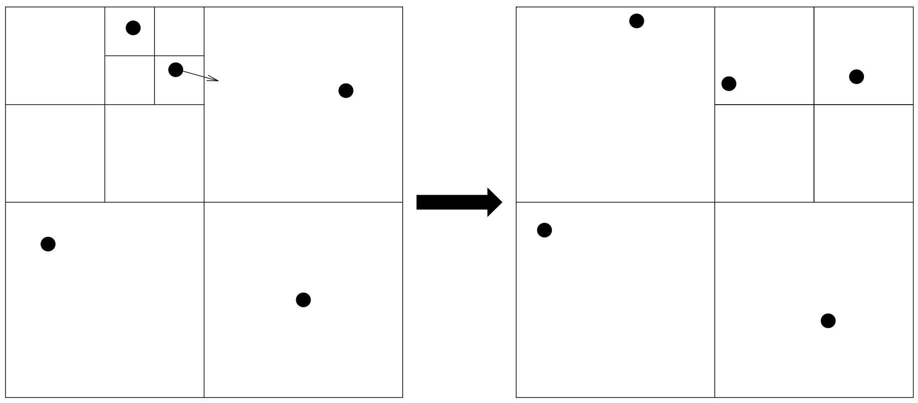


Figure 3.3: An example of tree repair. Here two nodes are destroyed and one is created.

repair is performed since the tree structure will not change. Otherwise, there are three possibilities: (1) the particle is moving between sub-boxes of its parent; (2) the particle is leaving the current node and at least two particles will be left behind; or, (3) the particle is leaving the node but only *one* particle will be left behind. In cases (1) and (2), the sub-box previously occupied by the particle is vacated; no further structural changes are needed since the original node remains intact. In the third case, however, when there is only one particle left behind (call it an "orphan"), the node and its parents must be destroyed until either (a) the new position of the first particle is contained within an ancestor of the current node, or (b) a companion can be found for the orphan in an ancestral node, whichever comes first. When such a node is found, it becomes the current node and the orphan is made into a leaf in the appropriate sub-box of the node. At this point in all cases, the current node and its ancestors are updated (§3.4.2) until the motion of the first particle is contained entirely within a node, or until the root node is reached. Once this is accomplished, the particle can be placed in its new position using the standard tree construction algorithm, which will often result in the creation of new nodes. Figure 3.3 is a pictorial example of a section of a 2D tree before and after repair.

Though somewhat difficult to describe, tree repair is fairly straightforward to implement, especially in its recursive form. Tree repair eliminates the need for full tree reconstruction at regular intervals, and is well suited to situations when only certain parts of the tree are undergoing rapid changes at any given time. Moreover, tree repair is extremely CPU cost-effective, taking less than 1% of the total computation time during a typical run (see §4.2).

3.4.2 Node Updates and Prediction

When particles are inserted into or removed from the tree (or, by extension, moved within it), branches may be created or destroyed. This means that the multipole moments of each affected branch must be recalculated so that the correct expansion may be formed when determining the interparticle gravitational forces. If the system were static, inserting and removing leaves from branches would be a simple matter of adding and subtracting the corresponding contributions to the monopole and quadrupole moments of the parent branches. Unfortunately, the system is *not* static, as was seen in the previous section. Moreover, after $t = 0$, particles are updated on different time scales so that some form of prediction of all the descendant leaf positions would be needed to bring the moments back up to date. But since updates are performed only during tree repair, the moments would be valid in any case only at each update time, and would become progressively worse approximations until the next update was performed. Subsequent force errors would be noticeably large and discontinuous depending on the time between updates and the importance of the particular force contribution.

The obvious solution to both these problems—inefficient updates and large, discontinuous force errors—is to introduce node prediction (e.g. MA). An effective treatment requires calculation of the node position and velocity (i.e. the centre-of-mass position and velocity of the node), the force acting on the centre of mass and its first time derivative, as well as the quadrupole moment and its first three time derivatives, at each update. The former components, those associated with the monopole, are straightforward to calculate, since they are simply the mass-weighted sum of the corresponding components of the node children. These quantities (position, velocity, etc.) are already known for the leaves, and can be determined recursively for any child branches. The quadrupole derivatives must be calculated explicitly, however, but are still subject to the recursive property of equation (2.11). For completeness, the quadrupole tensor derivatives are given by:

$$\begin{aligned}
\dot{Q}_{jk} &= 3(\dot{x}_j x_k + x_j \dot{x}_k) - 2\delta_{jk} \textstyle\sum_{k'} x_{k'} \dot{x}_{k'}, \\
\ddot{Q}_{jk} &= 3(\ddot{x}_j x_k + \dot{x}_j \dot{x}_k + x_j \ddot{x}_k) - \delta_{jk}(\dot{r}^2 + 2 \textstyle\sum_{k'} x_{k'} \ddot{x}_{k'}), \\
\dddot{Q}_{jk} &= 3(\dddot{x}_j x_k + \ddot{x}_j \dot{x}_k + \dot{x}_j \ddot{x}_k + x_j \dddot{x}_k) - 2\delta_{jk} \textstyle\sum_{k'} (x_{k'} \dddot{x}_{k'} + \dot{x}_{k'} \ddot{x}_{k'}),
\end{aligned} \tag{3.2}$$

where $\boldsymbol{r} = (x_1, x_2, x_3)$ is the position relative to the node's centre of mass, as before. Note that $\ddot{x}$ and $\dddot{x}$ have been divided by 2 and 6 respectively, and that the summations over m_i have been omitted for clarity.

When calculating forces, then, the *predicted* centre-of-mass position of each node is used to determine whether or not a multipole expansion should be performed. In the case of an expansion, both the monopole and predicted quadrupole are used to obtain the force contribution. Further, for updates of a given node following tree repair, only the immediate descendants of the node need be considered: both child leaves and branches are predicted to low order, and their contributions to the monopole and quadrupole moments of the node are added in explicitly. These refinements reduce the average force errors considerably, though at the cost of a noticeable but bearable increase in computation time. In fact, with node prediction in place, force errors are dominated by the approximate nature of the expansion itself, so that any improvement would require introducing the octupole moment (see §4.3).

Note that it is possible for nodes (and particles for that matter) to have predicted positions that lie outside the box system, which could lead to noticeable asymmetries in the force distribution depending on the velocities, time-steps, and masses of the nodes or particles. To minimize this effect, a "tree wrap" is performed after predicting node or particle positions in the force routines. That is, appropriate multiples of three box lengths are either added to or subtracted from y-positions found to lie outside of the system prior to measurement of the node or particle distances. Corrections are not performed in the x-direction since the radial mean free path is expected to be small at all times. Note that this simple wrap routine is not applied to a central particle being advanced at a given time-step, but rather to all other particles whose positions and velocities have been predicted to low order for the purpose of calculating the force on the central particle. For the particle being advanced, a proper boundary condition treatment is performed after the new force has been calculated (§2.1.3).

It should also be noted that, as with any predicted quantity, there is a time interval over which the prediction can be considered reliable and after which the prediction can no longer be used. MA have put forward recipes for assigning time-steps to the monopole and quadrupole moments of a node, based on standard time-step formulae (see discussion in Press & Spergel 1988). For the monopole:

$$\Delta t_{\rm M} = \epsilon_{\rm M} \left(\frac{s_{\rm cell} \sum_k |\mathcal{F}_k| + \dot{x}^2}{\sum_k |\dot{x}_k| \sum_k \left|\dot{\mathcal{F}}_k\right| + \mathcal{F}^2} \right)^{\frac{1}{2}}, \tag{3.3}$$

and for the quadrupole:

$$\Delta t_{\rm Q} = \epsilon_{\rm Q} \left(\frac{\sum_i |Q_i| \sum_i \left|\ddot{Q}_i\right| + \sum_i \dot{Q}_i^2}{\sum_i \left|\dot{Q}_i\right| \sum_i \left|\dddot{Q}_i\right| + \sum_i \ddot{Q}_i^2} \right)^{\frac{1}{2}}, \tag{3.4}$$

where $\epsilon_{\rm M}$ and $\epsilon_{\rm Q}$ are constants, and the index i in the second equation is taken over the $2n-1$ unique matrix elements. A check is made to ensure the time-steps do not grow too large too quickly. When performing or checking for expansions in the tree force routines, the current time is compared with the last update of the current node plus its appropriate time-step. If the node needs updating, this is done immediately. In general, it is the larger branches that need more frequent updating, since particles take a long time to cross the bigger boxes, while smaller nodes are repeatedly being created and destroyed as a result of tree repair. The appropriate choice of $\epsilon_{\rm M}$ and $\epsilon_{\rm Q}$ is discussed in §4.3.

There are further complications that result from node prediction, but discussion of these will be deferred to §3.4.5.

3.4.3 Boundary Conditions

Since the box and tree overlap perfectly, when a particle crosses the central box boundary it also crosses the tree boundary. At the same time, the particle's ghost enters the tree from the opposite side. Fortunately these events are easily handled by the tree repair routines (§3.4.1). In this case the particle's node and *all* its ancestors, including the root node, must be updated. For technical reasons, however, the repair algorithm as described will actually miss updating the moments of the root node following a boundary crossing because the new particle position is automatically offset by a box length before the tree repair routine is called. Thus, as far as the repair algorithm is concerned, the particle never left the tree but was merely displaced within it by a large distance. The solution is to pass a flag from the main integration routine to the tree repair routine indicating whether or not a particle boundary crossing occurred and the root node should be updated.

3.4.4 Problems in 2D

Since planetesimals are more or less confined to a plane, it seems inefficient to use a 3D tree to represent the system. However, it is important to study planetesimal dynamics in the direction normal to the plane in order to, for example, measure the evolution of the thickness of the disk. The additional degree of freedom also prolongs the evolution process since the collision timescale increases by at least an order of magnitude (PLA). Hence for an efficient but accurate treatment it is necessary to impose a 2D tree on a 3D system. A few unique problems arise from this configuration, but their solutions are straightforward enough to justify the unconventional approach.

The major difficulty with 2D trees is that expansions tend to be performed more often than they should. This is because particles projected onto the xy-plane seem to be closer together and may therefore be placed in boxes that are smaller than they would be in the 3D case. The solution to this problem is conveniently part of the solution to a more general problem that stems from node prediction, and will be discussed in §3.4.5 below.

A less crucial but nonetheless important problem can occur in the relatively rare case when two or more particles almost line up perpendicular to the xy-plane. As the particles line up, more and more subdivisions of the tree are required to separate their projections. In theory, two particles could overlap perfectly in this way, resulting in an infinite number of subdivisions. In practice this situation has never arisen, but particle projections have overlapped enough to cause problems. The first difficulty is that machine precision fundamentally limits the number of times a quantity can be accurately divided by 2. The second more restrictive problem is that nodes in `box_tree` are each tagged with a unique index given by:

$$i_{\texttt{node}} = (i_{\texttt{node->parent}})2^n + j + 1, \qquad (3.5)$$

where $i_{\texttt{Root}} \equiv 0$ and j $(0 \le j < 2^n)$ is the position of the node inside its parent (j can also index leaf positions in a node). In the case $n = 2$, the bottom left sub-box is $j = 0$, bottom right $j = 1$, top left $j = 2$, top right $j = 3$. The index is primarily used for book-keeping purposes and has proved useful when debugging. Unfortunately, on a typical workstation, $i_{\texttt{node}}$ must be less than 2^{31} as a signed integer ($i_{\texttt{node}} = -1$ is used for initialization), which for $n = 2$ means ℓ, the tree level, must always be 15 or less. In fact, $\ell > 14$ is quite rare, so this is a reasonable restriction. But these unusual configurations do occur nonetheless, so a scheme called "node packing" has been introduced. If ℓ is found to exceed (`int`)$(31/n)$ for a particular node, then the leaves involved are packed into the first available locations in the node, regardless of their exact physical position. That is, the particles are given arbitrary j values, which presupposes that not more than 2^n particles will overlap at the same time (such an over-packed situation has yet to occur in a 2D tree). Packing is possible because the exact location of a particle in its node is used only at the beginning of the tree repair algorithm (see §3.4.1 or Fig. 3.2). So for a packed node, the first test of the repair routine—whether a particle has moved between sub-boxes—is automatically forced to fail. No other changes to the code are required. The flag for a packed node remains set until the node is destroyed, which is rarely more than a few time-steps after it was created. Packing essentially means that there is a minimum node size (given by $s_{\texttt{Root}}/2^{\ell_{\max}}$, where $s_{\texttt{Root}}$ is the tree size), but for expansion purposes it is highly probable that the effective size (see §3.4.5 below) will always be greater, so expansions will not be affected at all.

It should be noted that in 3D the restriction on ℓ is much more severe (a maximum of 10 levels can be accommodated). For a simulation with large N ($N \gtrsim 10^3$) and particularly dense regions, many nodes may become packed and overflow. For this reason, node packing is disabled in 3D and the tree indices must be assumed to be unreliable. Fortunately this has no effect on the simulation itself.

3.4.5 Stretchable Nodes

Perhaps the greatest difficulty that results from not reconstructing the entire tree every time-step is that particles may move outside their own nodes between repairs, despite the fact that both node and particle positions are predicted. Consequently, force errors may result that noticeably exceed the expected intrinsic error of a finite-term multipole expansion (§4.3). Generally this problem is most acute in the y-direction, where local shear relative to each node centre would tend to make the node twist and stretch diagonally if it were flexible. Unfortunately, such parallelograms are not well suited to a tree structure! Instead, a compromise has been made by introducing an "effective size" for each node that is recalculated during node updates: the node is allowed to stretch equally in all directions to form a larger square box that accommodates its children at the time of update. It must be stressed that this effective size is used *only* when deciding on expansions in the force routines, i.e. the opening angle is redefined as $\theta = s_{\mathrm{eff}}/r$. Naturally, s_{eff} can be made smaller by reducing the node update time-step coefficients ϵ_{M} and ϵ_{Q} [cf. equations (3.3) & (3.4)]. However, it has been found (§4.3) that the coefficients can retain reasonably large values and still give good results.

Particles can *still* wander outside their nodes, however, since the effective size remains fixed between node updates. Usually this is not a problem if updates are sufficiently frequent, but, in extreme cases, an expansion may be performed over a node to which the particle itself actually belongs, causing a potentially severe force error due to self-gravity (severe because both the node sizes and the separations tend to be small in this case). This problem can be overcome by recursively checking all the descendants of a node before performing an expansion. However, to do this in all cases results in a very noticeable increase in CPU time. It has been found that this check need only be performed when a particle is within $0.01 s_{\mathtt{Root}}$ of the node. Note, however, that this method is reliable only if $\theta_{\mathrm{C}} \lesssim n^{-(1/2)}$, which can easily be shown to guarantee elimination of the self-gravity problem in the static case.

As mentioned in §3.4.4, imposing a 2D tree on a 3D system gives rise to some troublesome projection effects. Fortunately, a simple generalization of the definition of s_{eff} can be made to allow for comparatively large z-separations of particles in a node. In particular, the effective size can be defined as the maximum of the actual size of the node and the predicted y- and z-extensions of each child from the centre of mass. Note that both leaves and branches of the node are considered, since the predicted branch positions are subject to the same effects as the leaf positions. The "extension" of a leaf in y or z is simply the corresponding projected distance between the leaf and the parent centre of mass, but for a branch the extension is the sum of the projected separation between centres of mass and half the maximum extension of the branch, to allow for extended and/or offset sub-boxes. Hence updates should be performed starting from the bottom of the tree hierarchy to ensure that child branches have the correct effective size. Fortunately, both the tree repair algorithm and the update routines inherently work from the bottom up (§3.4.1 and §3.4.2), so this requirement adds no additional computational burden. Note that deviations in the x-direction are not considered since any radial excursions are expected to be small.

3.5 Collision Handling

Collisions are a fundamental aspect of planetesimal evolution, providing a mechanism for energy dissipation to balance heating derived from the shearing flow. Because of the dynamical importance of collisions, it is crucial to have an accurate collision detection and resolution algorithm. But due to the inherent complexity of an N-body gravitational system, it is generally impossible to predict in advance exactly when a collision between two particles will occur. This means that in order to detect collisions reliably, particle time-steps must be sensitive to the proximity of potential colliders. However, the level of sensitivity must be tempered by practical considerations: an infinitely precise algorithm would take an infinite amount of time. As a result, collisions are generally not detected the instant they occur, but rather *after* some mutual penetration of the colliding bodies has taken place. The goal is to minimize the penetration distance prior to collision without incurring too heavy a computational burden. The method developed for `box_tree` is designed to handle regimes at very high optical depth and is therefore relatively complex, but it has proved to be quite effective. All of the elements that go into this method will be presented in this section.

3.5.1 Time-step Formulae

If interparticle gravity is not treated in the simulation (as for the WT models), the following formula is used to calculate the optimal time-step of a particle given the position and velocity of its nearest *approaching* neighbour:

$$\Delta t = \eta\sqrt{\frac{r^2 - \varepsilon^2(R_1 + R_2)^2}{v^2}}, \tag{3.6}$$

where η is the dimensionless time-step coefficient, r is the magnitude of the relative position $\boldsymbol{r} = \boldsymbol{r}_2 - \boldsymbol{r}_1$ measured between the centres of the particle and its neighbour, $R_1 + R_2$ is the sum of the particle radii, and v is the magnitude of the relative velocity $\boldsymbol{v} = \boldsymbol{v}_2 - \boldsymbol{v}_1$. An approaching neighbour is defined as one for which $\boldsymbol{r}\cdot\boldsymbol{v} < 0$ (particle trajectories are generally well-behaved when there is no interparticle gravity, so approaching neighbours are the most likely colliders). The parameter ε determines what fraction of the finite size of the particles is included. If $\varepsilon = 0$, the sizes are ignored and the equation reduces to $\varepsilon(r/v)$. If $\varepsilon = 1$, the "true" particle separation is used, measured from surface to surface along the line connecting the particle centres. Since the latter case can result in very short time-steps, the computation time may increase significantly when there are many close encounters and collisions. The CPU dependence turns out to be more or less exponential for ε near unity. In fact, $\varepsilon = 0.99$ is typically twice as fast as $\varepsilon = 1$ but still gives very good accuracy.

Equation (3.6) is inadequate for models that include interparticle gravity. Consider, for example, a very close encounter in which the particle trajectories undergo considerable distortion. The above formula may give good results during the approach, but immediately after the encounter the second particle will be ignored in favour of a different approaching particle when determining the new time-step. This neglects the strong gravitational effects that will still be present after closest approach. Hence for the full gravity case, the $\boldsymbol{r}\cdot\boldsymbol{v} < 0$ criterion is no longer safe. Further, the time-steps will generally be too large immediately following a collision, since the quantity $\boldsymbol{r}\cdot\boldsymbol{v}$ may change sign in just a few time-steps due to the mutual gravitational attraction. In short, the time derivatives of the total force acting on a particle must be taken into account when choosing a time-step to allow for complicated interactions with close neighbours. The formula adopted in `box_tree` is based on a similar expression used by Aarseth (1985) [also compare with equations (3.3) and (3.4) in §3.4.2]:

$$\Delta t = \left(\eta\frac{\sum_k |\mathcal{F}_k| \sum_k \left|\ddot{\mathcal{F}}_k\right| + \dot{\mathcal{F}}^2}{\sum_k \left|\dot{\mathcal{F}}_k\right| \sum_k \left|\dddot{\mathcal{F}}_k\right| + \ddot{\mathcal{F}}^2}\right)^{\frac{1}{2}}. \tag{3.7}$$

Here, unlike in Aarseth (1985), the absolute value of the components of the force and its derivatives are used to calculate "magnitudes", eliminating two square roots at the expense of a slightly less sensitive formula. To conform with Aarseth (1985), the time-step coefficient η has been placed inside the square root in this expression. Note that information regarding the closest neighbour is only implicit in this formula, contributing the larger part of the force derivatives. Identification of the closest neighbour is only used for collision determination (§3.5.2). This formula is expensive to compute, especially if `box_tree` first needs to convert from divided differences to the Taylor series terms (which is usually the case), but the benefits gained from increased sensitivity far outweigh such considerations.

To save time, equation (3.6) could be used when the nearest neighbour is approaching and equation (3.7) when it is receding. But since it is short-range interactions that are of crucial importance in high-density collisional simulations, usually equation (3.7) is favoured for these models. It should be noted that the time-step coefficients η in equations (3.6) and (3.7) need not be the same, but, in the form they are given here, η values in the range 0.002–0.02 generally give good results.

3.5.2 Collision Detection

Potential colliders are identified as part of the force calculation procedure in order to minimize the number of predictions and distance measurements to be performed. Conveniently, the tree code automatically eliminates particles that are too far away to be likely colliders when it applies the opening-angle criterion to its nodes (§2.2). The closest particle (with $\boldsymbol{r}\cdot\boldsymbol{v} < 0$ if there is no interparticle gravity) is noted, along with its position and velocity predicted to first order and appropriately adjusted if the neighbour is a ghost. After the fourth-order semi-iteration correction to the current particle position and velocity, a check is made to see whether a collision may have occurred by comparing the distance r with the sum of the radii $R_1 + R_2$. If $r < R_1 + R_2$ (i.e. there is some overlap), and if interparticle gravity is included in the simulation, a check is made to ensure that the particles are indeed approaching one another, and are not left over from a previous collision.

If the pair satisfies these conditions, then a collision is probable. The position and velocity of the collider is predicted to high (third) order, adjusted for ghosts, and the sum of radii and $\boldsymbol{r}\cdot\boldsymbol{v}$ checks are repeated. If the particles are no longer found to be colliding, a "near miss" message is generated and the collision is not recognized. Otherwise, a position correction is performed to adjust the particles so that they are just touching (§3.5.3), and $\boldsymbol{r}\cdot\boldsymbol{v}$ is checked for the last time. Though unlikely, it is possible that position correction prevents the particles from actually colliding. Otherwise `box_tree` proceeds to calculate the post-collision velocities.

Though tedious, these tests are needed to ensure correct behaviour under close-packed conditions.

3.5.3 Position Corrections

Ideally, a collision should be detected the instant it occurs. Unfortunately, such precision is impractical so a certain amount of "penetration" or temporary overlap is unavoidable. For low collision rates, errors introduced by such penetrations can be ignored, but when there are many collisions, especially between the same particles, two major problems arise: (1) angular momentum is not conserved because the collision equations (§3.5.6) assume the particles are just touching; and (2) self-gravitating particles may "collide" again after their first bounce, *before they have completely separated.* The latter problem is potentially disastrous because under extreme conditions the particles involved will simply "sink" into one another. A naïve solution to this problem would be to switch off the gravitational attraction between particles with $r < R_1 + R_2$, simulating a surface normal force. Unfortunately, a third particle could simply bounce into either of the first two before they drift apart, resulting in another sinking problem. Introducing normal forces between *all* touching particles in an arbitrary aggregate, and including perhaps a restoring force at the surface to simulate "stickiness" (e.g. Watanabe, in preparation), is beyond the scope of the current project.

The most straightforward solution, which also addresses the problem of angular momentum conservation, is to simply displace the particles so that they are just touching before applying the collision equations. There are two ways of accomplishing this: (1) moving the particles outward along the line connecting their centres; or (2) tracing the particles back along their respective velocity vectors. The latter solution has the advantage, for small displacements, of reproducing the "true" geometry just prior to the collision. However, to be consistent, the particles should really be advanced forward in time after resolving the collision, but this introduces far too many complications to be practical. Hence the post-collision positions and velocities will still be slightly inaccurate with this method. A more serious problem is that for grazing collisions (which are quite frequent in a shearing disk), the displacements can become arbitrarily large and unrealistic.

The first method, displacing the colliders along the line connecting their centres, ensures a minimum displacement, namely half the penetration depth. Since the depth of penetration can be controlled somewhat by the choice of time-step coefficient (§3.5.1), this is a desirable property. The disadvantage is that the collision geometry is altered slightly, and it is possible (though unlikely) that the particles involved may no longer be colliding after the displacement. However, given the simplicity of the technique, and the fact that it effectively eliminates the sinking effect for self-gravitating particles, this method has been implemented in the code. In equation form, the new particle positions are given by:

$$\boldsymbol{r}_1^{\mathrm{new}} = \boldsymbol{r}_1 - \left(\frac{\Delta r}{r}\right)\boldsymbol{r},\ \boldsymbol{r}_2^{\mathrm{new}} = \boldsymbol{r}_2 + \left(\frac{\Delta r}{r}\right)\boldsymbol{r},$$

where

$$\Delta r \equiv \frac{R_1 + R_2 - r}{2}.$$

A further refinement would be to weight the offsets according to particle mass, to minimize the effects of offsets on the larger particles. Note that $\varepsilon = 1$ can no longer be used in equation (3.6) with these corrections in place, as it would result in zero time-steps following collisions.

For completeness, a simple procedure for backtracking along particle velocity vectors is also given here. Write:

$$\boldsymbol{r}_i^{\mathrm{new}} = \boldsymbol{r}_i + \lambda_i \boldsymbol{v}_i \Delta t \quad (i = 1, 2),$$

where, to allow for particles moving in roughly the same or opposite directions,

$$\lambda_1 = \begin{cases} -1, & \boldsymbol{r}\cdot\boldsymbol{v}_1 < 0 \\ 1, & \boldsymbol{r}\cdot\boldsymbol{v}_1 > 0 \end{cases},\ \lambda_2 = \begin{cases} -1, & \boldsymbol{r}\cdot\boldsymbol{v}_2 > 0 \\ 1, & \boldsymbol{r}\cdot\boldsymbol{v}_2 < 0 \end{cases}.$$

To force the particles to just touch, set:

$$(\boldsymbol{r}_2^{\mathrm{new}} - \boldsymbol{r}_1^{\mathrm{new}})^2 = (R_1 + R_2)^2.$$

Now solve the quadratic to obtain Δt, rejecting the root with the larger absolute value (both roots should be negative):

$$\Delta t = \frac{-\boldsymbol{r}\cdot\boldsymbol{v}^\star \pm \sqrt{(\boldsymbol{r}\cdot\boldsymbol{v}^\star)^2 - |\boldsymbol{v}^\star|^2\left[r^2 - (R_1 + R_2)^2\right]}}{|\boldsymbol{v}^\star|^2},$$

where $\boldsymbol{v}^\star \equiv k_2\boldsymbol{v}_2 - k_1\boldsymbol{v}_1$. This procedure is evidently more complicated than the first, but may be more suitable for isolated collisional systems that are not subjected to a strong tidal field.

3.5.4 Velocity Corrections

The position corrections described in the previous section have the unfortunate side-effect of changing the total energy of the system for self-gravitating particles. If the total energy is being monitored as a performance check (§4.4.3), it is necessary to allow for the change in gravitational potential energy that results from applying position corrections. Conversely, the requirement that the total energy be conserved (allowing for heat dissipation from inelastic collisions) can be used to adjust the velocity of the colliders prior to applying the collision equations (§3.5.6). Currently `box_tree` reduces the radial (normal) component of the velocities of both particles by a factor γ ($0 < \gamma < 1$) to compensate for the increase in gravitational potential energy V of the system. The value of γ is given by:

$$\gamma = \sqrt{1 - \frac{\Delta V}{T_n}}, \tag{3.8}$$

where ΔV is the change in gravitational potential energy, and T_n is the kinetic energy arising from the normal component of the velocities of the two particles. The γ correction is not applied if $\gamma < 0.9$ (because the velocity correction is considered too large) or if $\gamma > 1.0$ (because this implies that V actually decreased, which can happen if one of the two particles was pushed into a third particle as a result of the position correction — see §3.5.5). In these cases, a correction is applied to the total energy itself, rather than to the velocities.

It turns out (§4.4.3) that V is not something easily measured in the shearing box approximation. Instead, the change ΔV for a two-body collision is approximated by:

$$\Delta V \approx m_1 m_2 \left(\frac{1}{r} - \frac{1}{R_1 + R_2} \right), \tag{3.9}$$

where r is the separation prior to the position correction. This is the correction that would be applied in an inertial frame that contained only the two colliders. The approximation is good enough to ensure that anomalous heating is minimized in close-packed systems, even though it's not eliminated altogether.

There is another small velocity correction that must be applied in the rotating frame: if the particles are displaced in the x-direction as part of the position adjustment, their mean orbital velocity must be changed. This is accomplished by subtracting $\frac{3}{2}\Omega\,\Delta x_i$ from the y-velocities of the two particles, where Δx_i is the change in x-position of particle i.

3.5.5 Missed Collisions

Currently `box_tree` is only capable of handling two-body collisions. Though these are by far the most common type of collision in N-body dynamics, under extreme conditions it is quite possible for two or more particles to collide with and hence overlap the same particle during the same time-step. The position correction described in §3.5.3 can worsen the situation, by adjusting the current particle or its original collider into new positions that overlap other particles. During reinitialization following a collision (§3.6), a check is made to see whether the new closest neighbour has indeed penetrated the current particle. If so, the current particle is assigned a very small step to ensure that it will be updated immediately, forcing collision resolution. In this way it is possible for many particles to bunch together quite tightly in a self-consistent manner. To do any better would probably require testing for multiple colliders at the outset, which for the current project would be an unnecessary added complication (especially since the assumption of spherical particles, for instance, is only an approximation in the models considered here).

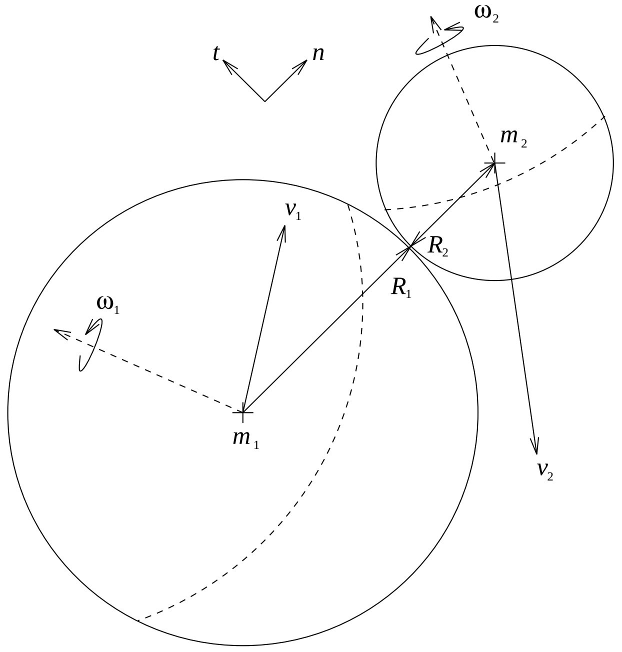


Figure 3.4: Diagram illustrating the basic collision definitions.

For regimes of moderately high particle density, it was found that setting $\Delta t = 10^{-15}t$ as the new time-step worked quite well, where t is the current simulation time. A relative value is favoured over an absolute value, as the former is not subject to precision limitations. However, at very high densities, there is a danger of entering a nearly infinite loop in the form of a repeated series of position adjustments and missed collision corrections to the same particles (picture three particles in a straight line, all touching). To avoid this, more moderate values of Δt should be used, based on the time-step and update time of the missed collider. Thus if a repeated series of corrections begins to take place, it will not be too long before another particle disturbs the cycle. A reasonable formula is:

$$\Delta t = 0.01(t_0 + \Delta t_0 - t), \tag{3.10}$$

where t_0 and Δt_0 are the last update time and time-step of the missed collider, respectively. The factor of 0.01 was determined empirically: the smaller the value, the closer the equation approaches the original formulation.

Note that it is possible that $\Delta t = 0$ in equation (3.10). This could happen if the missed collider was already due to be considered during the current time-step. Zero time-steps are not allowed in `box_tree`, so it is necessary to define a strict minimum step, which is taken as $10^{-15}t$. Since the missed collider was due for immediate consideration anyway, this is not a concern.

3.5.6 Collision Resolution

Once a collision event has been firmly established, the post-collision velocities (both linear and angular) of the colliders must be determined. The following derivation is appropriate for rough spheres of arbitrary mass.

Consider two uniform colliding spheres with masses m_1 and m_2, radii R_1 and R_2, located at $\boldsymbol{r}_1$, $\boldsymbol{r}_2$ in a Cartesian space, with linear velocities $\boldsymbol{v}_1$, $\boldsymbol{v}_2$ and angular velocities (spins) $\boldsymbol{\omega}_1$, $\boldsymbol{\omega}_2$ (Fig. 3.4). Let $\boldsymbol{r} = \boldsymbol{r}_2 - \boldsymbol{r}_1$ and $\boldsymbol{v} = \boldsymbol{v}_2 - \boldsymbol{v}_1$ be the relative position and velocity of the spheres, respectively, and define a new coordinate system in the collision plane with orthogonal axes n (normal component) and t (transverse component). The normal component is directed along the line connecting the centres of the two spheres ($\hat{\boldsymbol{n}} \equiv \boldsymbol{r}/r$). Also define vectors connecting the sphere centres to the point of impact: $\boldsymbol{R}_1 = R_1\hat{\boldsymbol{n}}$, $\boldsymbol{R}_2 = -R_2\hat{\boldsymbol{n}}$. Hence define the (linear) spin velocities at the point of impact: $\boldsymbol{\sigma}_i = \boldsymbol{\omega}_i \times \boldsymbol{R}_i$, $i = 1, 2$. Let $\boldsymbol{\sigma} = \boldsymbol{\sigma}_2 - \boldsymbol{\sigma}_1$ be the relative spin velocity at the point of impact and $\boldsymbol{u} = \boldsymbol{v} + \boldsymbol{\sigma}$ be the total relative velocity. Lastly, let $M = m_1 + m_2$ be the total mass of the two-body system, and denote the moments of inertia of the spheres by I_1 and I_2, respectively ($I_i = \frac{2}{5}m_iR_i^2$ for uniform spheres). In the following treatment, all post-collision quantities are denoted by primes ($'$).

The linear impulse suffered by body 1 as a result of the collision is given by $m_1(\boldsymbol{v}_1' - \boldsymbol{v}_1)$. By Newton's Third Law, this must be the negative of the impulse suffered by body 2, hence:

$$m_1(\boldsymbol{v}_1' - \boldsymbol{v}_1) = -m_2(\boldsymbol{v}_2' - \boldsymbol{v}_2). \tag{3.11}$$

By inspection, equation (3.11) is a statement of linear momentum conservation. The two spheres also suffer impulsive torques:

$$\begin{aligned} I_1(\boldsymbol{\omega}_1' - \boldsymbol{\omega}_1) &= m_1\boldsymbol{R}_1 \times (\boldsymbol{v}_1' - \boldsymbol{v}_1), \\ I_2(\boldsymbol{\omega}_2' - \boldsymbol{\omega}_2) &= m_2\boldsymbol{R}_2 \times (\boldsymbol{v}_2' - \boldsymbol{v}_2). \end{aligned} \tag{3.12}$$

It is straightforward to show that equations (3.11) and (3.12) together imply that angular momentum about the centre of mass of the two-body system is conserved. Finally, an expression for the energy loss resulting from the collision can be written as:

$$\boldsymbol{u}' = -\epsilon_n\boldsymbol{u}_n + \epsilon_t\boldsymbol{u}_t, \tag{3.13}$$

where ϵ_n and ϵ_t are the normal and transverse coefficients of restitution, respectively, and $\boldsymbol{u}_n = (\boldsymbol{u}\cdot\hat{\boldsymbol{n}})\hat{\boldsymbol{n}}$ and $\boldsymbol{u}_t = \boldsymbol{u} - \boldsymbol{u}_n$ are the corresponding components of the total relative velocity.

Solving equations (3.11)–(3.13) for $\boldsymbol{v}_1'$ yields:

$$\boldsymbol{v}_1' + \alpha\mu(\hat{\boldsymbol{n}} \times \boldsymbol{v}_1') \times \hat{\boldsymbol{n}} = \boldsymbol{v}_1 + \alpha\mu(\hat{\boldsymbol{n}} \times \boldsymbol{v}_1) \times \hat{\boldsymbol{n}} + \frac{m_2}{M}\left[(1 + \epsilon_n)\boldsymbol{u}_n + (1 - \epsilon_t)\boldsymbol{u}_t\right],$$

where:

$$\alpha \equiv \frac{R_1^2}{I_1} + \frac{R_2^2}{I_2}$$

and

$$\mu \equiv \frac{m_1m_2}{M}.$$

In the case of uniform spheres, $\alpha = \frac{5}{2}\mu^{-1}$. Solving by components and combining the results, the post-collision linear and angular velocities are given by:

$$\begin{aligned} \boldsymbol{v}_1' &= \boldsymbol{v}_1 + (m_2/M)\,\boldsymbol{p}, & (3.14) \\ \boldsymbol{v}_2' &= \boldsymbol{v}_2 - (m_1/M)\,\boldsymbol{p}, & (3.15) \\ \boldsymbol{\omega}_1' &= \boldsymbol{\omega}_1 + (R_1/I_1)\,\boldsymbol{q}, & (3.16) \\ \boldsymbol{\omega}_2' &= \boldsymbol{\omega}_2 + (R_2/I_2)\,\boldsymbol{q}, & (3.17) \end{aligned}$$

where

$$\boldsymbol{p} \equiv (1 + \epsilon_n)\,\boldsymbol{u}_n + \beta\,(1 - \epsilon_t)\,\boldsymbol{u}_t,$$

and

$$\boldsymbol{q} \equiv \mu\beta\,(1 + \epsilon_t)\,\hat{\boldsymbol{n}}\times\boldsymbol{u},$$

with

$$\beta \equiv \frac{1}{1 + \alpha\mu}.$$

For uniform spheres, $\beta = \frac{2}{7}$. It can readily be shown that equations (3.14)–(3.17) reduce to equations (65-1)–(65-4) of Araki & Tremaine (1986) for the equal-mass case, though the results were derived independently.

An expression for the kinetic energy lost during a collision can be derived from equations (3.14)–(3.17):

$$\Delta T = \frac{1}{2}\mu p^2 + \frac{1}{2}\alpha q^2 - \mu\,(\boldsymbol{v}\cdot\boldsymbol{p}) + \boldsymbol{w}\cdot\boldsymbol{q}, \tag{3.18}$$

where

$$\boldsymbol{w} \equiv R_1\boldsymbol{\omega}_1 + R_2\boldsymbol{\omega}_2.$$

For the case of no particle spin ($\epsilon_n = \epsilon$, $\epsilon_t = 1$), this expression reduces to the familiar form:

$$\Delta T = -\frac{1}{2}\mu\left(1 - \epsilon^2\right)v^2.$$

3.5.7 Mergers

Once a dissipative collision has taken place between two particles, it may be the case that the colliders do not retain sufficient kinetic energy to escape their mutual gravitational attraction. Depending on the simulation, it may be desirable to replace the two particles with a single particle of mass $m = m_1 + m_2$ situated at the centre-of-mass position of the pair and having the centre-of-mass velocity (the spin is determined by the constraint that the angular momentum about the centre of mass must be conserved). This treatment saves considerable CPU time at the expense of several assumptions, namely that: (1) the new body is a sphere of the same density as the colliders; (2) there will not be any perturbations that will separate the pair at a later time; and (3) that it is statistically valid to reduce N. The last point can be addressed by deciding on a minimum N (or a maximum mass) for the simulation that still satisfies the self-similarity criterion of the model. The second point could perhaps be reduced in severity by introducing the possibility of fragmentation following collisions (§6.4.1), but motion caused by differential forces or gravitational torques would still be ignored for the merged body. The first point is a fundamental assumption of `box_tree` and cannot be changed without significant code modification.

The choice of whether to allow mergers depends largely on the problem being modelled. For simulations with a long time base and only a moderate number of collisions, allowing merging speeds up the calculations considerably and gives a realistic model of the accretion rate (e.g. §5.2). But for short integrations of high density regions (e.g. §5.3), mergers would almost certainly result in runaway accretion of unacceptable proportions. At the same time, much of the fine structure that may develop would be lost. It may be helpful to study simulations with and without mergers to see if they provide reasonable speed improvement without generating unrealistic results.

If merging is allowed, `box_tree` applies three tests to determine whether two particles should be merged. First, before calculating the final post-collision velocities, the mutual escape velocity of the pair is calculated. If the relative velocity before the collision is less than 1% of the escape velocity, the particles are merged immediately. For reference, the two-body escape velocity is given by:

$$v_{\rm esc} = \sqrt{\frac{2m}{r}}, \tag{3.19}$$

where $r = R_1 + R_2$ following the position correction described in §3.5.3. Otherwise the collision is allowed to take place and the new semi-major axis a of the two-body system is calculated according to:

$$\frac{1}{a} = \frac{2}{r} - \frac{v^2}{m}. \tag{3.20}$$

If $a > 0$ (implying a bound orbit) and $a < R_1 + R_2$, the particles are merged. Finally, if a is not too large (currently less than five times the sum of the particle radii), the pericentre distance r_p is checked using:

$$r_p = (1-e)a = \left\{1 - \left[\left(1 - \frac{r}{a}\right)^2 + \frac{(\boldsymbol{r}\cdot\boldsymbol{v})^2}{am}\right]^{\frac{1}{2}}\right\} a, \tag{3.21}$$

where e is the eccentricity. If the pericentre distance *exceeds* $R_1 + R_2$, the particles are also merged. This is intended to prevent two particles from orbiting each other in tight near-circular orbits that never result in a collision but take a considerable amount of time to compute. Such a situation can arise after repeated dissipative collisions between two particles: since angular momentum is conserved, as the particles lose energy the orbits circularize and the pericentre distance actually increases. Perturbations from nearby particles may enhance this effect.

3.6 Discontinuity Effects

Since boundary crossings, collisions and mergers create discontinuities in position and velocity, the particles involved get special treatment. In particular, after each such event, the force polynomials of the affected particles (or the remaining particle after a merger) must be reinitialized. This is because the integrator relies on past history to predict positions and velocities forward in time. Recall that polynomial initialization requires explicit calculation of the force and force derivatives, so this is a costly procedure. To be consistent, the position and velocity of all other particles are predicted to high order before reinitializing as well.

After polynomial reinitialization, the particle(s) must also be removed completely from the tree and replaced so that the derivatives of the moments of the ancestors are changed to reflect the new position, velocity, and force terms of the affected particles. A special stream-lined version of the repair algorithm was written to handle this case. It may also be necessary to update the time-step list, so special routines were written to handle insertions and deletions from the TSL cleanly. In the case of mergers, one of the two particles (the choice is arbitrary) is removed completely from memory, so data pointers must also be revised. Note that a particle may collide with a ghost, in which case the "real" counterpart of the ghost must be updated in the manner just described, since, logically, that particle must simultaneously be in collision with a ghost of the former particle.

Discontinuity events generally happen infrequently, so that the extra work is negligible over the course of a run. In certain situations, however, reinitializations can become significant. Note that high-order polynomial initialization is unnecessary when using a Hermite integrator (Makino & Aarseth 1992), which is a strong argument for its adoption. See §6.4.3 for further discussion.

Chapter 4

Performance Tests

The purpose of `box_tree` is to provide a fast N-body gravitational simulator for flattened systems that scales like $\mathcal{O}(N \log N)$ without introducing large error. The attainment of that goal is assessed in this chapter. It should be noted (as pointed out in the Introduction) that `box_tree` has undergone considerable evolution since it was first used in "real" simulations. In particular, the latest version of `box_tree` is about twice as fast ($1.9\times$) as the version used in the original 2D timing tests presented in §4.1. The Appendix describes a test suite of initial conditions and parameter files that have been developed specifically for performing many of the checks described here.

4.1 Timing Tests

4.1.1 Two-dimensional Trees

The first timing tests of `box_tree` were performed on typical planetesimal simulations that use 2D trees (§5.2), varying the opening angle parameter θ_C between 0 and 1 (inclusive) in increments of 0.1 rad. Figure 4.1 shows the CPU time as a function of the number of (central) particles N used in each simulation (total number for force calculations $= 9N$), for each value of θ_C. Also shown (dashed line) is the time taken for the direct force when used in place of the tree force. The central particle number was varied between 25 and 250 in increments of 25. Each run was carried out for only one orbit (1 yr) on a Sparc IPX to allow evaluation of all cases in a reasonable time. Note that the same random seed was used for each N (10 in all), which is why kinks in the lines—caused, for example, by the presence or absence of collisions in a particular run—tend to line up vertically. Figure 4.2 shows the result of dividing the CPU time by the number of time-steps actually taken during each run. Notice that the direct force case is a straight line, as would be expected in the $\mathcal{O}(N^2)$ limit. The $\theta_C = 0$ case is also a straight line, since no expansions are performed, but the line is much steeper due to the overhead of searching the tree. The tree force curves for $\theta_C > 0.1$ are much shallower than the direct force case, showing that the tree method becomes more and more advantageous as the particle number increases, consistent with an $\mathcal{O}(N \log N)$ algorithm. For the $\theta_C = 0.6$ case with $N = 250$, the tree method is approximately 7 times more efficient than the direct method. It should be noted that the direct force calculation does not incorporate any special time-saving mechanisms such as a neighbour scheme or fast square root (cf. ALP). However, a further test revealed that `box_tree` was still approximately 50% faster (in CPU/step) than the ALP method for $N \sim 100$ and $\theta_C = 0.6$ at the time of these tests, and 2–3 times faster for $N = 250$.

A timing test with fewer particle cases (N between 20 and 100 in increments of 20)

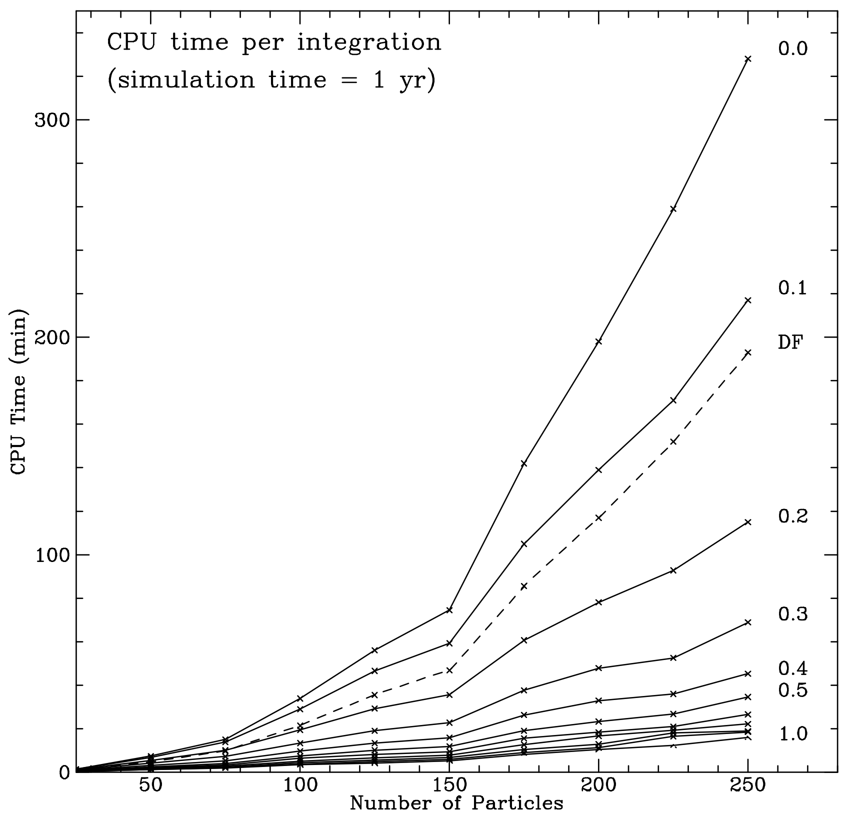


Figure 4.1: CPU time per run for 10 values of θ_C ($0 \leq \theta_C \leq 1$) and for 10 values of N ($25 \leq N \leq 250$). Also shown is the direct force case (dashed line). Initial conditions were identical for each N. Runs were performed on a Sparc IPX using optimized `box_tree` code. Each run was carried out for one orbit (1 yr). Note that all overheads, including particle and tree initialization, are included in the run time.

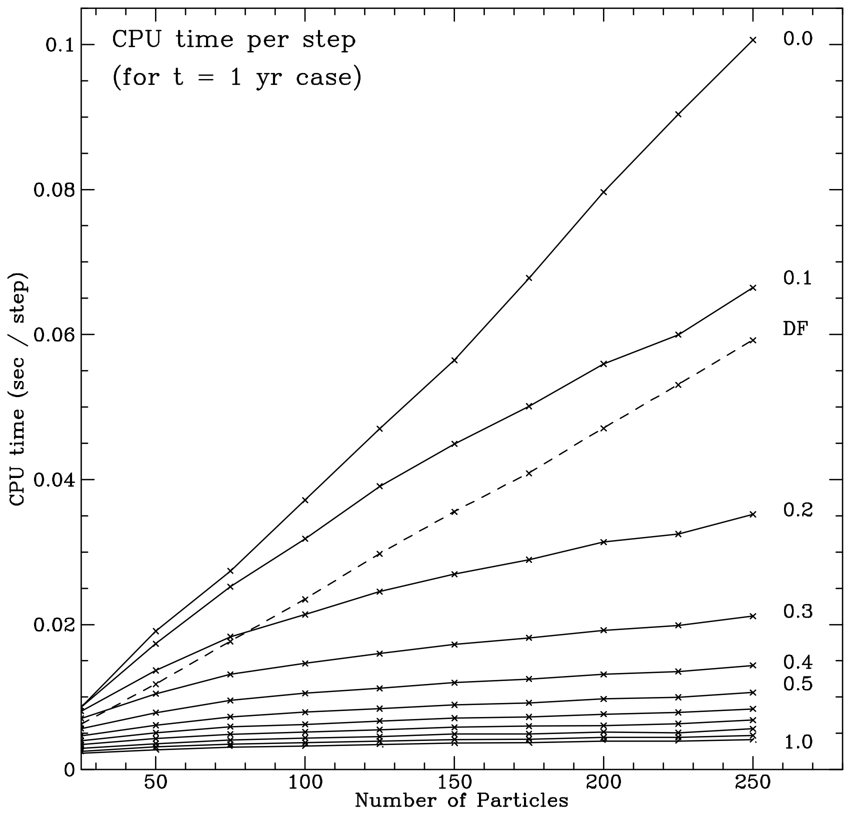


Figure 4.2: CPU time per integration step for the runs shown in Fig. 4.1. Note that the direct force case gives a straight line, as expected for an $\mathcal{O}(N^2)$ algorithm. For reasonable values of θ_C, the lines become virtually flat, indicating that the CPU time per step is almost independent of N. This is consistent with an $\mathcal{O}(N \log N)$ algorithm. Recall that all overheads are included in the CPU time, so the CPU/step values shown here are slightly larger than the true values.

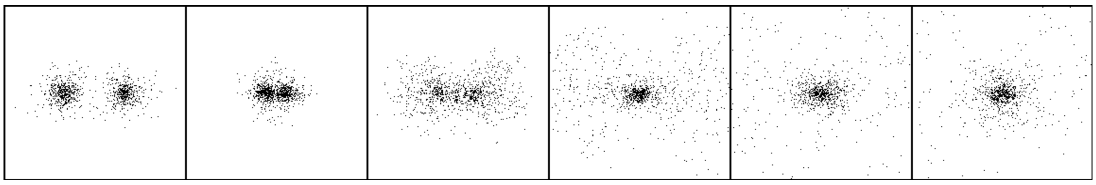


Figure 4.3: Two King models colliding from infinity. Evolution proceeds from left to right in steps of 1 time unit. The two systems collide and pass through each other in the first few frames. The nuclei ultimately merge into one system.

but a longer integration time of 10 orbits was performed for comparison with the shorter runs. The CPU time for each N increased by a factor of 10, but the CPU per step was unchanged. This can be explained by the fact that the equilibrium timescale for the system is much longer than 10 years (see ALP), so that no significant dynamical evolution took place over the run. Again, the tree force was found to be much faster than the direct force, with roughly the same performance ratio for each θ_C.

4.1.2 Three-dimensional Trees

A recent large ($N = 4\,096$) run was performed to compare `box_tree` with `NBODY2` (Aarseth 1994) for a 3D problem. The initial conditions were the same as those used in Barnes & Hut (1989): two unit-mass King models on a head-on collision from infinity. Softening was set to 0.025 for all particles. Figure 4.3 shows snapshots of the evolution of the system. Both runs were performed on a Sparc 10, `NBODY2` requiring 9.0 and `box_tree` 8.1 CPU hrs with comparable integration parameters to evolve the system for 6 time units. Due to the nature of other comparisons being made at the time, a large value of θ_C (0.7) was used in the `box_tree` run, and a minimum time-step of 1/256 was introduced. As a consequence, the root-mean-square deviation in total energy for the `box_tree` run was much greater than that of `NBODY2`, namely 0.3% compared to $\sim$ 0.01% (see §4.4.3). However, for a short simulation, an energy error of a few tenths of a percent is tolerable. To achieve better accuracy, a smaller value of θ would be required and the minimum time-step restriction would need to be removed, presumably making the run time comparable or even greater than that of `NBODY2`. This demonstrates that the simulation geometry plays an important role in tree code performance: flattened systems are better suited to tree codes (for a given value of N), especially when a 2D tree can be employed successfully. Nevertheless, `box_tree` does perform quite well even for difficult models such as this collision between two spherical systems. Discussion of how `box_tree` was adapted to handle collisionless simulations in an inertial frame can be found in Chapter 6 and the Appendix.

4.2 Performance Profile

A performance profile of `box_tree` was obtained using the `gprof` utility available with `SunOS 4.1.3`. The profiler gives a detailed analysis of program flow, and, most important, a breakdown of CPU usage by function. The 11 most expensive functions (those that required more than 1% of the total CPU time) are shown in Table 4.1 for a planetesimal run lasting 100 yr (just under 3.7×10^6 time-steps in $\sim$ 188 CPU min on a Sparc 10). The times shown are for the functions themselves, *not* their children. Note that capitalized `box_tree` functions are global to the entire code. Also, `mcount()` is actually the main routine of `gprof` itself, and would not be present in an optimized version of `box_tree`. The table shows that the most time was spent calculating the multipole expansion during

Table 4.1: Profile of top 11 CPU-intensive `box_tree` functions.

Function	% CPU	Time (sec)	No. Calls
`add_node_multipole_force()`	28.1	3144.82	290531528
`add_tree_force()`	26.6	2984.83	33290226
`sqrt()`	10.6	1190.86	not avail
`add_direct_force()`	7.4	833.95	123329034
`mcount()`	7.1	792.85	not avail
`initialize()`	3.7	417.22	3698914
`add_to_quadrupole()`	2.3	252.64	11629302
`Integrate()`	1.9	216.48	1
`pred_leaf_mono()`	1.6	180.73	17700970
`UpdateMonopole()`	1.4	155.96	7884131
`CheckForCp1()`	1.1	120.16	78376707

Table 4.2: Some important macros used in `box_tree`.

Macro	Purpose
`ZERO(v)`	Zeroes vector ($\boldsymbol{v} = \mathbf{0}$)
`COPY(v1, v2)`	Copies vector ($\boldsymbol{v}_2 = \boldsymbol{v}_1$)
`ADD(v1, v2, v)`	Adds vectors ($\boldsymbol{v} = \boldsymbol{v}_1 + \boldsymbol{v}_2$)
`SUB(v1, v2, v)`	Subtracts vectors ($\boldsymbol{v} = \boldsymbol{v}_1 - \boldsymbol{v}_2$)
`CROSS(v1, v2, v)`	Calculates cross product ($\boldsymbol{v} = \boldsymbol{v}_1 \times \boldsymbol{v}_2$)
`PREDICT_POS_LO()`	Predicts particle position to low order
`PREDICT_VEL_LO()`	Predicts particle velocity to low order
`PREDICT_COM_POS()`	Predicts node centre-of-mass position
`PREDICT_Q_MOM()`	Predicts node quadrupole moment
`CALC_R2_DATA()`	Calculates distance and relative position

the profile run, but note that there were more than twice as many expansions as direct force calculations. The next most expensive `box_tree` routine was `add_tree_force()` itself, which decides whether to use a multipole expansion or direct summation for each node. The system square root function was the third most expensive routine, suggesting that a fast square root algorithm would benefit the code. Notice that less than 2% of the CPU time was spent inside the main integration routine itself between calls to the other functions. Further note that functions directly involving tree repair and extended node checks do not appear on the list as they took less than 1% of the total CPU time.

The latest version of `box_tree` makes use of preprocessor macros to code simple but key functions in-line at compile time. For the purposes of discussion, a macro is considered distinct from a preprocessor "define" directive if it represents an entire code fragment, rather than a simple expression (hence the use of the `macros.h` header file — see Appendix). Many of the macros in `box_tree` perform 3-vector operations, such as dot products and cross products. Others are used for fast (low order) predictions. Use of macros generally saves CPU time (by eliminating repeated function calls and loop variables for example) but results in a larger executable, since every occurrence of a macro must be replaced by the corresponding code fragment. Table 4.2 lists the most important in-line macros used in `box_tree`. These are all global, with the exception of `CALC_R2_DATA()` which is used only in the force routines. Note that macro names are all in capitals, to distinguish them from other function names. Many more macros could be added, but only at the risk of reducing code readability. It is felt that the current set is the most efficient choice. Note that these macros are "hidden" when profiling: they do not appear explicitly in Table 4.1 since they have been replaced by actual code in the executable at compile time. Macros are described in greater detail in the Appendix.

4.3 Force Accuracy

There are a number of factors that contribute to differences between the force calculated using the tree method and that using the direct method. First and foremost is the intrinsic error in the multipole approximation, which is a function of the expansion order, the choice of θ_{C}, and the geometry. Next is the accuracy loss that results from using predicted moments and stretchable nodes rather than updating the tree every time-step. Finally there is the intrinsic error in these predictions, which is a function of the monopole and quadrupole time-steps, and the maximum order of the derivatives used for the predictions.

The absolute error between the two methods is defined by:

$$\text{error} \equiv \frac{|\mathcal{F}_M - \mathcal{F}_D|}{\mathcal{F}_D}. \tag{4.1}$$

In the case of a $\theta_{\mathrm{C}} = 0.6$ expansion to quadrupole order for two equal-mass particles with perfectly determined positions, the largest possible error is $\sim 24\%$ ($\sim 58\%$ for the monopole alone). This is the error obtained when the two particles are at opposite corners of their (3D) box, with the test point located a distance s/θ_{C} from the centre of mass along the diagonal joining the two particles. The *average* error over all possible two-particle configurations is much smaller, less than 0.2%. Figure 4.4 shows the average and maximum errors that result from using finite-order multipole expansions over a unit box seen from distances of 1, 2, and 5 units ($\theta_{\mathrm{C}} = 1.0$, 0.5, 0.2) along the diagonal. Between 2 and 1024 particles (in powers of 2) were placed randomly inside the box, and the total monopole, quadrupole, and octupole contributions were calculated explicitly (the expansion was performed over the *entire* box: particles were not divided into sub-boxes). The average and maximum errors were computed from 10 000 configurations for each choice of N. The figures show that the errors increase by almost two orders of magnitude between $\theta_{\mathrm{C}} = 0.2$ and $\theta_{\mathrm{C}} = 1.0$. As expected, the octupole is always better than or as good as the quadrupole, which is always better than or as good as the monopole (the apparent oscillation at small particle number is a spurious artifact of the natural spline fitting routine). Note, however, that the largest differences occur for moderate values of N; as N increases, there is less and less advantage to using higher orders. Also note that the improvement gained by using the octupole over the quadrupole is far less than that gained by using the quadrupole over the monopole. Hence, the optimal expansion is probably monopole plus quadrupole, with an expected average error of less than $\sim \frac{1}{2}\%$ (maximum error of order 10–20%) for reasonable values of θ_{C} ($0.375 \lesssim \theta_{\mathrm{C}} \lesssim 0.625$ in a 2D tree, $0.35 \lesssim \theta_{\mathrm{C}} \lesssim 0.55$ in 3D) and for systems that have a fairly uniform particle distribution. Recall that this expected error is for a *static* configuration; in the dynamic case the errors will generally be larger.

There are provisions in `box_tree` to perform some fairly extensive force checking by comparing multipole expansions with their direct sum equivalents. As a result of these tests it was discovered that it is possible for the quadrupole (or the octupole even) to make the force approximation *worse* in some cases. Experimentation with this led to Fig. 4.5, which shows that for the $N = 3$ case there exists a particle configuration whose monopole contribution is a better approximation of the force than the quadrupole for certain ranges in position angle of the test point. In this example the system is confined to a plane and

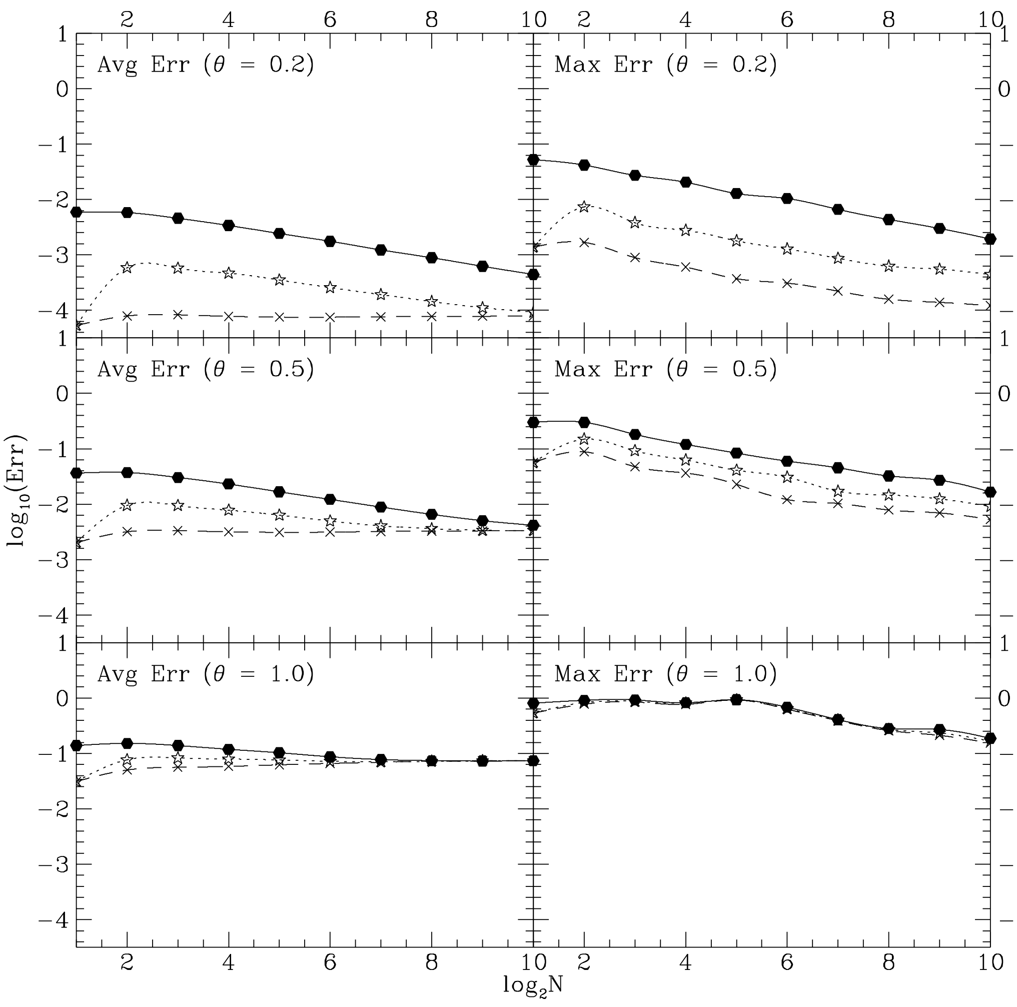


Figure 4.4: Average and maximum force errors [cf. equation (4.1)] that result from terminating multipole expansions at the monopole (solid line), quadrupole (dotted line), and octupole (dashed line) terms. Graphs are shown for three values of θ_C (0.2, 0.5, and 1.0) taken along the diagonal of a box packed with 2^m particles, where m runs from 1 to 10. In general, both average and maximum errors increase with θ_C and decrease with N. Maximum errors are typically an order of magnitude larger than the average errors. Differences between the expansion orders increase initially but converge with larger N. Also, these differences are more marked for the smaller values of θ_C.

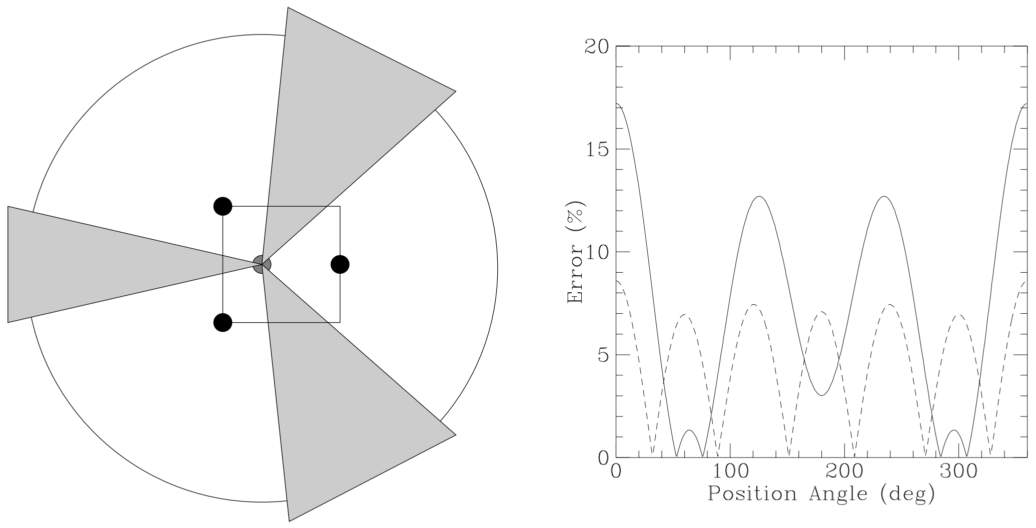


Figure 4.5: Illustration that the quadrupole approximation can sometimes be worse than the monopole. The diagram on the left shows three particles (solid black dots) in a symmetric planar configuration at the edges of a box. The centre of mass is marked by the shaded dot. The outer circle is the locus of points at which the box subtends an angle of 0.5 rad (i.e. 2 box lengths from the centre of mass). The shaded triangular wedges indicate the regions on the outer circle where the quadrupole approximation is worse than the monopole. The graph at the right shows the actual percentage errors in the force as a function of position angle. The solid line is the absolute monopole error (average 7.4%), and the dashed line is the absolute quadrupole error (average 4.7%).

the opening angle is fixed at 0.5. Unfortunately, it is not feasible to predict in advance which configurations are likely to do worse with the quadrupole. However, as seen in Fig. 4.4, the quadrupole contribution always does better on average.

It should be noted that because boundary conditions are applied to predicted node positions (§3.4.2), some care needs to be exercised when comparing the tree force to the direct force. The problem is that when a node's predicted position lies outside the box system, *all* of its leaves are moved to the other side of the system, regardless of whether or not the leaves themselves are predicted to cross the boundary. In the direct force routine, however, only individual particles are subject to boundary conditions, since there is no concept of a collective cell. Hence apparent force errors may result simply because some particles have been wrapped in the first case, but not in the second. The solution is always to use the tree code when testing force routines, but to replace multipole expansions with a direct summation over the leaves without applying boundary conditions to the children. Naturally this makes the force testing routine much slower, but it ensures that the test is valid. Note that this "direct tree" method was *not* used in the timing tests discussed above (§4.1), where these kinds of force discrepancies were immaterial.

In order to improve on predictions, either the number of terms in the prediction polynomial must be increased or the size of the update time-steps must be decreased. Since it is easier to adjust time-steps than to add or remove prediction terms, several runs were performed to determine the optimal values of the multipole time-step coefficients [$\epsilon_{\rm M}$ and $\epsilon_{\rm Q}$ in equations (3.3) & (3.4)] that minimize both force errors and CPU expense. The values of $\epsilon_{\rm M}$ and $\epsilon_{\rm Q}$ were independently varied in powers of 10 between 10^0 and 10^{-7} for two cases, one with $N = 50$ for 5 years, and the other with $N = 100$ for 1 year. From these tests and earlier runs, $\epsilon_{\rm M}$ and $\epsilon_{\rm Q}$ have been chosen as 0.001 and 0.01, respectively. These values tend to result in a roughly equal number of monopole and quadrupole updates per unit time (about 10^5 in the first test case and 4×10^4 in the second).

A box_tree run with the usual planetesimal initial conditions (§5.2) was performed in order to illustrate these various aspects of the code accuracy. The run was carried out for 100 particles over 100 yr and required roughly 2 CPU days to complete because of the force checking. The average absolute error (i.e. equation (4.1) averaged over every expansion for each integration step of the entire run) was 0.176% and the maximum error was 18.0%. In only 268 instances did force errors occur above the 10% level, out of a total of 4.7×10^6 time-steps. Of these 268, only 44 were unique to a particular particle at a particular time, since particles tend to stay within expansion range of any given node for several time-steps. Approximately 130×10^6 direct force calculations and 370×10^6 multipole expansions were performed during the run. There were 6.3×10^6 monopole updates and 4.8×10^6 quadrupole updates. The total z angular momentum of the particles—with the boundary corrections applied (cf. §4.4.2)—remained within 1.5×10^{-6} of its starting value, staying above and below in roughly equal amounts. In all there were 14 collisions, of which 8 resulted in mergers. Both the average and maximum absolute errors agree well with Fig. 4.4, and the number of collisions is reasonable given that roughly 30 first-time collisions would be expected to occur by the time dynamical equilibrium is established in the system [cf. equation (8-123) in Binney & Tremaine 1987].

4.4 Constants of Motion

The linear momentum, angular momentum, and total energy of an isolated system of particles are all constants of motion if the only forces acting on the particles are those arising from mutual gravitational attraction. Consequently, conservation of these quantities can be used as a check on the accuracy of the integration. Unfortunately, such checks are difficult to apply in bounded and/or non-inertial systems since boundary conditions introduce discontinuities and the usual conservation laws do not necessarily hold in accelerated frames. Particle collisions also introduce various discontinuities. However, it is possible in many instances to make allowances for such difficulties, as will be shown in the following section.

4.4.1 Linear Momentum Conservation

Conservation of linear momentum is equivalent to conservation of centre-of-mass velocity in an inertial frame. The centre-of-mass position $\boldsymbol{r}_g$ and velocity $\boldsymbol{v}_g$, and the deviation from their initial values, are calculated as part of the regular box_tree output routines. For most simulations the initial value of $\boldsymbol{v}_g$ is arranged to be zero to simplify analysis. In a system with periodic boundary conditions, the following correction must be applied to $\boldsymbol{r}_g$ and $\boldsymbol{v}_g$ following a boundary crossing:

$$\begin{aligned}
\boldsymbol{r}'_g &= \boldsymbol{r}_g - (m_i/M)\boldsymbol{r}_b, \\
\boldsymbol{v}'_g &= \boldsymbol{v}_g - (m_i/M)\boldsymbol{v}_b,
\end{aligned}$$

where m_i is the mass of the particle crossing the boundary, M is the total mass of all (central) particles in the system, and $\boldsymbol{r}_b$ and $\boldsymbol{v}_b$ are the position and velocity, respectively, of the neighbouring box that the particle is entering. In the case of shearing boxes in the rotating frame, equation (2.8) can be used to determine the box position and velocity. In an inertial frame, the box positions are fixed (e.g. set $t = 0$ in the transformation equation). It should be noted that boundary crossings are detected after they occur, just as collisions are detected after the actual impact (§3.5.3). Unlike in the collision case however, there is no need to adjust the position or velocity to correct for the slight overlap, since a ghost particle will temporarily fill the gap in the central box.

Note that in the shearing model, if a particle crosses a boundary in the $\pm x$-direction, $\boldsymbol{v}_g$ will take on a non-zero value, so that a systematic motion of the centre-of-mass position will develop. This is because the particle's y-velocity changes sign as the particle is translated from one side of the box to the other. This problem can be eliminated by using the centre-of-mass velocity with respect to the local shear, that is, by subtracting $-\frac{3}{2}\Omega x_i$ from the y-component of the velocity of each particle when calculating $\boldsymbol{v}_g$. A typical `box_tree` run with WT-type initial conditions (§5.3) generally conserves this adjusted centre-of-mass velocity to within $\sim 10^{-2}\Omega s$ (a few percent) with interparticle gravity included in the simulation, and $\sim 10^{-6}\Omega s$ without, where, as usual, s is the box size.

4.4.2 Angular Momentum Conservation

The total z angular momentum (TZAM) is monitored by `box_tree` in both the inertial and non-inertial (rotating) case. Fortunately, the TZAM in the rotating frame differs from the true TZAM as seen in the inertial frame only by a multiplicative and an additive constant. To see this, note that the z angular momentum, with respect to the inertial frame, of a particle with coordinates (x, y, z) and z-spin ω_z in the rotating frame is given by:

$$L_z = m\left(X\dot{Y} - \dot{X}Y\right) + I\Omega_Z,$$

where to first order $X = a + x$, $Y = y$, $\dot{X} = \dot{x}$, $\dot{Y} = \Omega(a + x) + \dot{y}$, and $\Omega_Z = \Omega + \omega_z$ in the notation of §2.1.1. If $a \gg x, y$ and $\dot{x}$ is small, then:

$$L_z = ma\left(\dot{y} + 2\Omega x\right) + I\omega_z + m\Omega a^2 + I\Omega.$$

Since a and Ω are constants, it is sufficient to call $\ell_z^i = m_i(\dot{y}_i + 2\Omega x_i) + I_i\omega_{z,i}$ the z angular momentum of particle i. In the case of periodic boundary conditions, the following correction must be made after each boundary crossing to keep the TZAM $\ell_z = \sum_i^N \ell_z^i$ constant:

$$\ell_z' = \ell_z + \frac{1}{2}\Omega m_i i_x s,$$

where $i_x s$ is the x-offset of the ghost box that the particle is entering [cf. equation (2.8)]. Note that with this linearized formulation of the TZAM, crossings in the $\pm y$-direction can be ignored. Also, the spin component is not affected by a boundary crossing. In the notation of the previous section:

$$\ell_z' = \ell_z + m_i\left(\boldsymbol{v}_{b,y} + 2\Omega \boldsymbol{r}_{b,x}\right). \tag{4.2}$$

In the non-shearing case, the correction is given by:

$$\ell_z' = \ell_z + m_i\left(\boldsymbol{r}_b \times \boldsymbol{v}_i\right)_z, \tag{4.3}$$

where $\boldsymbol{v}_i$ is the velocity vector of the particle.

TZAM conservation is also affected by particle collisions. The position and velocity offsets described in §3.5.3 and §3.5.4 change the angular momenta of the colliders, but since the exact adjustments are known, corrections similar to those just presented can be applied to the TZAM to compensate. For example, in the rotating frame:

$$\ell_z' = \ell_z + m_1\left(\Delta\boldsymbol{v}_1 + 2\Delta\boldsymbol{r}_1\right) + m_2\left(\Delta\boldsymbol{v}_2 + 2\Delta\boldsymbol{r}_2\right).$$

There is a similar expression for the inertial frame. Note that no correction is necessary following a merger since the relative angular momentum of the colliders is transferred to the spin of the new particle (§3.5.7).

A typical WT-type simulation conserves TZAM to within $10^{-6}M$, where M is the total mass of the system.

4.4.3 Total Energy Conservation

The total energy of a particle is defined as the sum of its kinetic energy (which may include rotational energy) and its gravitational potential energy. Due to the approximate nature of the linearized equations of motion [cf. equation (2.7)] and the presence of ghosts particles, the total energy is *not* used as a check in the box model. For a system without ghosts in an inertial frame, it is found that the root-mean-square (RMS) deviation of the total energy generally scales inversely as the time-step coefficient η (§3.5.1) to the fourth power, as expected for a fourth-order integrator. The TZAM and linear momentum conservation are similarly controlled by η.

Boundary crossings and corrections for particle collisions also introduce discontinuities in the total energy. In addition, there are losses in kinetic energy following inelastic bounces [cf. equation (3.18)] and merger events. Changes to the kinetic energy can be accounted for by using expressions similar to those presented in the previous section, since the change in velocity is known precisely. However, discontinuities in the gravitational potential energy V that result from a position change can only be corrected in practical fashion by calculating V both before and after the discontinuity event. This is expensive and only approximate because particle positions must be predicted to the current time for such calculations. These calculations are required, however, in order to perform the collision velocity adjustments described in §3.5.4 in the unbounded inertial frame.

Typical conservation accuracies for the total energy are presented for some inertial systems in the Appendix.

Chapter 5

Simulations

To date, `box_tree` has been used to study two major areas of planetesimal dynamics: the intermediate stage of early planet formation, and the present-day evolution of planetary rings. Results from these simulations are presented in this chapter. Application of `box_tree` to other areas of study is discussed in Chapter 6.

5.1 Units

To simplify discussion, the system of units used in `box_tree` for planetesimal simulations will be outlined first. Further details can be found in the Appendix.

In order to eliminate inconveniently sized numbers and redundant multiplications, `box_tree` implicitly assumes—in the notation of Chapter 2—that $a = 1$, $\Omega = 1$, and $GM = 1$. For the early planetesimal simulations (§5.2), the length unit is one astronomical unit (1 AU $\simeq 1.5 \times 10^{11}$ m), the mass unit is one solar mass ($M_\odot \simeq 2 \times 10^{30}$ kg), and the time unit is one sidereal year (1 yr $\simeq 3.2 \times 10^7$ s). For the simulations of Saturn's rings in §5.3, the length unit is the orbital radius of the centre of Saturn's B ring ($R_B \sim 10^8$ m), the mass unit is the mass of Saturn ($M_S \simeq 5.7 \times 10^{26}$ kg), and the time unit is the orbital period at R_B ($T_B \sim 3 \times 10^4$ s). Speeds are measured in units of the orbital velocity (i.e. ~ 30 km/s at 1 AU from the Sun, ~ 19 km/s at 1 R_B from Saturn). Some quantities have been converted back to familiar units for presentation.

5.2 Early Planetesimals

Much of the work presented in this section follows on from earlier studies by PLA and ALP. Their work concentrates on the third stage of terrestrial planet formation, namely the coagulation of planetesimals into protoplanets (Safronov 1969; etc.). Simulations of this evolutionary stage require large dynamic range, high spatial resolution, and a long time base, and are therefore well suited to treatment by `box_tree`.

5.2.1 Initial Conditions

There are several constraints on the choice of initial conditions for early planetesimal simulations. First, the number N and mass m_p of the planetesimals must be consistent with estimates of the total planet-forming mass M_t in the early solar system. ALP took $M_t \simeq 5M_\oplus$ inside 1 AU, where $M_\oplus \simeq 3 \times 10^{-6} M_\odot$ is the mass of the Earth. Second, the box size s must be chosen small enough so that the linearized equations of the box model [cf. equation (2.7)] remain valid, but large enough so that a reasonable number of particles (a few hundred or more) can be accommodated in order to satisfy the self-similarity criterion. In particular, if the regime of interest is at $a = 1$ AU in the planetesimal disk, then it is required that $s \ll 1$ and $s \gg R_p$, where R_p is the planetesimal radius (more properly, $s \gg R_R$, where R_R is the Roche or Hill sphere radius of a planetesimal given by $R_R = (m_p/3M_\odot)^{1/3}\, a$). Finally, s must be greater than the epicyclic frequency $\Delta = 2\sigma/\Omega$, where σ is the velocity dispersion, to ensure a small radial mean free path.

For the models that follow, initial velocity dispersions appropriate for a "warm start" were chosen. Particle positions in x and y were determined randomly and particle velocities were set to the appropriate dispersion multiplied by a Gaussian deviate. If required, positions and velocities in z were obtained by randomly choosing a maximum distance above the midplane equal to the initial z velocity dispersion times a Gaussian deviate and randomly choosing a phase between 0 and 2π for the subsequent harmonic motion [cf. equation (2.7)]. Initial bound pairs were rejected if the semi-major axis was smaller than one Roche radius.

5.2.2 Long Integration (ALP Model)

As a test of `box_tree` the initial conditions of model Z2 in ALP were chosen for a long integration of 10 000 years. In this simulation, $N = 100$ (central particles), $m_p = 8\times10^{-11}$, and $\rho = 1.4$ g/cm^3 (so $R_p \simeq 2 \times 10^{-6}$). Hence this simulation represents a system of $M_t/m_p \sim 2 \times 10^5$ early planetesimals. The box size s was 0.04 (half-width of 66 Roche radii) and the initial velocity dispersions were $\sigma_x = 3 \times 10^{-4}$, $\sigma_y = \sigma_z = \sigma_x/2$ (so the initial epicyclic amplitude $\Delta \ll s$). The radial coefficient of restitution was 0.5. Fig. 4 of ALP shows the velocity dispersion evolution of the planetesimals over 10 000 years: the system heats up rapidly in the first few hundred years and then gradually begins to approach an equilibrium between collisional dissipation and gravitational excitation after several thousand years. Throughout the run, σ_x always remains roughly twice as large in magnitude as σ_y and σ_z. This behaviour was found to be in agreement with analytical calculations performed by PLA.

Figure 5.1 shows the velocity dispersion evolution as calculated by `box_tree` over the same interval. The graphs are qualitatively identical, if differences in initial positions and velocities (owing to a different set of random values) are allowed for. The opening angle θ_C was 0.6 and the monopole and quadrupole time-step coefficients were 0.001 and 0.01 respectively (cf. §4.3). Fluctuations in σ_x and σ_y are at the $\sim 100N^{-1/2}\%$ level and may therefore be attributed to random noise (note that this noise increases with time since the particle number decreases as a result of mergers). There is a systematic oscillation (period $\sim$ 50–100 yr) in σ_z present in both graphs, but the higher sampling rate makes the effect much more visible in Fig. 5.1. This phenomenon is explained in detail in §5.2.4. Collision statistics for the `box_tree` run are shown in Fig. 5.2 and should be compared with Fig. 9 of ALP. Fewer collisions (and therefore fewer mergers) occurred in the `box_tree` run, but variations up to a factor of 2 in the collision rate can be attributed to differences in the initial conditions. Hence the `box_tree` and ALP runs are essentially in agreement.

A check of the self-similarity assumption made in the box method was also performed in the same manner as in ALP. Two runs with equal surface density but different particle number were performed and the subsequent evolutions compared. Figure 5.3 shows the growth in σ_x over 100 yr for (1) $N = 200$, $s = 0.04$ (solid line); and (2) $N = 50$, $s = 0.02$ (dashed line). Both systems evolve to $\sigma_x \sim 0.0018$ over the course of the run, even though the noise in the $N = 50$ case is greater due to the smaller particle number. The latter case also has a slightly higher fractional collision/merger rate, but again this may be attributable to differences in initial conditions. Hence `box_tree` obeys the self-similarity

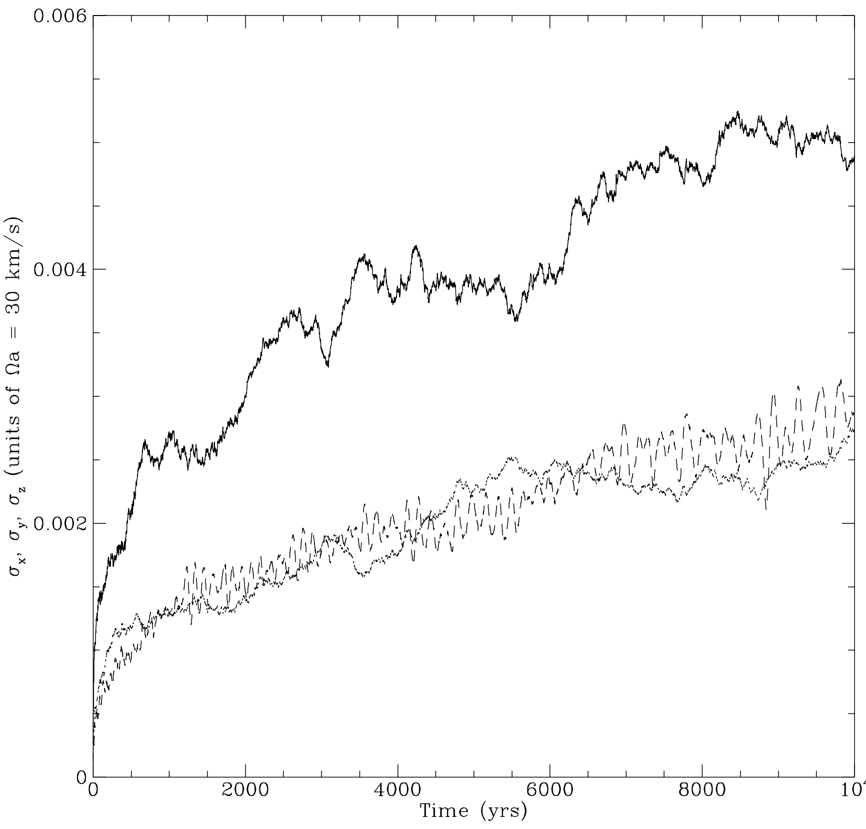


Figure 5.1: Velocity dispersion evolution over 10 000 yr for a typical early planetesimal simulation (here $N = 100$). The solid line follows the radial component σ_x; the dotted and dashed lines trace σ_y and σ_z, respectively. The system undergoes rapid heating in the first few hundred years, then slowly climbs towards equilibrium. The periodic oscillation in σ_z is explained in §5.2.4. Otherwise fluctuations are mostly random in nature.

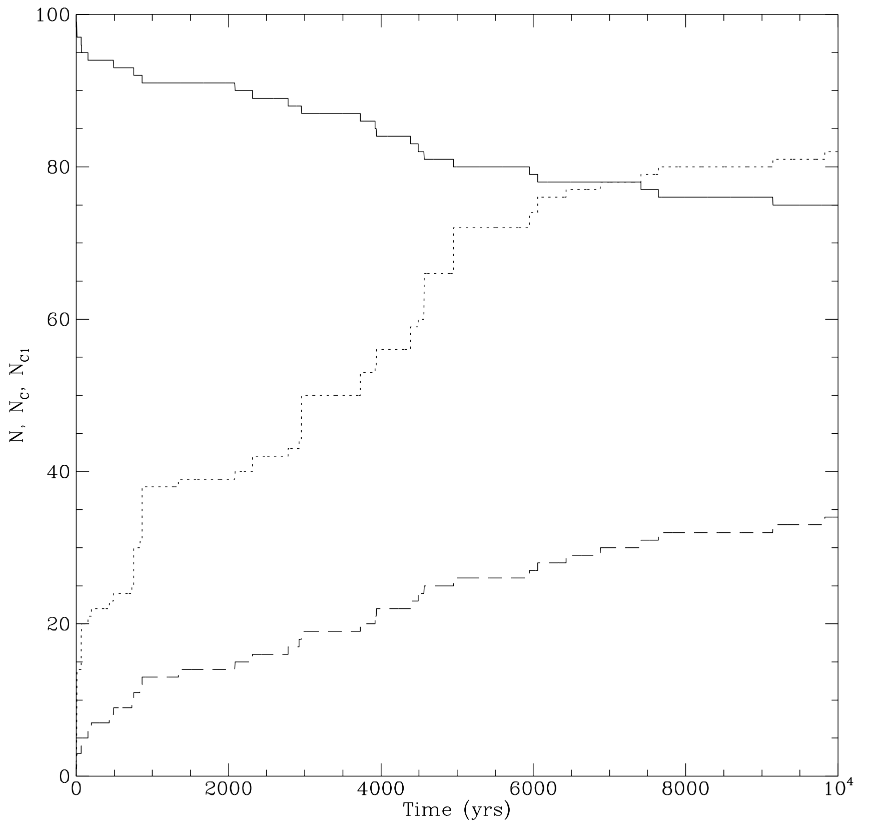


Figure 5.2: Collision statistics for the system described in Fig. 5.1. The solid line shows the number of particles as a function of time; the dotted line shows the total number of collisions; and the dashed line shows the total number of “first-time” collisions, that is, the number of collisions with repeated bounces between the same particles over an unbroken interval removed. The graph shows that most encounters lead to a merger after one or two collisions.

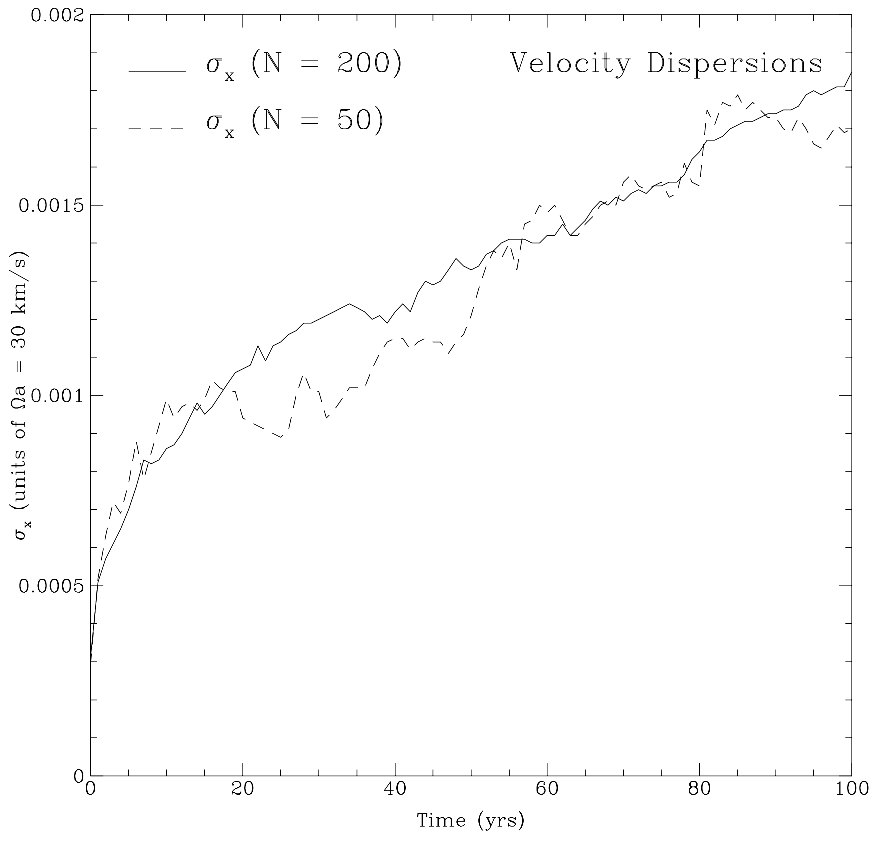


Figure 5.3: Velocity dispersion in x over 100 yr for two systems of particles in 3D: (1) 200 particles in a box of width 0.04 (solid line); and (2) 50 particles in a box of width 0.02 (dashed line). The systems should behave similarly since they have identical surface densities. This is confirmed by the fact that the velocity dispersions in both systems follow the same general trend, with deviations in the $N = 50$ case being larger due to the $N^{-1/2}$ statistics.

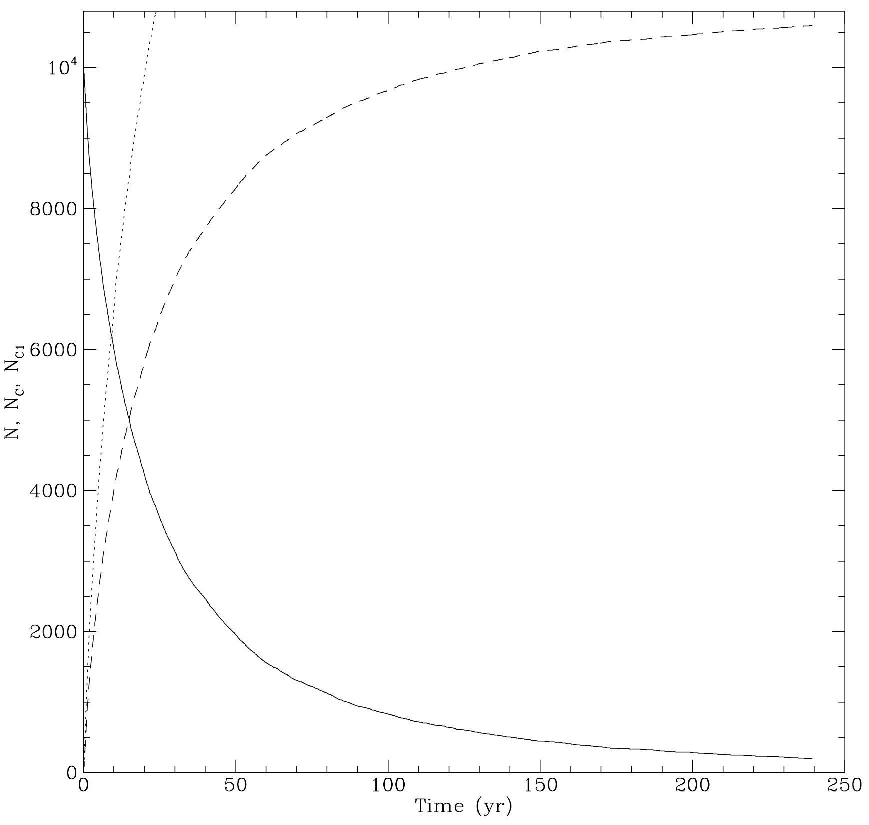


Figure 5.4: Collision statistics for an early planetesimal system of 10 000 particles in 2D. There were roughly 20 000 collisions in total (dotted line). The particle number (solid line) drops off roughly as $t^{-3/2}$ over the interval shown. The number of first-time collisions (dashed line) mirrors the particle number, implying that most initial encounters ultimately result in mergers.

criterion, at least within the current regime of interest.

5.2.3 Large N Model

Since the value of N in the long integration described in the previous section was relatively small, `box_tree` only offered a modest gain over traditional direct integration methods. To take advantage of `box_tree` then, a model with $N = 10\,000$ was devised. To speed up the evolution, however, the run was restricted to 2D (see ALP for a justification of using 2D simulations to model 3D). For the large N model, the initial planetesimal mass was $m_p = 3 \times 10^{-15}$, so the system represents 5×10^9 planetesimals. The box size was set to $s = 0.008$ and the initial velocity dispersions were $\sigma_x = 1 \times 10^{-5}$ and $\sigma_y = 5 \times 10^{-6}$. A transverse coefficient of restitution of 0.5 was introduced for this run. The tree parameters were the same as those used in the previous section. The run was carried out for ~ 240 yr, at which point N had been reduced to just under 200 particles.

Figure 5.4 shows the collision statistics for the run. The number of particles drops smoothly from the initial 10 000, and asymptotically approaches zero. The number of first-time collisions closely mirrors this behaviour, indicating that most initial encounters result in mergers. The particle number as a function of time can be modelled approximately by

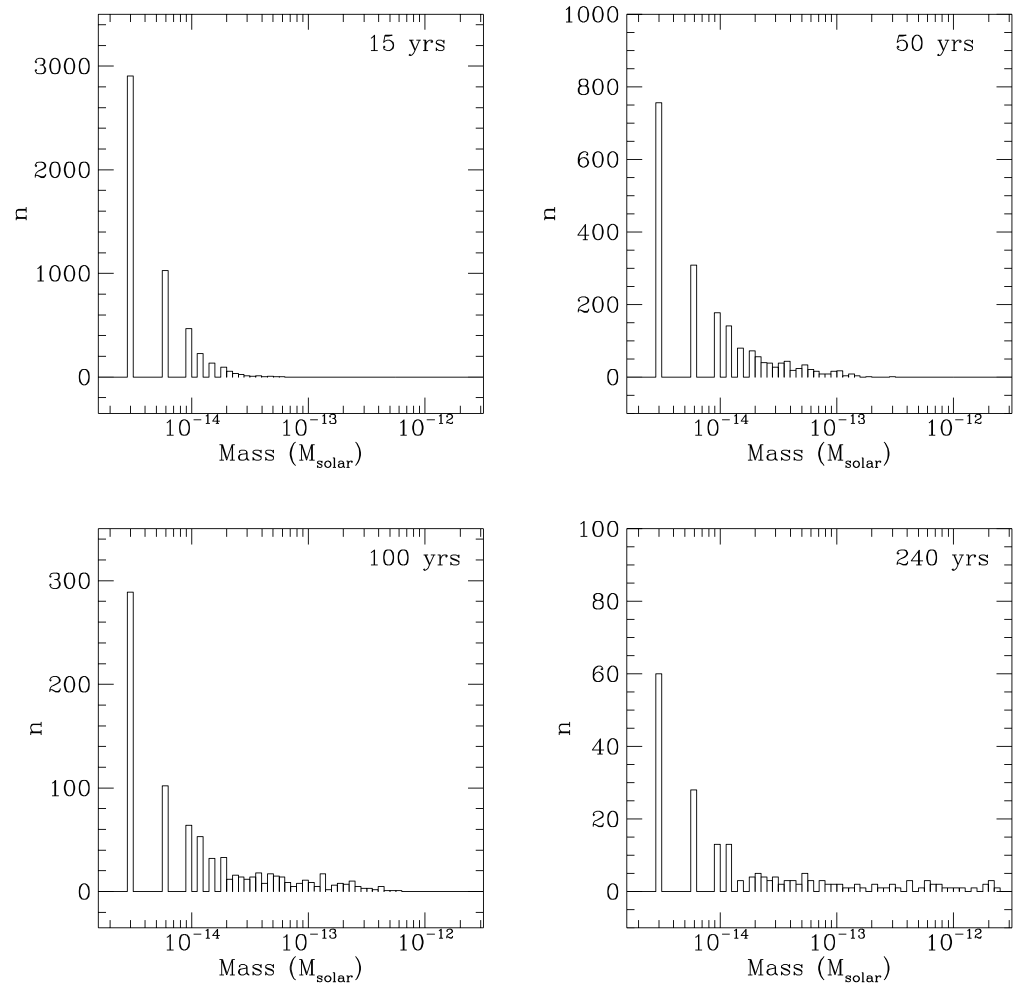


Figure 5.5: Mass spectrum evolution of the system described in Fig. 5.4. The starting mass is 3×10^{-15}. From the graphs, there is no evidence of runaway accretion, although the evolution is rapid (note the change in scale between graphs).

a function of the form:

$$N(t) = \frac{N_0 k}{t^{3/2} + k}, \tag{5.1}$$

where N_0 and k are constants ($N_0 = N(0) = 10\,000$, $k \sim 75$). This function falls off faster than an exponential over the time interval shown. From equation (5.1), the merger rate goes roughly as $dN/dt \propto -t^{-5/2}$.

Figure 5.5 shows histograms of the mass spectrum at $t = 15$, 50, 100, and 240 yr. Recall that all particles start with equal mass, so as t increases the $m = 3 \times 10^{-15}$ bin is depleted, and bins at integer multiples of the starting mass are populated. The model exhibits runaway accretion in the sense that the largest particles tend to grow fastest, stretching the mass spectrum out to the high-mass end. At the end of the simulation, the largest mass has grown by 750 times. Note, however, that the largest population still consists of the smallest particles.

Figure 5.6 shows the spin evolution at four time intervals. Recall that since the model is restricted to 2D, only spins directed about the z-axis develop. There is no evidence for preference of either clockwise or anti-clockwise rotation. By time $t = 240$ yr, only

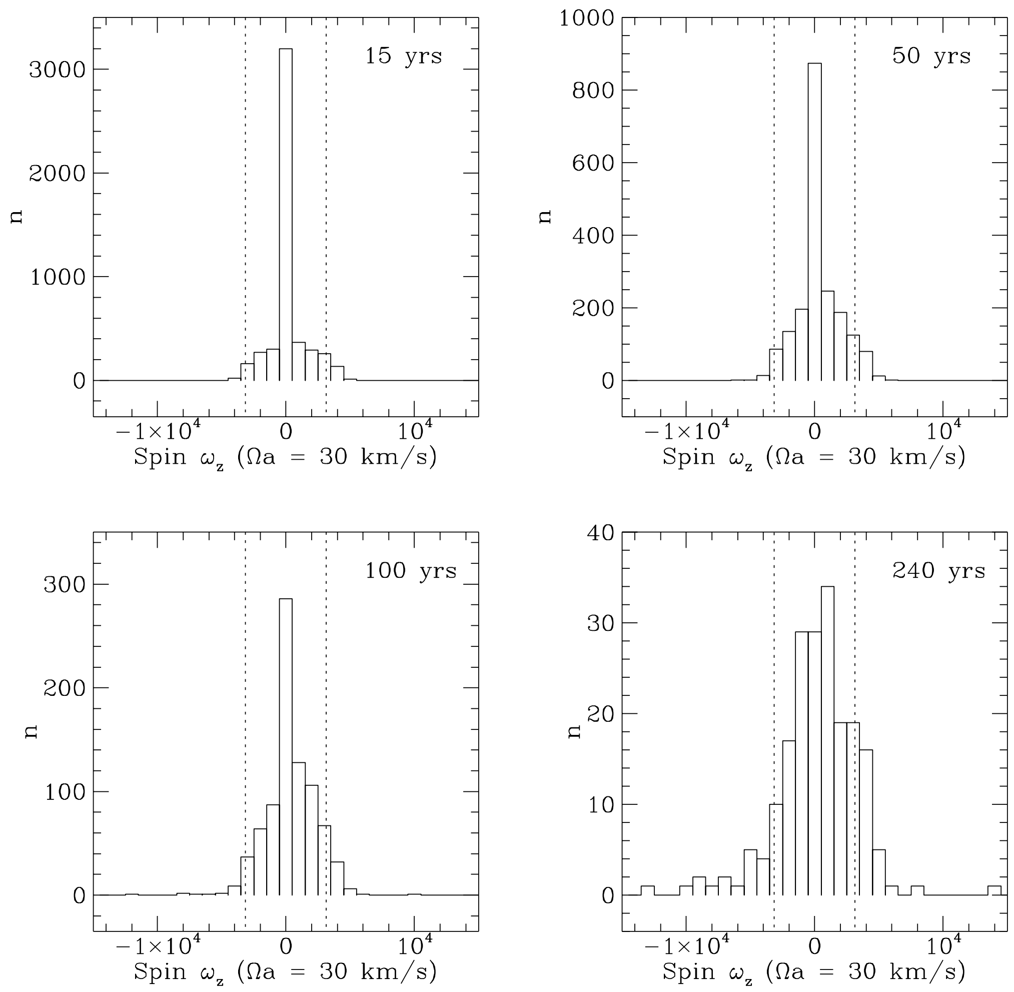


Figure 5.6: Spin evolution of the $N = 10\,000$ planetesimal system. The dotted lines denoted the classical maximum spin rate before break-up. There is no apparent preference for clockwise or anti-clockwise spin.

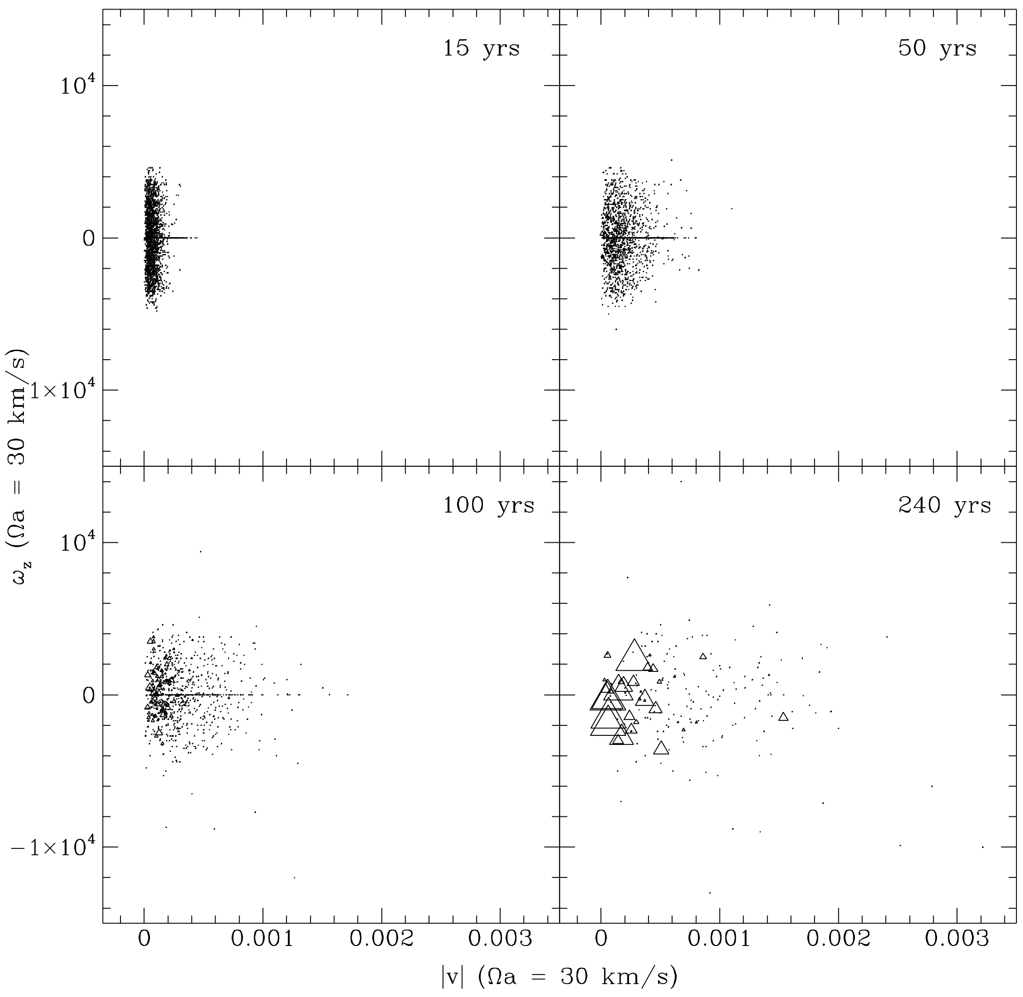


Figure 5.7: Spin-velocity phase space of the $N = 10\,000$ planetesimal system. The triangular symbols are proportional to the particle mass. Equipartition of energy begins to develop near the end of the run.

a handful of particles have zero spin, though many more have not merged. Therefore, there are particles in the simulation that have gained spins only through the mechanism of glancing dissipative collisions. Indeed, the particles with spins $|\omega_z| > 10^4$ are unmerged particles. The dotted line shows the classical maximum spin rate (both positive and negative) before a uniform solid body breaks apart. The maximum spin is a function of density only and is given by $|\omega_{\max}| \simeq (\frac{4}{3}\pi\rho G)^{1/2}$, in unscaled units. Not many particles cross this boundary, and those that do tend to have small mass. Nevertheless, an allowance for fragmentation that takes spin into account would probably improve the model (§6.4.1).

The last figure in this section (Fig. 5.7) shows the spin-velocity phase space at the same four time intervals. The size of the triangular symbols is proportional to the particle mass, emphasizing the late development of the largest masses. The velocity is taken with respect to the local shearing motion. The figure illustrates that the system develops a rough equipartition of energy, with the largest masses having the smallest velocities and spins.

5.2.4 Vertical Oscillations

The apparent oscillation in the z velocity dispersion seen in Fig. 5.1 is due to a sampling effect. For the purpose of analysis and plotting, it has always been convenient to record simulation data in yearly intervals. Unfortunately, planetesimals in a disk at 1 AU have a fundamental oscillation period of $T = 1$ yr. This can be illustrated easily with plots similar to Fig. 5.1 by choosing a sampling interval that is smaller than 1 yr. Such plots exhibit large half-yearly oscillations roughly centred on the relatively smooth curves obtained with 1 yr sample intervals. The period of these oscillations is 0.5 yr because the dispersion is the square root of a quadrature sum of 1 yr sinusoids (see below), introducing a factor of 0.5. Physically, these fundamental oscillations arise from the linearized equations of motion [cf. equation (2.7)]. In the absence of interparticle gravity, these equations have solutions containing periodic terms with angular frequency Ω (see WT). Indeed, the equation for $\ddot{z}$ describes a simple harmonic oscillator with such a frequency.

When interparticle gravity is included, the oscillation frequency can differ slightly from Ω. Consider the case of the z coordinate. For any particle i, the equation of motion in z is given by:

$$\ddot{z}_i = (\mathcal{F}_i)_z - \Omega^2 z_i,$$

where

$$(\mathcal{F}_i)_z = \sum_{j \neq i}^{N} \frac{m_j (z_j - z_i)}{|r_{ij}|^3}.$$

To simplify the argument, assume $m_j = m$ and replace $|r_{ij}|$ with an *average* distance, denoted by $\bar{d}$. This value tends to increase with time. Further assume that $\sum_j z_j \sim 0$, appropriate for a random (Gaussian) distribution in z. Then, dropping the i subscript:

$$\mathcal{F}_z \simeq -\frac{Nm}{\bar{d}^3} z.$$

Thus

$$\ddot{z} \simeq -\left(\Omega^2 + \frac{Nm}{\bar{d}^3}\right) z,$$

which for $\Omega = 1$ describes a simple harmonic oscillator with period

$$T_{\text{new}} = \left(1 + \frac{Nm}{\bar{d}^3}\right)^{-\frac{1}{2}} \simeq 1 - \frac{Nm}{2\bar{d}^3}.$$

For the simulation illustrated in Fig. 5.1, $N = 100$ and $m = 8 \times 10^{-11}$. If $\bar{d}$ is set to about half the initial average distance between planetesimals (viz. $\bar{d} \sim \frac{1}{2}s$, where the box size $s = 0.04$), then

$$T_{\text{new}} \simeq 1 - 0.004 \equiv T + \delta T.$$

Note that $\delta T \ll 1$, justifying the expansion to first order of the inverse square root. The new oscillation period is smaller than the fundamental period, as expected for an overall gravitational enhancement.

Since the sampling shown in Fig. 5.1 has an interval of exactly T, a spurious oscillation of period $T/\delta T \gg T$ results ($T/2\delta T$ on the dispersion graph). To prove this, consider a sinusoid of the form:

$$y = \sin\left[\frac{2\pi}{T}\left(1 + \frac{\delta T}{T}\right)^{-1} x\right].$$

If $\delta T \ll T$,

$$\begin{aligned} y &\simeq \sin\left[\frac{2\pi}{T}\left(1-\frac{\delta T}{T}\right)x\right], \\ &= \sin\left(\frac{2\pi x}{T}-\frac{2\pi\delta T x}{T^2}\right), \\ &= \sin\frac{2\pi x}{T}\cos\frac{2\pi\delta T x}{T^2}+\cos\frac{2\pi x}{T}\sin\frac{2\pi\delta T x}{T^2}. \end{aligned}$$

Now if $x = nT$, where n is a positive integer,

$$y \simeq \sin 2\pi\frac{\delta T}{T}n,$$

which is a sinusoid of period $\sim T/\delta T$. Hence in the case worked out above, a spurious signal in σ_z with period ~ 100 yr would be expected as a result of the choice of sampling frequency. This period would increase with time as the mean planetesimal separation increases, to as much as ~ 200 yr in the present case, in rough agreement with the plot shown in Fig. 5.1. Spurious oscillations are not seen in σ_x or σ_y, most likely because interparticle gravity is more evenly balanced parallel to the orbital plane than perpendicular to it, especially in the presence of surrounding ghost particles.

It should be noted that the *amplitude* of the oscillations (both the fundamental oscillation and its spurious counterpart) have been found to decrease with increasing N. This is due to the fact that the randomness of the planetesimal distribution improves as more particles are added to the system. Ideally, the velocity *dispersions* should initially be free of all systematic oscillations resulting from the 1 yr period, regardless of whether or not interparticle gravity is included, since a purely random distribution should give uncorrelated positions and velocities. Individual particles will exhibit the oscillations, but there should be no net periodicity (see below however). Whether or not some degree of coherence can develop later in the simulations due to some other mechanism has not been investigated, but it is felt that the sampling phenomenon adequately describes systematic oscillations seen to date. In any case, it would be difficult to distinguish between true long-period oscillations and the spurious oscillation without an improved description of the quasi-periodicity introduced by particle perturbations.

The existence, at least initially, of a net 1 yr oscillation in the velocity dispersions may be explained by considering the solution for z in the equations of motion for the case of no interparticle forces. The solution in z is a simple harmonic oscillator which can be written in the form:

$$z = \alpha\cos(\Omega t - \phi)$$

where α and ϕ are constants of integration. Note that time is being measured in units of 2π. Differentiating,

$$\dot{z} = -\alpha\Omega\sin(\Omega t - \phi).$$

Assuming equal masses for simplicity, the velocity dispersion in z is given by the superposition of N harmonic oscillators:

$$\begin{aligned} \sigma_z^2 &= \frac{1}{N}\sum_j \dot{z}_j{}^2, \\ &= \frac{1}{N}\sum_j \alpha_j^2\Omega^2\sin^2(\Omega t - \phi_j), \end{aligned}$$

where N is the total number of particles. Use the identity:

$$\sin^2 x = \frac{1}{2}(1 - \cos 2x)$$

and introduce complex notation to give:

$$\sum_j \alpha_j^2 \cos(2\Omega t - 2\phi_j) = \Re\left[\underbrace{\left(\sum_j \alpha_j^2 \mathrm{e}^{-i2\phi_j}\right)}_{A\mathrm{e}^{i\theta}} \mathrm{e}^{i2\Omega t}\right],$$

where

$$\begin{aligned} A^2 &= \left(\sum_j \alpha_j^2 \cos 2\phi_j\right)^2 + \left(\sum_j \alpha_j^2 \sin 2\phi_j\right)^2 \leq \left(\sum_j \alpha_j^2\right)^2, \\ \tan\theta &= -\frac{\sum_j \alpha_j^2 \sin 2\phi_j}{\sum_j \alpha_j^2 \cos 2\phi_j}. \end{aligned}$$

Taking the real part,

$$\sigma_z^2 = \frac{\Omega^2}{2N}\left[\sum_j \alpha_j^2 - A\cos(2\Omega t + \theta)\right],$$

which gives rise to a 0.5 yr oscillation in the dispersion. The magnitude of the initial oscillation can be minimized by choosing sufficiently large N (a few hundred or more). For example, if $\alpha_j \in [0, 1)$, then:

$$\lim_{N\to\infty} \sum_j \alpha_j^2 = \frac{N}{3},$$

whereas $A \to 0$ as $N \to \infty$. This fact is used when choosing initial values of z_j and $\dot{z}_j$ (see Appendix).

5.3 Planetary Rings

Although the initial aim of `box_tree` was to investigate planetesimal dynamics in the early solar system, it is possible to apply the code to a different kind of planetesimal dynamics, namely planetary rings. After all, `box_tree` is based on the box method used by WT in their study of Saturn's rings. But `box_tree` offers several enhancements over these initial studies, most importantly "true" interparticle gravity and spinning particles. It was decided, therefore, to return to the planetary ring problem with these new tools in hand. The remainder of the chapter is devoted to this investigation and the interesting results derived from it.

5.3.1 Model Parameters

Six models were investigated, the first three reproducing the earlier WT results as a check, and the remainder exploring the new regimes of self-gravity, mass ranges, and particle spin. An attempt was made to conform as much as possible to the initial conditions and analysis method used in WT. In particular: box sizes were fixed by the number of particles and the desired optical depth; initial particle positions were chosen randomly within the box, with a uniform distribution in z up to a preset distance above and below the plane; particles were placed in pairs symmetrically (mass-weighted in the case of non-uniform sizes — see §3.3.1) to make the centre of mass coincide with the origin; particle velocities were chosen randomly in each coordinate with a uniform distribution up to Ω times the maximum particle radius, with the velocity of the last particle (the largest particle in the case of a mass distribution) being set so that the centre-of-mass velocity was zero. Initial disk thicknesses varied for the models, initially ten particle radii but later fifteen when problems developed in packing the box. Generally, initial disk thickness is not important so long as the initial configuration is not too close to equilibrium. Most runs consisted of 30 orbits of 50 central particles (slightly more particles than were used in WT). Collection of statistics was performed at fixed intervals, typically every tenth of an orbit. Quantities measured included the CPU time, the number of collisions, the velocity dispersion in each coordinate, the filling factor at the midplane ($z = 0$), the mean free path, the local viscosity, and the vertical particle distribution. Non-local viscosity data were collected for every tenth collision. Equilibrium properties were determined by estimating the onset of equilibrium from a plot of the number of collisions per particle per orbit, and averaging the desired quantities over the equilibrium interval. Errors reported are the standard deviation of the mean. It was found that a simple estimate of equilibrium onset is sufficient; a precise determination makes little difference to the results. Note that the WT technique of inhibiting the "sliding phase"—where two or more particles come to rest and begin to roll around one another (Petit & Hénon 1987)—was also incorporated into `box_tree`.

Non-uniform particle sizes complicate the definitions of some of the basic model parameters used in WT. For example, the dynamical optical depth τ (which fixes the box size) is defined by:

$$\tau \equiv \frac{\sum_{i=1}^{N} \pi R_i^2}{L^2}. \tag{5.2}$$

If necessary, the sum in equation (5.2) can be estimated using the technique described in §3.3.1. New expressions for the local and non-local viscosity must also be derived for the case of non-uniform particle sizes. Using the notation of WT, the local viscosity (LV) is given by:

$$\overline{\nu}_{\mathrm{L}} = \frac{3}{2\Omega} \frac{< \sum_i m_i \dot{x}_i \dot{y}_{\mathrm{r},i} >_t}{\sum_i m_i}, \tag{5.3}$$

where $\dot{x}_i$ is the radial speed of particle i, $\dot{y}_{\mathrm{r},i} \equiv \dot{y}_i + \frac{1}{2}\Omega x_i$ is the tangential speed relative to the mean shear at x_i, and $< \cdot >_t$ denotes a time average measured from the onset of equilibrium to the end of the run. The non-local viscosity (NLV), which arises from collisional transport of angular momentum, is given by:

$$\overline{\nu}_{\mathrm{NL}} = \frac{3}{2\Omega} \frac{1}{N\Delta T} \frac{\sum m_>(x_> - x_<)\Delta\dot{y}_>}{\overline{m}_>}, \tag{5.4}$$

where the subscripts "<" and ">" denote particles with $x < x_0$ and $x > x_0$, respectively, where x_0 is the radial coordinate of the impact point (that is, the surface contact point between the two particles), $\Delta\dot{y}_>$ is the change in y-velocity of the particle with $x > x_0$, and the sum is over all collisions between the onset of equilibrium and the end of the run, a total time ΔT.

For the first five models, twenty values of τ were used, ranging from 0.2 to 4.0 in steps of 0.2 (WT studied $0.2 \leq \tau \leq 3.0$). A few τ values were chosen in each model for more detailed study. The models are:

Table 5.1: Some equilibrium values of σ_z for models (i) & (ii).

Model	τ	σ_z (cm/s)	WT σ_z (cm/s)
(i)	0.2	0.0454 ± 0.0004	0.0450 ± 0.0007
(i)	1.0	0.0294 ± 0.0002	0.0292 ± 0.0003
(ii)	1.0	0.0217 ± 0.0001	0.0218 ± 0.0003
(ii)	2.0	0.0187 ± 0.0001	0.0193 ± 0.0002

(i) velocity-dependent (normal) coefficient of restitution as obtained from experiment by Bridges, Hatzes & Lin (1984): $\epsilon_n(v_n) = \min[0.34 v_n^{-0.234}, 1]$, with v_n in cm/s (note the factor of 0.34 is actually given as 0.32 in the original Bridges et al. paper).

(ii) velocity-independent restitution coefficient $\epsilon_n = 0.5$.

(iii) $\epsilon_n = 0.5$ as for model (ii) with mean self-gravity vertical frequency enhancement $g = 3.6$ such that $\ddot{z} = \mathcal{F}_z - g^2\Omega^2 z$ in equation (2.7).

(iv) velocity-dependent ϵ_n as for model (i) with full interparticle gravity using the tree code with $\theta_C = 0.6$ and monopole and quadrupole time-step coefficients of 0.001 and 0.01, respectively.

(v) velocity-dependent ϵ_n as for model (i), velocity-independent transverse restitution coefficient $\epsilon_t = 0.5$, full interparticle gravity, and a smooth size distribution (cf. §3.3.1) with $\alpha^\star = -3$ ($\alpha = -\frac{5}{3}$). The exponent was chosen on the basis of observational data obtained by Voyager (Cuzzi et al. 1984). For this preliminary model, a conservative size range of 0.5–1 m was chosen (the observed range is ~1 cm–5 m for the R^{-3} power law).

(vi) as for model (v) but with several test cases for larger values of N and greater size ranges, including a comparison with work by Salo (1992b). Only a few values of τ were investigated, owing to the significant CPU expense of these models.

Particle radii of $R = 100$ cm and a transverse coefficient of restitution $\epsilon_t = 1.0$ were used for all models except models (v) & (vi). The density of water ice ($\rho = 1$ g/cm^3) was used for the particles in models (iv) and (v), and $\rho = 0.9$ g/cm^3 was used for comparisons with Salo (1992b) in model (vi). Models (i), (ii), and (iii) had a time-step coefficient $\eta = 0.05$ in equation (3.6) (§3.5.1) while models (iv), (v), and (vi) used $\eta = 0.005$ in equation (3.7). Models (i) and (ii) had $\varepsilon = 0$ while model (iii) used $\varepsilon = 0.99$. Any other exceptions will be noted as appropriate.

5.3.2 Models (i)–(iii): Comparison With WT

Table 5.1 gives equilibrium z velocity dispersions (σ_z) for two dynamical optical depths in model (i) and two in model (ii). The fourth column lists the equilibrium values found by WT (these are the only numerical values available from the paper; most results were presented graphically). The agreement is excellent and demonstrates the stability of the technique. The errors in the WT data are consistently larger, presumably due to the fact that fewer particles and a shorter integration time were used.

A vertical distribution histogram is given in Fig. 5.8 for model (i) with $\tau = 0.2$

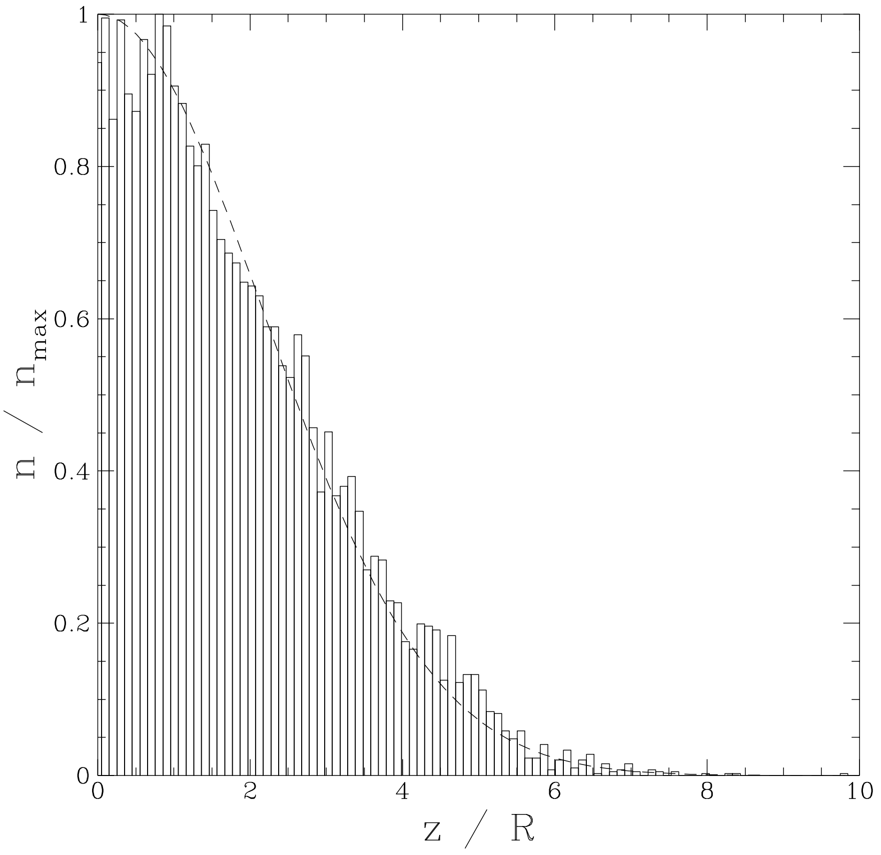


Figure 5.8: Histogram of the relative particle number density as a function of height above and below the $z = 0$ plane averaged over the equilibrium interval for model (i) with $\tau = 0.2$. The curve represents equation (5.5).

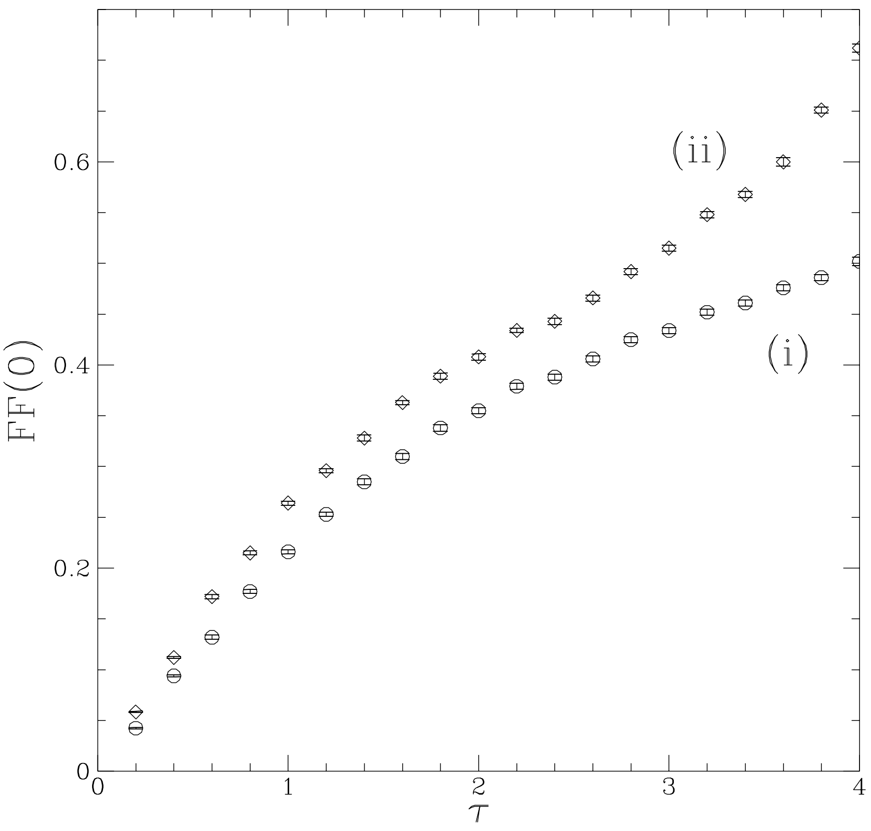


Figure 5.9: Filling factor at the midplane versus optical depth for models (i) (circles) and (ii) (diamonds). Note the turn-up in the model (ii) curve for $\tau > 2.5$.

(compare with Fig. 3 of WT). The dashed curve is the analytical model of Goldreich and Tremaine (1978):

$$n(z) = n_{\max} \exp\left(-\frac{1}{2} g^2 \Omega^2 / \sigma_z^2\right), \tag{5.5}$$

which is a good approximation at low optical depth. It can be seen that `box_tree` reproduces the expected behaviour quite well.

Figure 5.9 is a plot of the filling factor at the midplane FF(0) versus dynamical optical depth for models (i) and (ii). This should be compared with Fig. 15 of WT. Note the change in behaviour of the two curves for $\tau \gtrsim 2.5$: the model (i) curve seems to have reached a constant slope while the model (ii) curve bends upward. This behaviour is suggested in the WT data, but the extended range in τ in Fig. 5.9 allows the effect to be seen more clearly. The behaviour can be understood by noting that the velocity-dependent coefficient of restitution makes collisions more elastic in model (i) as the velocity dispersion decreases, i.e. at higher optical depth. The filling factor still increases because the particles become more confined with optical depth, but the increase levels out as the collisions become more elastic. The constant coefficient of restitution in model (ii) allows the particles to bunch more strongly together with increasing τ, beyond the confinement effect.

Comparison with model (iii) begins with a plot of the Cartesian components of the velocity dispersion (Fig. 5.10). This plot in analogous to Fig. 17 of WT. The dispersions are even more nearly equal in the `box_tree` run, emphasizing the fluid nature of the ring

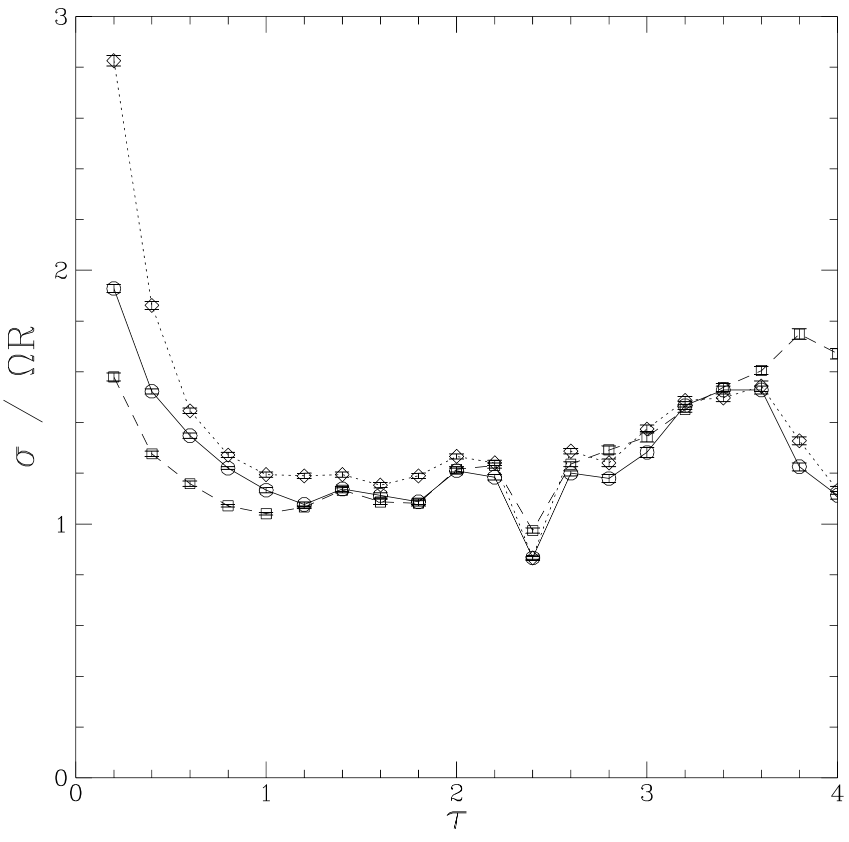


Figure 5.10: Cartesian components of the equilibrium velocity dispersion versus optical depth for model (iii). The (radial) x-component is traced by diamonds (dotted line), y by squares (dashed line), and z by circles (solid line). Note the event near τ = 2.4; this region is magnified in Fig. 5.12(a).

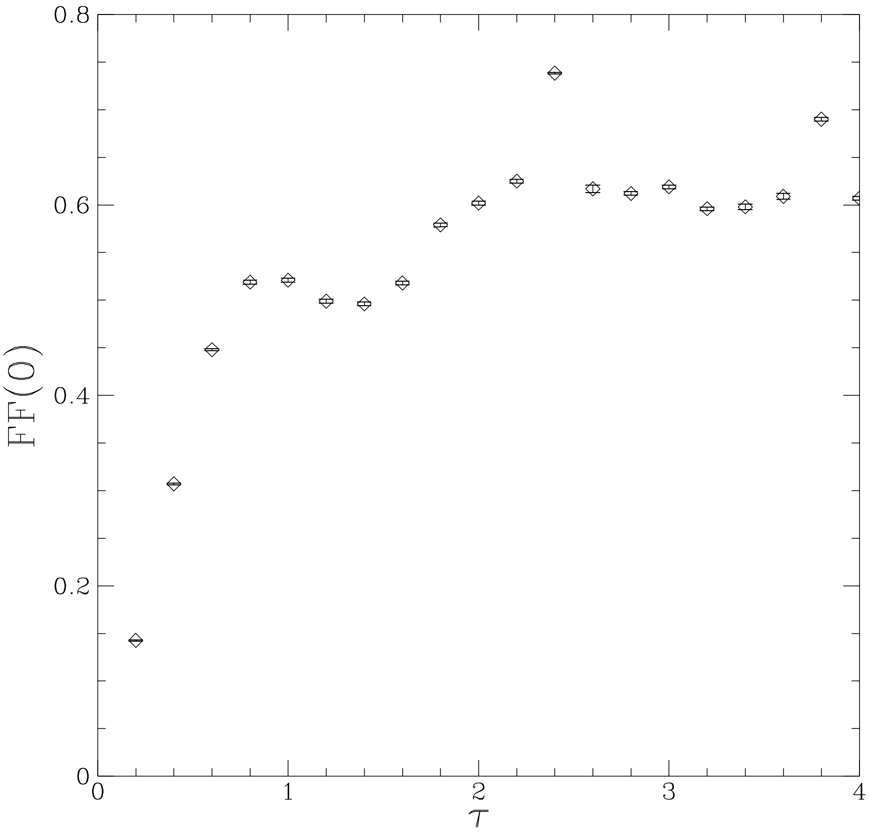


Figure 5.11: Filling factor at the midplane versus optical depth for model (iii). A blow up of the region near $\tau = 2.4$ is shown in Fig. 5.12(b).

in this model. The splitting at $\tau \gtrsim 3.6$ is not understood. The most intriguing feature of the plot, however, is the strong downward spike near $\tau = 2.4$, which is not seen in the WT data. A corresponding feature is also seen in a plot of the filling factor (Fig. 5.11; compare with Fig. 20 of WT). Blowups of the regions are shown in Fig. 5.12(a) and (b). From the magnified plots, it would appear that some kind of critical point lies in the region $2.45 \lesssim \tau \lesssim 2.55$. Figure 5.13 gives the vertical distribution for $\tau = 2.5$. The histogram shows that the system has developed almost perfect stratification, with three equally-populated layers in the middle and two strongly underpopulated layers on the outside. This can be understood by noting that $N = 50$ and $\tau \simeq 2.5$ give a box size of $s \approx 8R$ in equation (5.2). This would result in 4×4 packing on three levels, with two particles left over, presumably one either side. Note that for close-packed sheets of spherical particles, the spheres on one level fill the gaps between spheres on neighbouring levels, hence the peaks in Fig. 5.13 do not occur at exact multiples of the particle size (except for the plane of particles at $z = 0$). Figure 5.14 illustrates what close packing actually looks like as seen looking down on the shearing plane. It should be emphasized that such layering is probably unphysical as it depends on the ratio of the particle size to the (arbitrary) box size.

Other critical values of τ can be estimated by choosing positive integers n such that $s = 2nR$ and $n^2 \leq N$. For $N = 50$, there are four possible critical points in the range $0.2 \leq \tau \leq 4.0$, namely 2.45 (3 layers, 2 particles left over), 1.57 (2 layers, 0 left over), 1.09 (1 layer, 14 left over), and 0.80 (1 layer, 1 left over). The number density would be expected to nearly vanish at $z = 0$ for an even number of layers since the particles

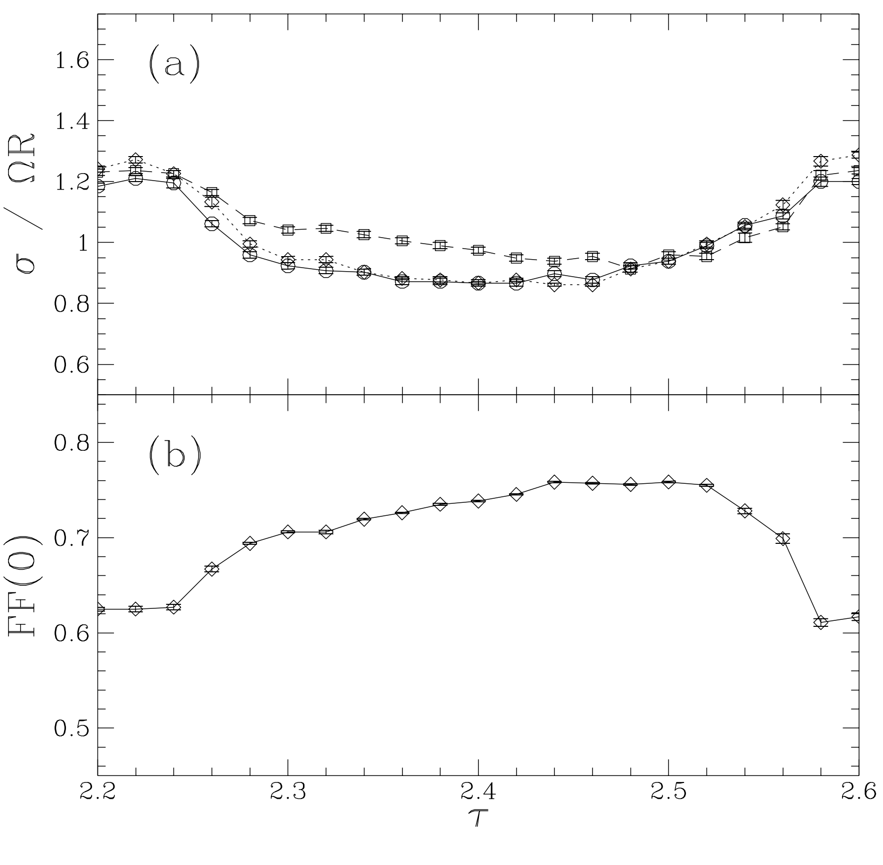


Figure 5.12: Blowups of the $2.2 \leq \tau \leq 2.6$ regions for the velocity dispersions (a) and filling factor (b) of model (iii). The transition is rapid but smooth.

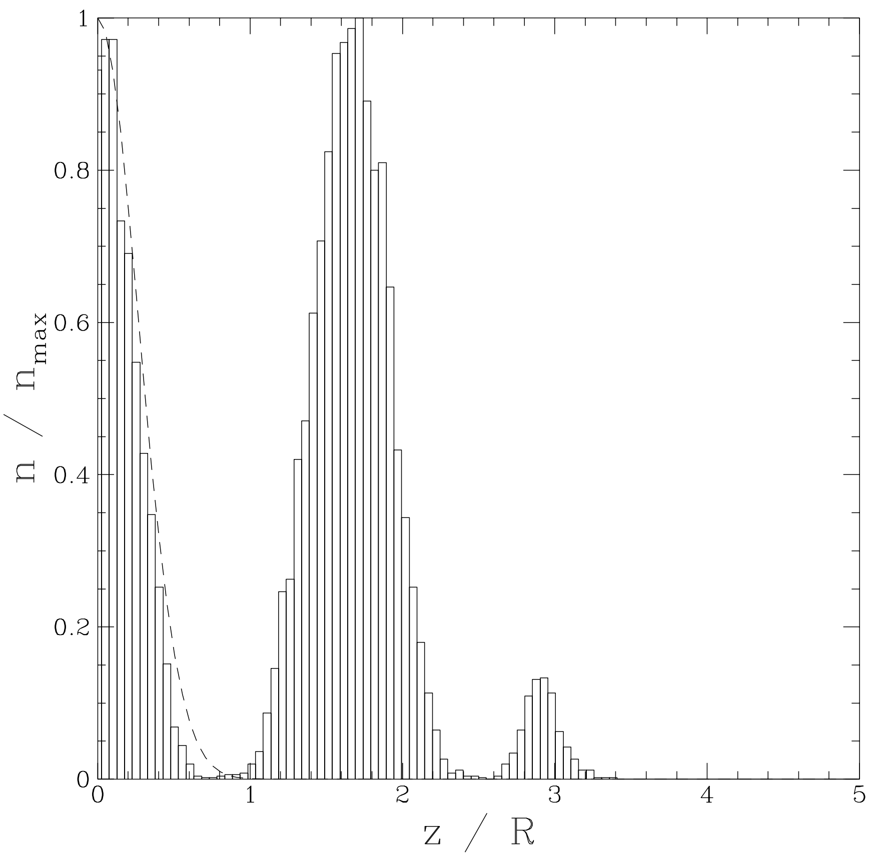


Figure 5.13: Vertical distribution of particles for model (iii) with $\tau = 2.5$. Five stratified layers have developed at this critical τ value.

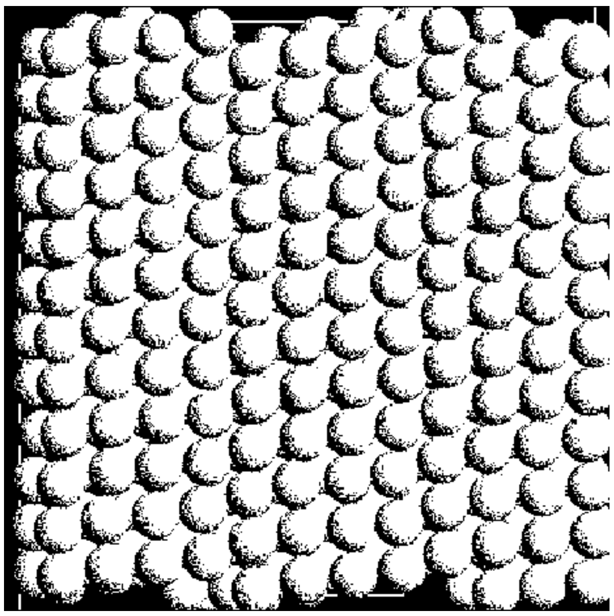


Figure 5.14: View looking down onto the ring plane for model (iii), $\tau = 2.5$, during the equilibrium phase. The central box plus all 8 ghost boxes are shown (450 particles in all, most hidden behind the top layer). At this critical τ value, several stratified sheets have formed, with particles lined up in y (bottom to top) to minimize resistance to the shearing flow.

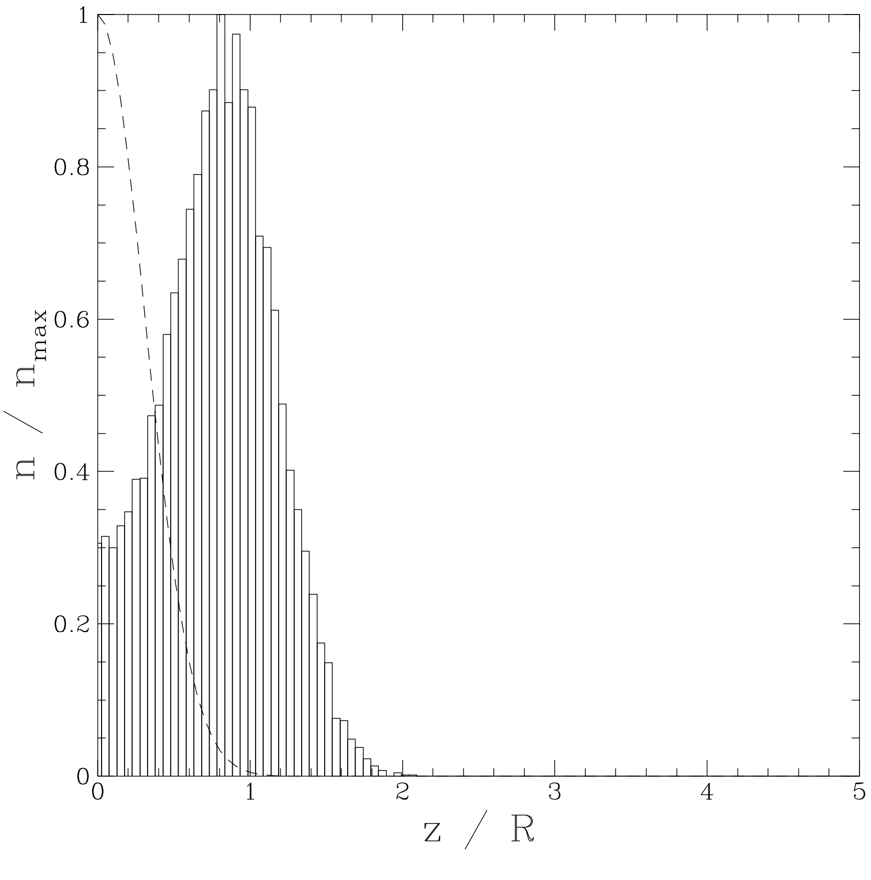


Figure 5.15: Number density histogram for model (iii) with $\tau = 1.3$. Though this value of τ gives a local minimum in FF(0) (cf. Fig. 5.11), the midplane is still moderately populated.

would straddle the midplane, rather than lie completely within it. This behaviour only partly explains the apparent oscillations in FF(0) for model (iii) however, because a local minimum occurs at $\tau \sim 1.3$, not $\tau = 1.57$ as might be expected (although a test was performed to confirm that $n(0)$ does indeed vanish for $\tau = 1.57$). A number density histogram for $\tau = 1.3$ shows the $z = 0$ level to be moderately populated (Fig. 5.15; compare with Fig. 23 of WT). This discrepancy comes about because the overall increasing trend in FF(0) causes the critical values of τ to be shifted somewhat from their expected values.

5.3.3 Model (iv): Improved Gravity Model

The first step towards a more realistic planetary ring simulation is the inclusion of full interparticle gravity. As a test of the reliability of the force calculations, the entire $0.2 \leq \tau \leq 4.0$ range was tested as before. Note that model (iv) lies somewhere between model (iii) and the zero gravity case: the enhancement of the vertical frequency in model (iii) represented the contribution of the entire disk, whereas in model (iv) only the gravity of the central particles and their ghosts is included. Compensation for this could be made by increasing the particle density ρ, or by introducing a smaller z frequency enhancement than was used in model (iii), but such detail was not considered important for these experiments. The particle number N was kept at 50: a plot of equilibrium velocity dispersion versus particle number was found to be essentially flat for $N \gtrsim 20$ (cf. Fig. 1

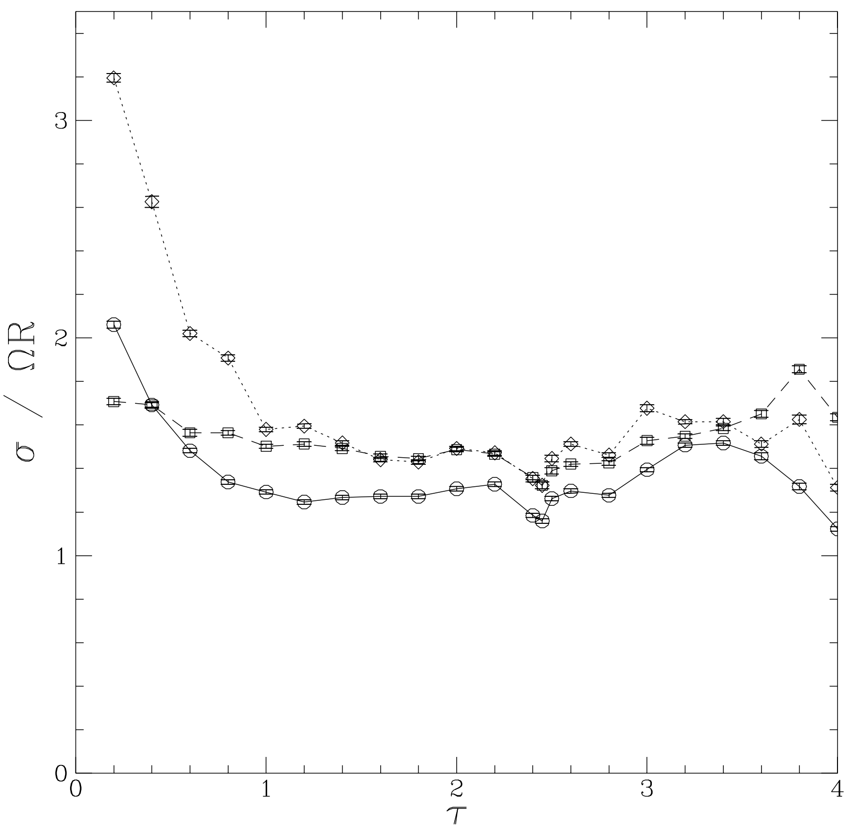


Figure 5.16: Equilibrium velocity dispersions for model (iv).

of WT).

Figure 5.16 shows the equilibrium velocity dispersions for model (iv) (compare with Fig. 5.10). Note that σ_z remains consistently smaller than σ_x and σ_y in this case. The blip near $\tau = 2.5$ still occurs, but with reduced significance. However, the filling factor (Fig. 5.17) shows that close packing still occurs to some extent in this model, though not as strongly. This behaviour is consistent with the reduced nature of the gravity enhancement compared with model (iii). The close packing is also noticeable in a plot of $\tau\overline{\nu}$ versus τ (Fig. 5.18), where the slope goes negative in the region of $\tau = 2.5$, indicating a viscous instability associated with the (unphysical) close packing. With the exception of another unphysical point at $\tau = 4.0$, the slope of $\tau\overline{\nu}$ remains positive. Lastly, Fig. 5.19 shows the vertical distributions for four values of τ, clearly demonstrating that stratification persists in this model.

These models show that inclusion of particle self-gravity yields results that are similar to the mean self-gravity model. As will be shown below, however, full interparticle gravity becomes much more important once an initial size distribution is introduced into the system.

5.3.4 Model (v): Size Distribution Model with Rough Spheres

Size distributions complicate the dynamics of numerical simulations considerably in the presence of interparticle gravity. As the system evolves, the largest particles gravitationally excite the smaller ones, making dynamical equilibrium harder to attain. The problem becomes more acute at low optical depth where there is little collisional dissipation. At

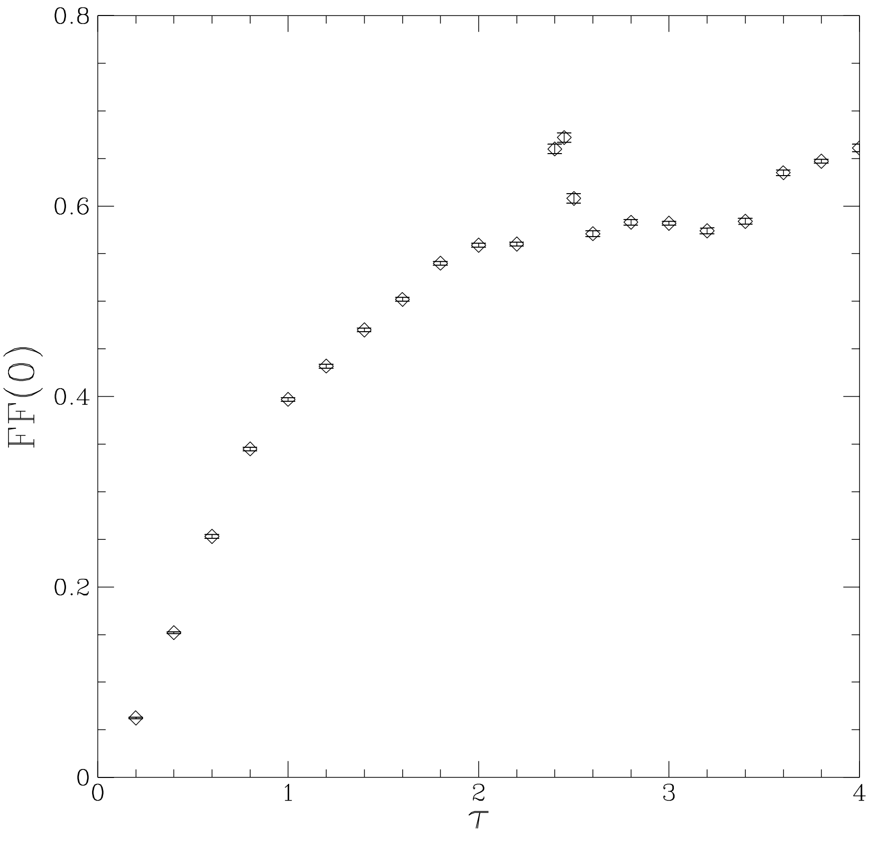


Figure 5.17: Midplane filling factor for model (iv).

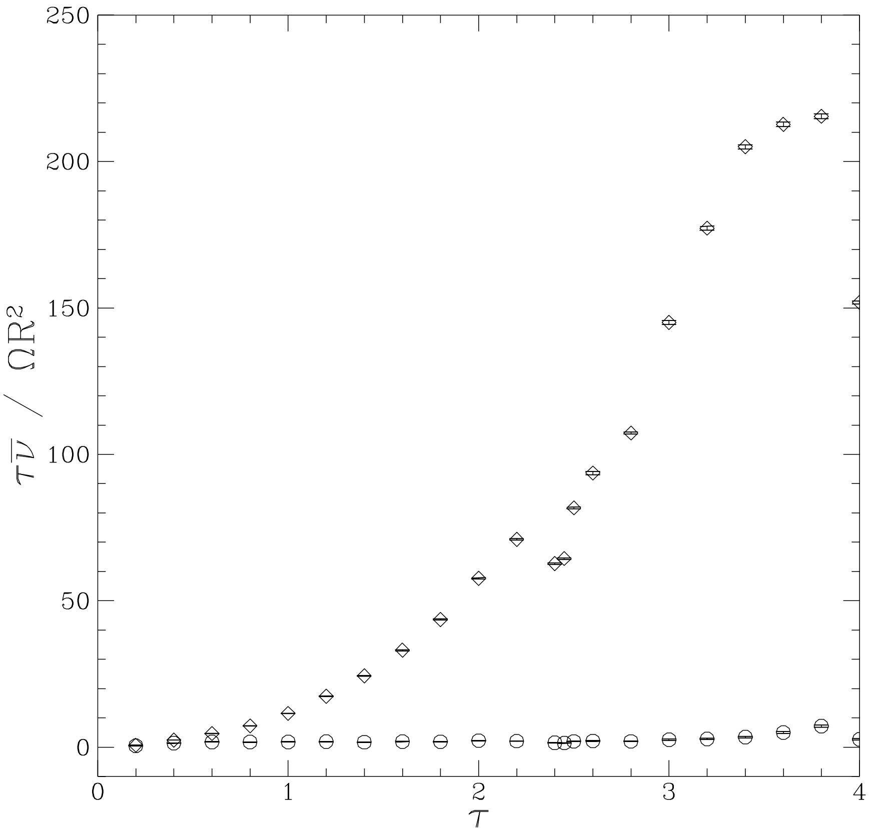


Figure 5.18: Dimensionless height-averaged kinematic viscosity times τ, versus τ for model (iv). The local contribution (circles) is much smaller than the total contribution (diamonds) for moderate to large τ. The slope changes sign near the critical point $\tau \sim 2.5$, indicating a regime of viscous instability.

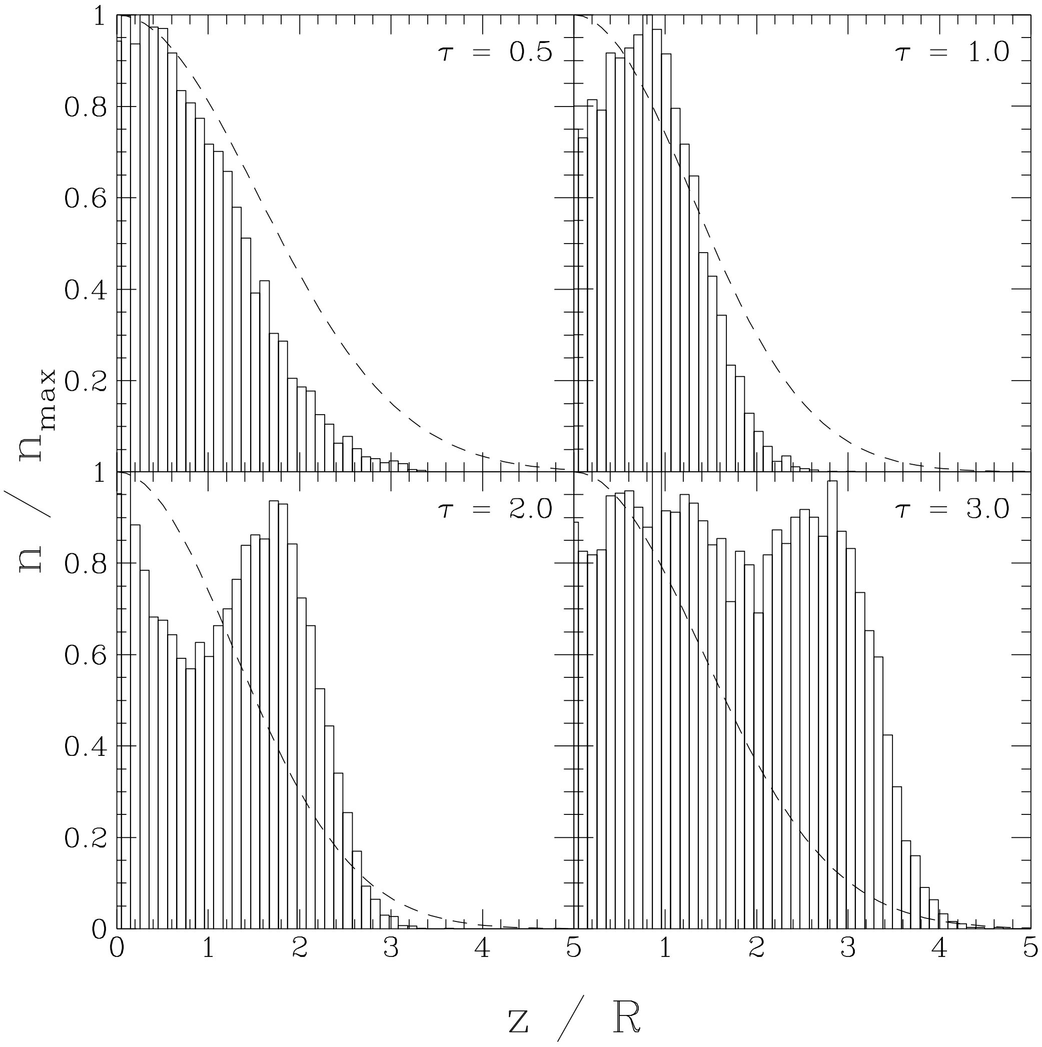


Figure 5.19: Number density vs. $|z|$ histograms for model (iv) for τ = 0.5, 1.0, 2.0, and 3.0. Stratification is still seen in this model.

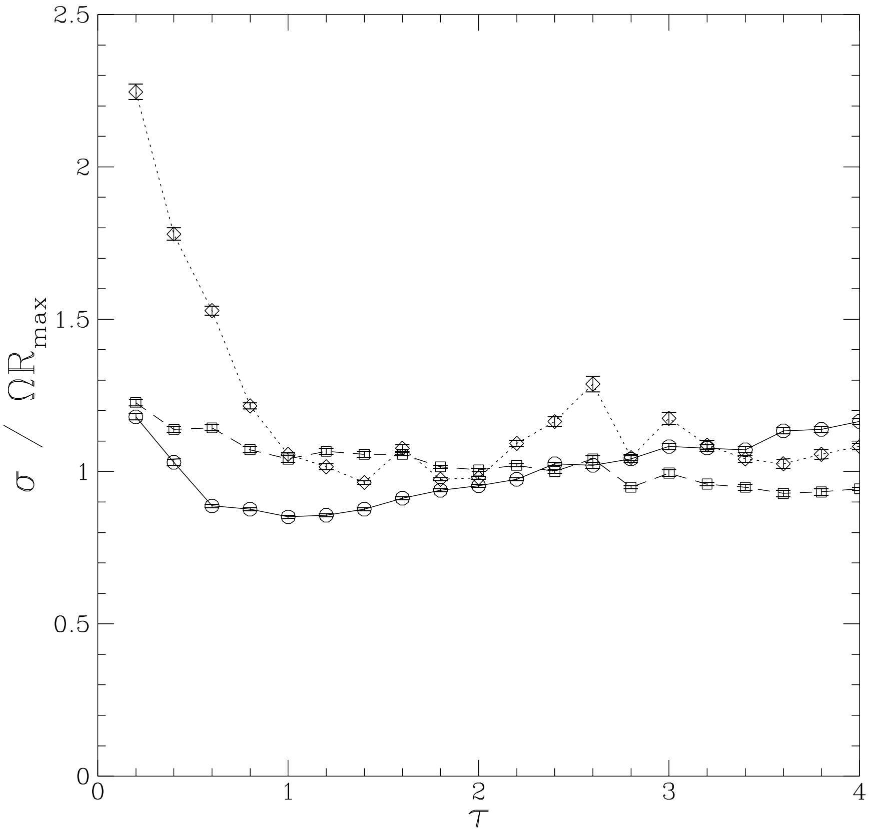


Figure 5.20: Equilibrium velocity dispersions for model (v).

higher optical depths, aggregates may form and further complicate the dynamics. Another problem is posed by the nature of the power law distribution itself. As pointed out by Salo (1992a), N must be chosen large enough to provide statistically sound sampling of the size distribution. These aspects will be examined in greater detail in §5.3.5.

As a first test, and to provide a bridge between WT simulations and later work by Salo, a series of runs in the spirit of model (iv) was performed for the conservative size range $\Delta R = 50$–100 cm. Although the size only varies by a factor of 2, the mass varies by a factor of 8, providing a good range of gravitational forces for this preliminary model. In addition, surface friction was incorporated in the form of a constant tangential coefficient of restitution, $\epsilon_t = 0.5$, in order to investigate the equilibrium distribution of particle spins. All spins were initially zero with respect to the local frame. The choice of $N = 50$ was checked in the usual way by verifying that the equilibrium velocity dispersion varied little for $N \in [10, 100]$ with $\tau = 1$; $N = 50$ also provides adequate sampling of the size distribution given the limited range.

Figures 5.20–5.22 show the usual statistical quantities for this model. The velocity dispersions (Fig. 5.20) behave much the same as for model (iv) (Fig. 5.16; note the change in scale). The midplane filling factor (Fig. 5.21) is seen to attain and sustain a higher level, consistent with the improved packing efficiency of a size distribution. The curve is quite smooth with the exception of an unexplained blip at $\tau \sim 3.3$. The viscosity curve (Fig. 5.22) is very smooth and shows no evidence of viscous instability in the range $0.2 \leq \tau \leq 4$. Note that the size distribution effectively eliminates the "crystalization" phenomenon found in models (iii) and (iv) for certain critical values of τ.

Figures 5.23–5.25 give the height, spin, and obliquity distributions for $\tau = 0.5$, 1.0,

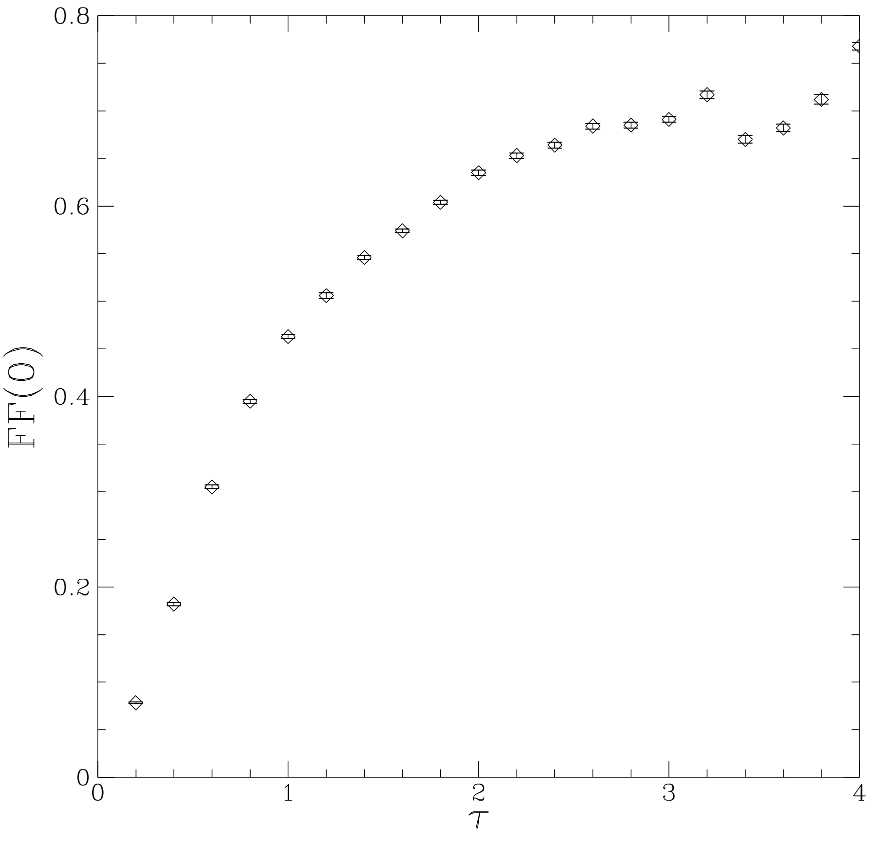


Figure 5.21: Midplane filling factor for model (v).

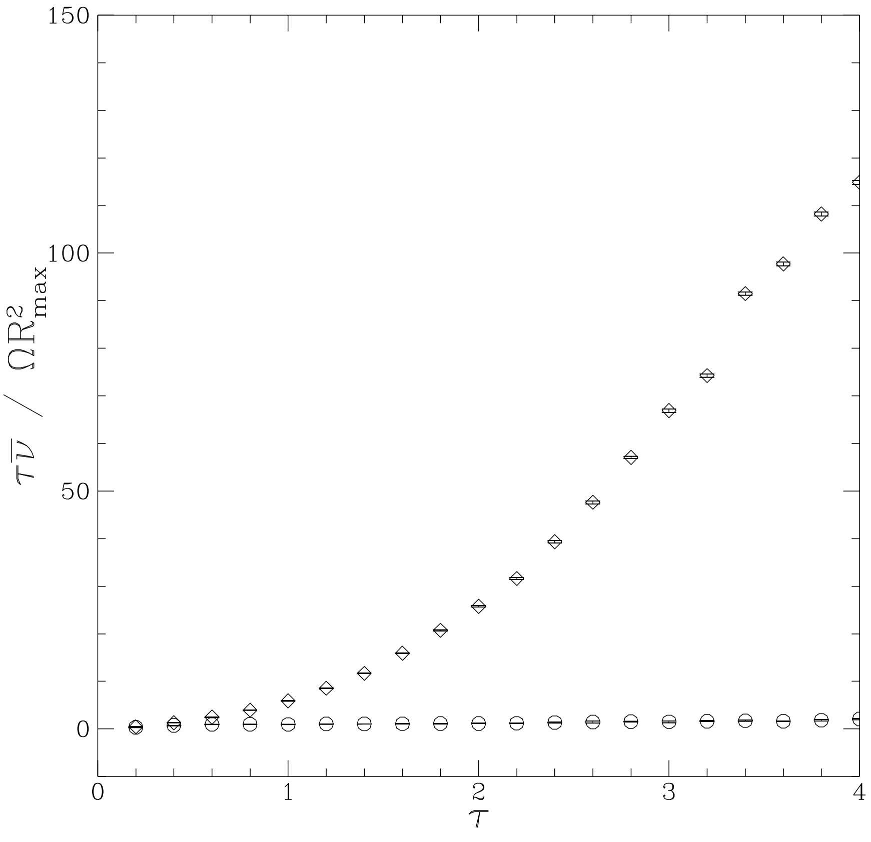


Figure 5.22: The quantity $\tau\overline{\nu}$ vs. τ for model (v). There is no evidence of viscous instability.

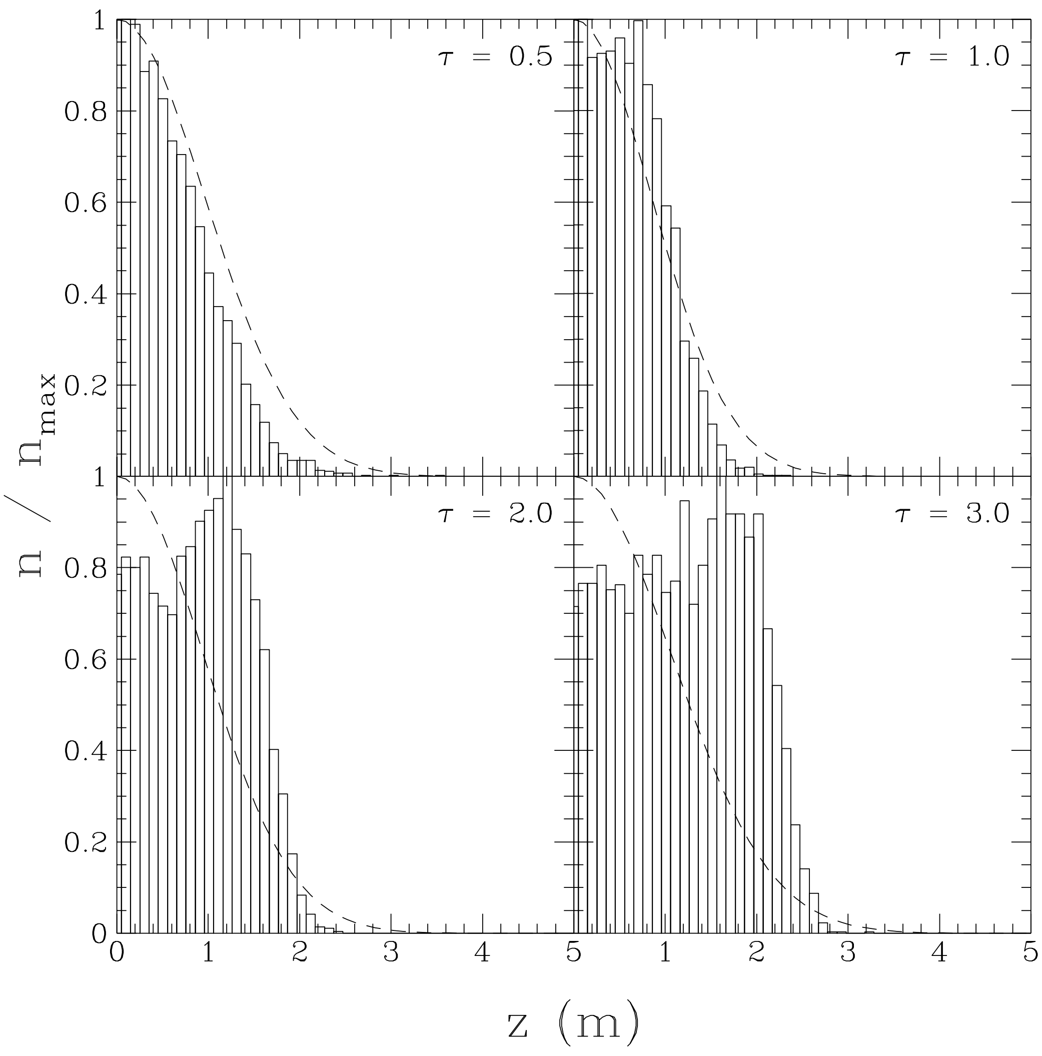


Figure 5.23: Histograms of number density vs. $|z|$ for model (v) and four values of τ. The particles continue to pile up at high optical depth but layering is not as well defined since smaller particles can start filling the gaps between larger particles.

2.0, and 3.0. The vertical distribution (Fig. 5.23) is similar to Fig. 5.19, though the dips are not as well defined and the mean height is smaller due to the tighter packing. The spin distribution (Fig. 5.24) evidently varies little with optical depth. The dashed curve superimposed on each histogram is given by:

$$n(\omega) = \omega e^{-\omega}. \tag{5.6}$$

This curve is drawn for reference only, without physical justification. Note that in all cases each particle has suffered at least one collision, so there are no zero spins remaining. The distribution in obliquity, or the angle ψ between the spin axis $\hat{\boldsymbol{\omega}}$ and the positive z-axis $\hat{\boldsymbol{z}}$ (Fig. 5.25), also varies little with optical depth, but shows a strong tendency toward retrograde spin ($\psi > 90°$). Recall that all spin properties are with respect to the local (rotating) frame of reference; in the fixed frame, the spins would be slightly prograde on average.

Further aspects of this model with be presented in the following section.

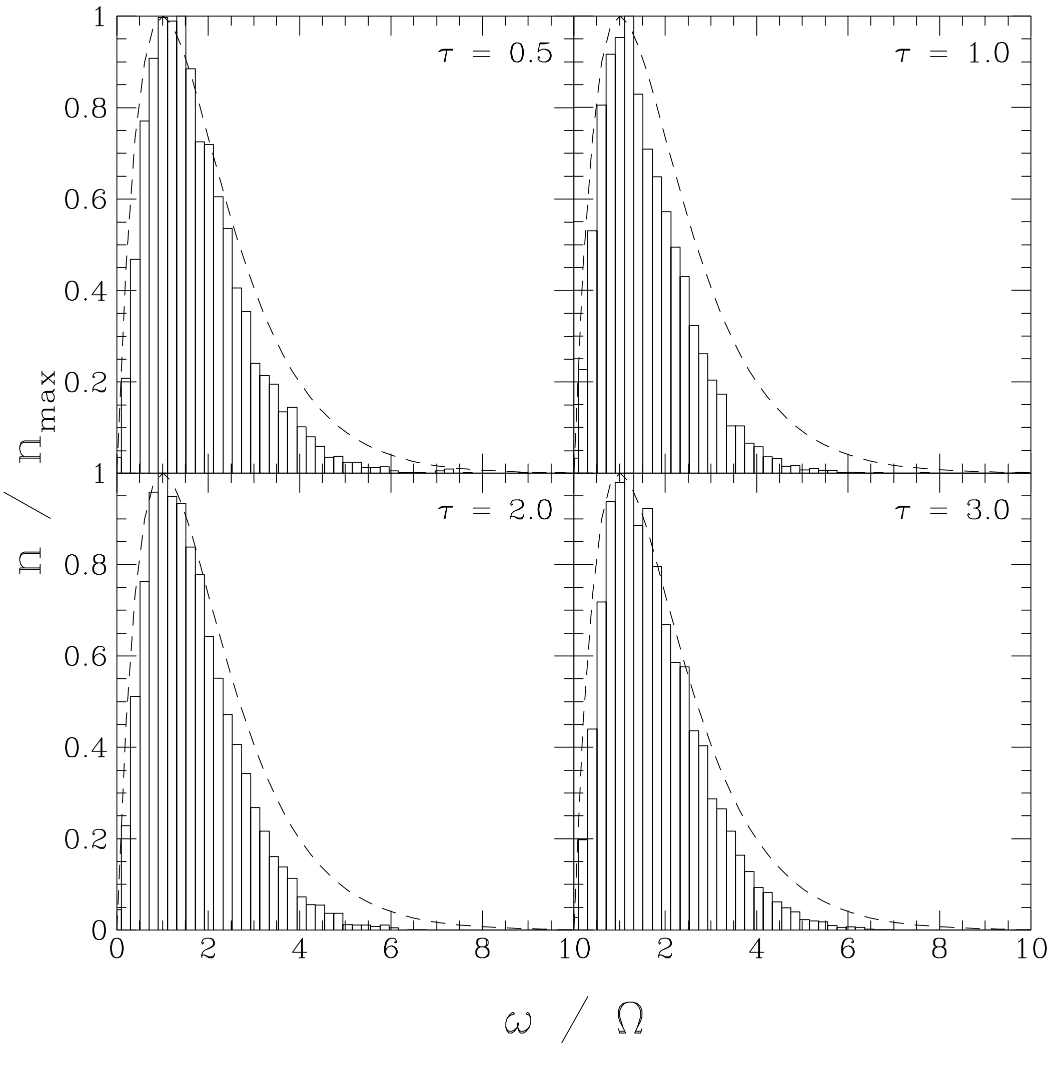


Figure 5.24: Relative number density as a function of spin magnitude ω for model (v). The dashed curve is given by equation (5.6). There is little dependence on optical depth for this model.

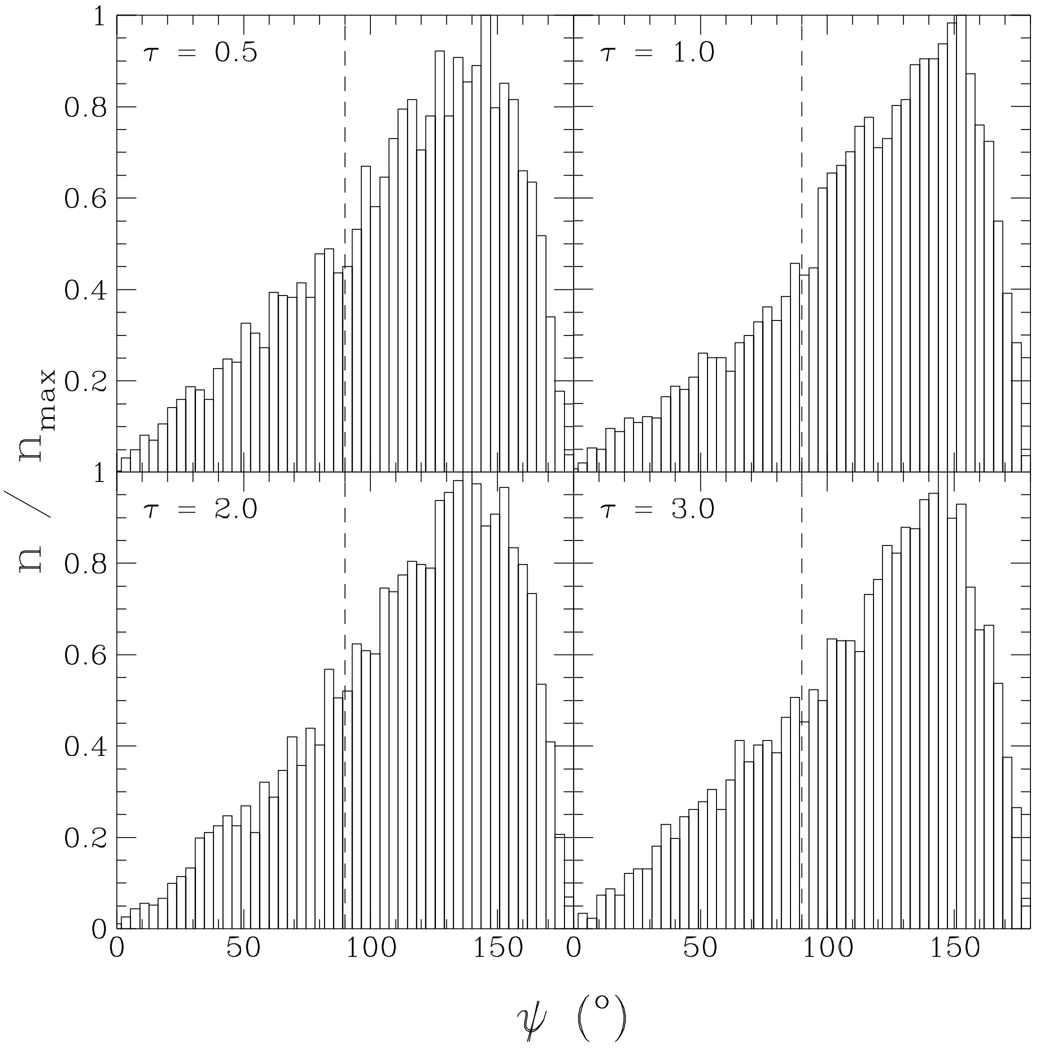


Figure 5.25: Relative number density as a function of spin obliquity (angle between spin and orbital momentum vectors) for model (v). Again there is little dependence on τ. The plots show most particles are spinning retrograde with respect to their orbital motion when viewed from the rotating frame.

Table 5.2: Binned equilibrium data for models (v) and (vi).

Model	N	ΔR (m)	τ	ΔT (orb)	FF(0)	N_b	$\mu^\star$ (Ω)	$\overline{\omega}$ (Ω)	$\overline{\psi}$ ($^\circ$)	$\overline{\lvert z\rvert}$ (m)	$\overline{v}$ (cm/s)
						20	0.37	2.47 ± 0.01	105.9 ± 0.5	0.769 ± 0.007	0.0482 ± 0.0003
(v.1)	50	0.50–1.0	0.5	4–30	0.243	50	0.36	1.853 ± 0.008	113.8 ± 0.3	0.640 ± 0.004	0.0414 ± 0.0002
						4	0.32	0.98 ± 0.01	135 ± 1	0.373 ± 0.008	0.0298 ± 0.0004
						20	0.22	2.20 ± 0.01	110.7 ± 0.5	0.744 ± 0.005	0.0385 ± 0.0003
(v.2)	50	0.50–1.0	1.0	3–30	0.465	50	0.264	1.676 ± 0.007	118.7 ± 0.3	0.623 ± 0.002	0.0336 ± 0.0001
						4	0.36	0.87 ± 0.01	138.0 ± 0.9	0.389 ± 0.008	0.0245 ± 0.0003
						20	0.23	2.34 ± 0.01	108.8 ± 0.5	1.074 ± 0.007	0.0400 ± 0.0004
(v.3)	50	0.50–1.0	2.0	4–30	0.631	50	0.27	1.786 ± 0.007	116.1 ± 0.3	0.923 ± 0.002	0.0358 ± 0.0004
						4	0.36	0.92 ± 0.01	133.3 ± 0.9	0.54 ± 0.01	0.0291 ± 0.0005
						20	0.23	2.45 ± 0.02	108.2 ± 0.5	1.38 ± 0.01	0.0399 ± 0.0003
(v.4)	50	0.50–1.0	3.0	3–30	0.690	50	0.264	1.880 ± 0.008	115.4 ± 0.3	1.228 ± 0.003	0.0366 ± 0.0003
						4	0.29	0.99 ± 0.01	136.6 ± 0.9	0.82 ± 0.01	0.0306 ± 0.0004
						135	0.38	4.00 ± 0.02	99.9 ± 0.1	3.10 ± 0.01	0.2192 ± 0.0009
(vi.1)	200	1.15–5.0	1.0	3–10	0.373	200	0.432	3.22 ± 0.01	102.3 ± 0.1	2.96 ± 0.01	0.2124 ± 0.0009
						5	0.707	0.655 ± 0.006	116.8 ± 0.7	3.23 ± 0.03	0.236 ± 0.002
						326	0.19	7.39 ± 0.03	97.1 ± 0.1	4.84 ± 0.06	0.274 ± 0.001
(vi.2)	400	0.70–5.0	1.0	4–10	0.178	400	0.27	6.43 ± 0.02	97.8 ± 0.1	4.80 ± 0.06	0.269 ± 0.001
						4	0.43	1.17 ± 0.01	120.3 ± 0.5	6.25 ± 0.08	0.332 ± 0.003
						518	0.13	10.16 ± 0.03	95.60 ± 0.06	5.81 ± 0.08	0.313 ± 0.002
(vi.3)	600	0.55–5.0	1.0	3–10	0.145	600	0.20	9.13 ± 0.03	95.97 ± 0.07	5.77 ± 0.08	0.310 ± 0.002
						4	0.990	1.044 ± 0.006	88.6 ± 0.5	7.2 ± 0.1	0.368 ± 0.003
						715	0.18	12.39 ± 0.04	94.66 ± 0.08	4.13 ± 0.03	0.309 ± 0.002
(vi.4)	800	0.46–5.0	1.0	2–5	0.193	800	0.23	11.36 ± 0.04	94.99 ± 0.08	4.08 ± 0.03	0.305 ± 0.002
						4	0.86	1.217 ± 0.008	97 ± 1	7.0 ± 0.1	0.385 ± 0.004
						914	0.12	14.40 ± 0.08	94.17 ± 0.06	4.28 ± 0.04	0.302 ± 0.002
(vi.5)	1000	0.40–5.0	1.0	1–3	0.177	1000	0.16	13.40 ± 0.08	94.68 ± 0.07	4.23 ± 0.04	0.299 ± 0.002
						4	0.76	1.11 ± 0.02	102.9 ± 0.4	6.9 ± 0.1	0.376 ± 0.004
						2820	0.128	12.4 ± 0.1	94.98 ± 0.04	4.86 ± 0.04	0.341 ± 0.002
(vi.6)	3200	0.50–5.0	0.4	1–3	0.0854	3200	0.197	11.2 ± 0.1	95.27 ± 0.05	4.69 ± 0.04	0.335 ± 0.002
						16	0.979	0.76 ± 0.01	92.2 ± 0.2	5.80 ± 0.05	0.350 ± 0.003
						3109	0.27	32.1 ± 0.1	92.09 ± 0.05	5.54 ± 0.09	0.333 ± 0.003
(vi.7)	3200	0.20–5.0	1.0	0.5–1	0.13	3200	0.28	31.3 ± 0.1	92.36 ± 0.06	5.52 ± 0.09	0.331 ± 0.003
						3	0.502	0.825 ± 0.007	125.6 ± 0.1	10.5 ± 0.3	0.41 ± 0.01

5.3.5 Model (vi): Large N Models

In order to model a realistic size range at optical depths of order unity or higher in a statistically valid way, larger values of N must be used. Conversely, for a given value of N, the size range is constrained. For example, suppose a conservative box size $s = 10R_{\mathrm{max}}$ is imposed. If $\tau = 1$ and $R_{\mathrm{max}} = 5$ m, then R_{min} can be determined for a given value of N by integrating the equivalent of equation (3.1) in R. This gives:

$$R_{\mathrm{min}} \approx \frac{20}{\sqrt{N\eta_5}}, \tag{5.7}$$

where $\eta_5 \equiv \ln(5/R_{\mathrm{min}})$ and sizes are measured in metres. Equation (5.7) can be solved iteratively for R_{min}. Models (vi.1–5) and (vi.7) were obtained by choosing $N = 200$, 400, 600, 800, 1 000, and 3 200, respectively, in equation (5.7). The results are summarized in Table 5.2. Also included in the table are the four runs $\tau = 0.5$, 1.0, 2.0, and 3.0 from model (v), labeled models (v.1–4). Model (vi.6) reproduces a run by Salo (1992b) and will be discussed in more detail below. The table shows the chosen equilibrium interval ΔT for each run as well as the mean midplane filling factor (error bars omitted to conserve space). The mean spin magnitude and obliquity as seen in the local frame, the mean vertical distance from the plane, and the mean velocity magnitude are given for each run, split into three cases according to size bin. The mean z-spin as seen from the fixed *inertial* frame ($\mu^\star = \overline{\omega}_z + \Omega$) is included for comparison with Araki (1991), who found $\mu^\star \sim 0.3$ for the equal-mass case. None of the quantities are mass-weighted since the binning automatically differentiates between the low- and high-mass regimes. It was found that five size bins gave adequate sampling (usually at least four particles in the largest bin): $\Delta R_b = \frac{1}{5}\Delta R$. The top line of each set is for $R_{\min} \leq R \leq R_{\min} + \Delta R_b$, the middle line is for the complete range $R_{\min} \leq R \leq R_{\max}$, and the bottom line is for $R_{\max} - \Delta R_b \leq R \leq R_{\max}$. The number of particles in each bin is listed in the column labeled N_b.

Note that the quantity $\mu^\star$ varies between 0.15 and 0.45 when averaged over all particles in each model, and there is a suggestion that the value decreases as the size range increases, possibly due to a stronger random contribution from the increasing number of smaller particles. This is supported by the fact that the mean z-spin in the large particle bins is typically much larger than that for the small particle bins (recall that at $t = 0$, $\omega_z = 0$ and so $\mu^\star = 1$ for all particles initially). Evidently there is a preferred prograde spin of about 0.3 for the particles when viewed from the planet frame (for reference note that in such a frame, Earth's moon would have a prograde spin of unity). The other statistical quantities in the table will be discussed below within the contexts of models (v) and (vi).

The data for models (v.1–4) confirm the trends shown in Fig. 5.20–5.25. The data also show that the smallest particles have the largest spin, consistent with equipartition of rotational energy (this will be illustrated for model (vi.1) below). Interestingly, the smallest particles also have their mean spin axes most nearly orthogonal to the mean orbital axis, though the peak of the distribution is rather wide (Fig. 5.25). It should be noted, however, that when the Cartesian components of the particles' spins are averaged *separately*, the x- and y-components turn out to have means $\ll \Omega$, leaving the spin in z dominant (i.e. $\mu^\star \sim 0.3$ as discussed above). This implies that the components of the spin vectors that lie in the orbital plane are isotropically distributed. As will be illustrated below, the mean component spins for all models have Lorentz profiles. Finally, note that the smallest particles have the largest z excursion and mean velocity, consistent with energy injection by gravitational scattering off the largest particles.

Models (vi.1–5) were studied for shorter intervals because of the increased CPU expense. This increase is due primarily to the scaling with N, but there is also an increase due to inhomogeneities in the particle distribution, namely the formation of cigar-shaped clumps or aggregates. These associations tend to pack quite tightly, increasing the force derivatives on the constituent particles and resulting in shorter time-steps according to equation (3.7). Figure 5.26 illustrates one such aggregate. This snapshot of model (vi.4) was taken at $t = 3$ looking down the z-axis. Animations of the formation and evolution of these aggregates show several common features: (1) they have a characteristic *pitch angle* $\lambda \sim 30°$ measured with respect to the positive y-axis (the direction of mean orbital motion); (2) they tend to form around the largest particles; (3) they form and "dissolve" within a few fractions of an orbit; (4) they are *not* seen when interparticle gravity is switched off. Aggregate formation will be discussed further below.

Particle aggregates help explain some of the anomalies seen in Table 5.2 for models (vi.1–5), namely the fact that the largest particles now seem to slightly dominate the z excursion and mean velocity, contrary to the case of models (v.1–4). Evidently the aggregates behave like "super-particles", trapping the smaller particles and reducing their contributions to the dispersions. The larger particles, meanwhile, are strongly perturbed by the clumps, so their contributions are increased.

The mean spin as a function of particle moment of inertia for model (vi.1) is shown in Fig. 5.27. The plot demonstrates that in the equilibrium state rotational energy is distributed according to moment of inertia: smaller particles generally spin faster or have a greater range of spin energy. The dashed line shown in the plot is the isocurve $I\omega^2 = I_{\max}\Omega^2$, tracing an upper envelope to the spin distribution.

Model (vi.6) was designed to be a direct comparison with a recent result by Salo (1992b, Fig. 1, B ring). The box size, dynamical optical depth, and size distribution were

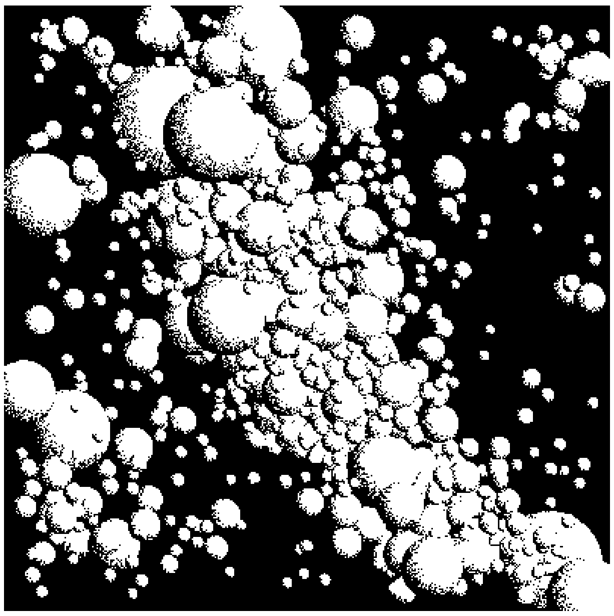


Figure 5.26: View of model (vi.4) at $t = 3$ looking along the negative z-axis. The box is 50 m on a side.

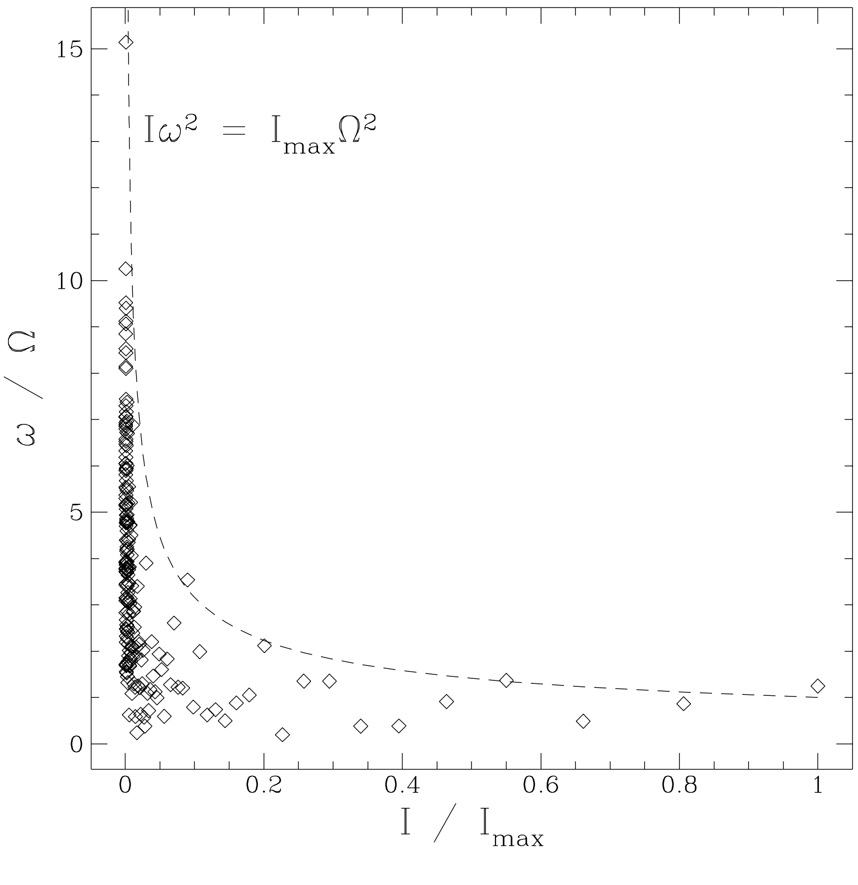


Figure 5.27: Mean spin as a function of moment of inertia for model (vi.1). The spins lie below an equipartition envelope $I\omega^2 = I_{\max}\Omega^2$.

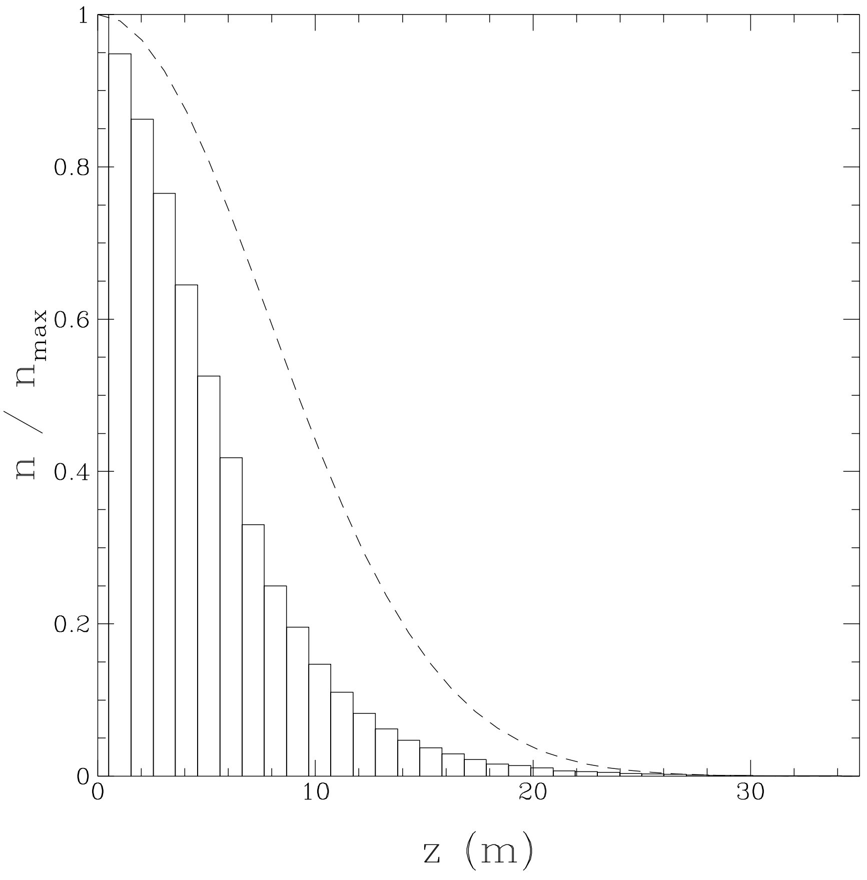


Figure 5.28: Number density vs. $|z|$ histogram for model (vi.6) after three orbits.

identical, as was the velocity-dependent (normal) coefficient of restitution. Tangential friction was included in model (vi.6), unlike the Salo model, but was not expected to affect the dynamics appreciably. The run was carried out for only 3 orbits owing to the CPU expense, but reached an acceptable equilibrium after 1 orbit. A combination of equations (3.6) and (3.7) was used for computing time-steps to improve the speed. Figure 5.28 shows a very smooth vertical distribution, packed tighter than the curve given for the theoretical equal-size case without self-gravity. The spin distributions (all three components plus the spin magnitude) are shown in Fig. 5.29. Lorentzians of the form $1/(\omega_i^2 + 10)$ have been drawn for the ω_x, ω_y, and ω_z distributions as an aid to the eye. Similar Lorentz distributions in spin components occur for all models that include spin effects. The curve for the spin magnitude distribution is equation (5.6) with ω weighted by a factor of 0.2 to approximate the actual width of the distribution. The spin rates are much higher in this model, again dominated by the smaller particles. The spin obliquities are shown in Fig. 5.30. The obliquity distribution is very broad (FWHM $\sim$ 120°), and only very slightly retrograde on average.

Figure 5.31 shows snapshots at t =0 and 3 (left and right), with views along the negative z-axis (top) and along the positive y-axis (bottom). The top right image should be compared with Fig. 1 of Salo (1992b). The system has evidently developed the gravitational wakes or density transients reported by Salo, and predicted by Julian and Toomre (1966) for rotationally supported disks that undergo gravitational perturbation. The snapshot shows that aggregates are generally associated with these unstable waves. Indeed, the equilibrium state for model (vi) in general appears to be the continual formation and dissolution of such structures. Their orientation can be explained qualitatively by

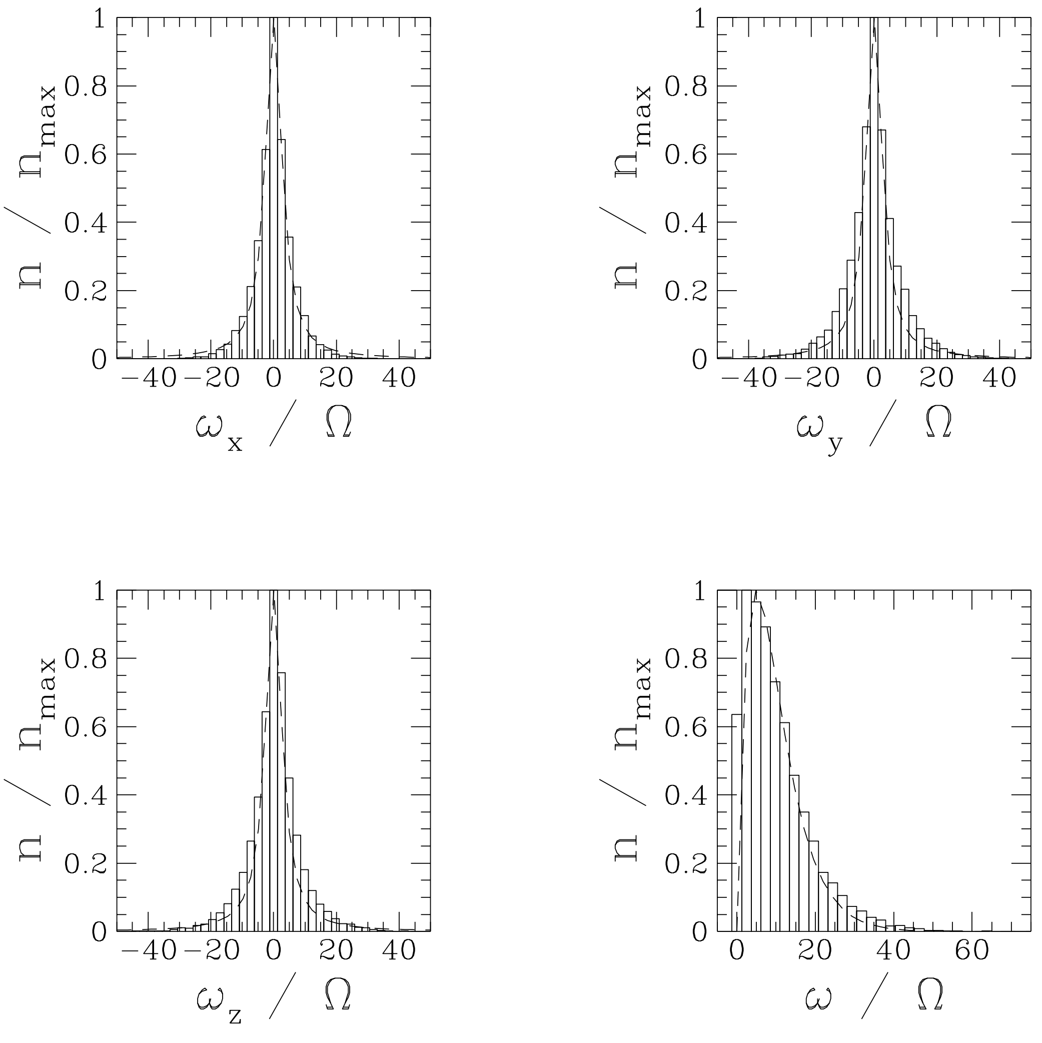


Figure 5.29: Spin distributions for model (vi.6). The dashed curves for ω_x, ω_y, and ω_z are Lorentzians, while the curve for ω (bottom right) is of the form $x\mathrm{e}^{-x}$. Though impossible to discern by eye, the ω_z distribution is centred slightly off zero, with $\overline{\omega}_z \sim 0.2$.

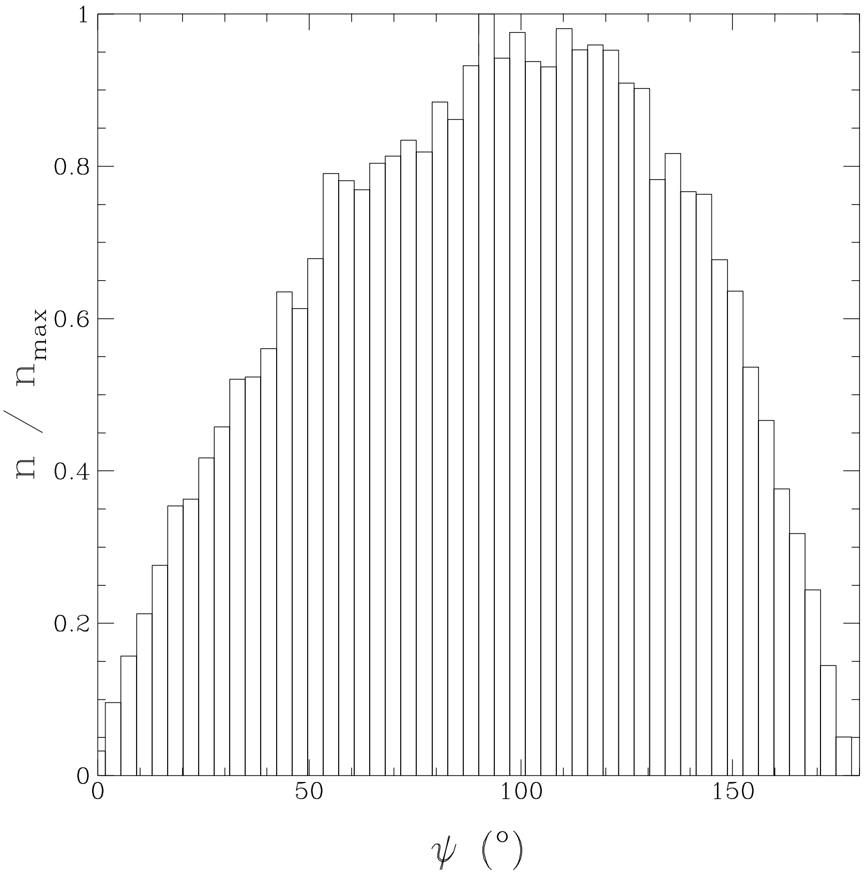


Figure 5.30: Obliquity distribution for model (vi.6). The FWHM is approximately 120°, or 2/3 of the complete range. On average, particles spin slightly retrograde.

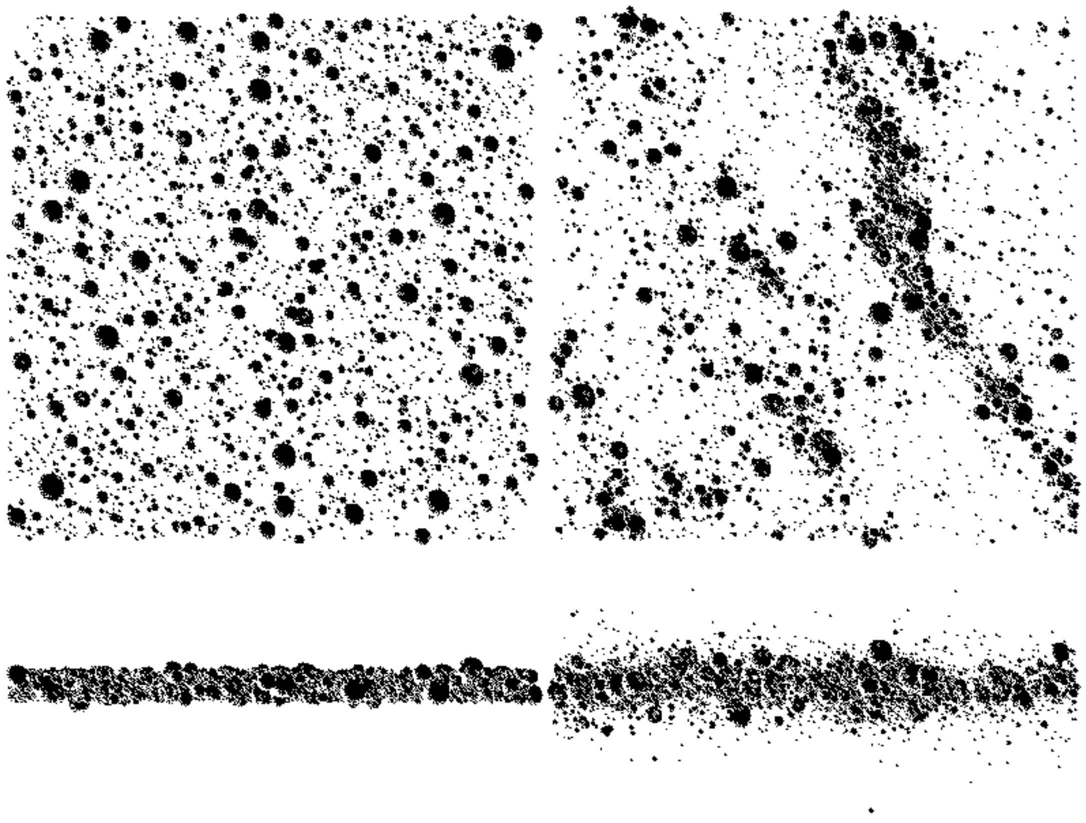


Figure 5.31: Views of model (vi.6) at time $t = 0$ (left) and $t = 3$ (right), looking down on the z-plane (top) and along the y-axis (bottom). Notice the symmetry in the starting conditions, recalling the mass-weighted balancing about the centre of mass. The evolved system shows several transient density features. Here the box is 170 m on a side.

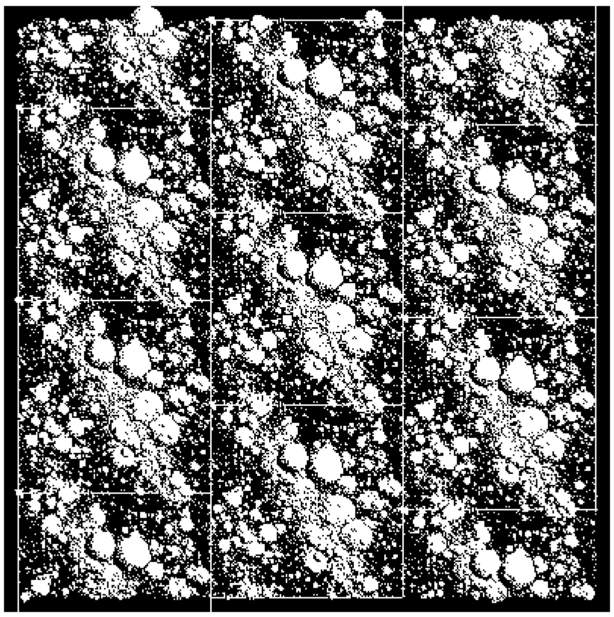


Figure 5.32: View looking down the z-axis at model (vi.7) after one quarter of an orbit. The view includes the ghost boxes, each of size 50 m. The optical depth is unity. A transient density feature is already forming; note how it seems to extend beyond the central box.

the differential rotation of the disk: any condensations that form suffer from shear in the $\pm y$-directions relative to their centres of mass, twisting and pulling the clumps into configurations that minimize the net differential force until other disruptive impacts occur. The observed pitch angle ($\sim 30°$) is consistent with values found by Salo (1992b) for three-dimensional simulations. Also note the evolution in z illustrated by the images at the bottom of Fig. 5.31: the system was started very flat (only a few $R_{\rm max}$) to encourage faster attainment of equilibrium; by $t = 3$ the system has relaxed into a typical equilibrium configuration.

The last entry in Table 5.2, model (vi.7), provides the most realistic simulation so far of conditions at the centre of Saturn's B ring, with a large size range $\Delta R = 0.2$–5 m and optical depth of unity. Unfortunately, because of the high density and large particle number, it is also the slowest to compute, taking several CPU days on a DEC Alpha workstation just to follow one complete orbit. Figure 5.32 is a snapshot of the system at $t = 0.27$, already showing the formation of a transient density feature. The central box and surrounding ghost boxes are shown to emphasize the presence of the density enhancement. Though not yet at equilibrium, the statistical properties for this system given in Table 5.2 are consistent with trends seen in the other model (vi) runs.

It turns out that a size distribution is not required in order to form aggregates as long as the central box is big enough. Figure 5.33 shows four snapshots of a system of 5 000 equal-size ($R = 5$ m) particles in a box ~ 1 km on a side ($\tau = 0.4$). The images are for $t = 0$, 0.75, 1.5, and 2.2. The strong striations are clearly visible here, developing rapidly from the uniform distribution and evolving into thicker bands after a few orbits. In contrast, the model (iv) runs showed no evidence of aggregation and the model (v) runs formed only loose associations. The critical factors governing aggregate formation in the box model for systems with interparticle gravity are evidently the number of particles and the ratio s/R between the box size and the particle radius. In Fig. 5.33, these quantities (N and s/R) are much larger than for any of the previous models. At the same time note that the density was relatively low, so the dynamical optical depth is not as important

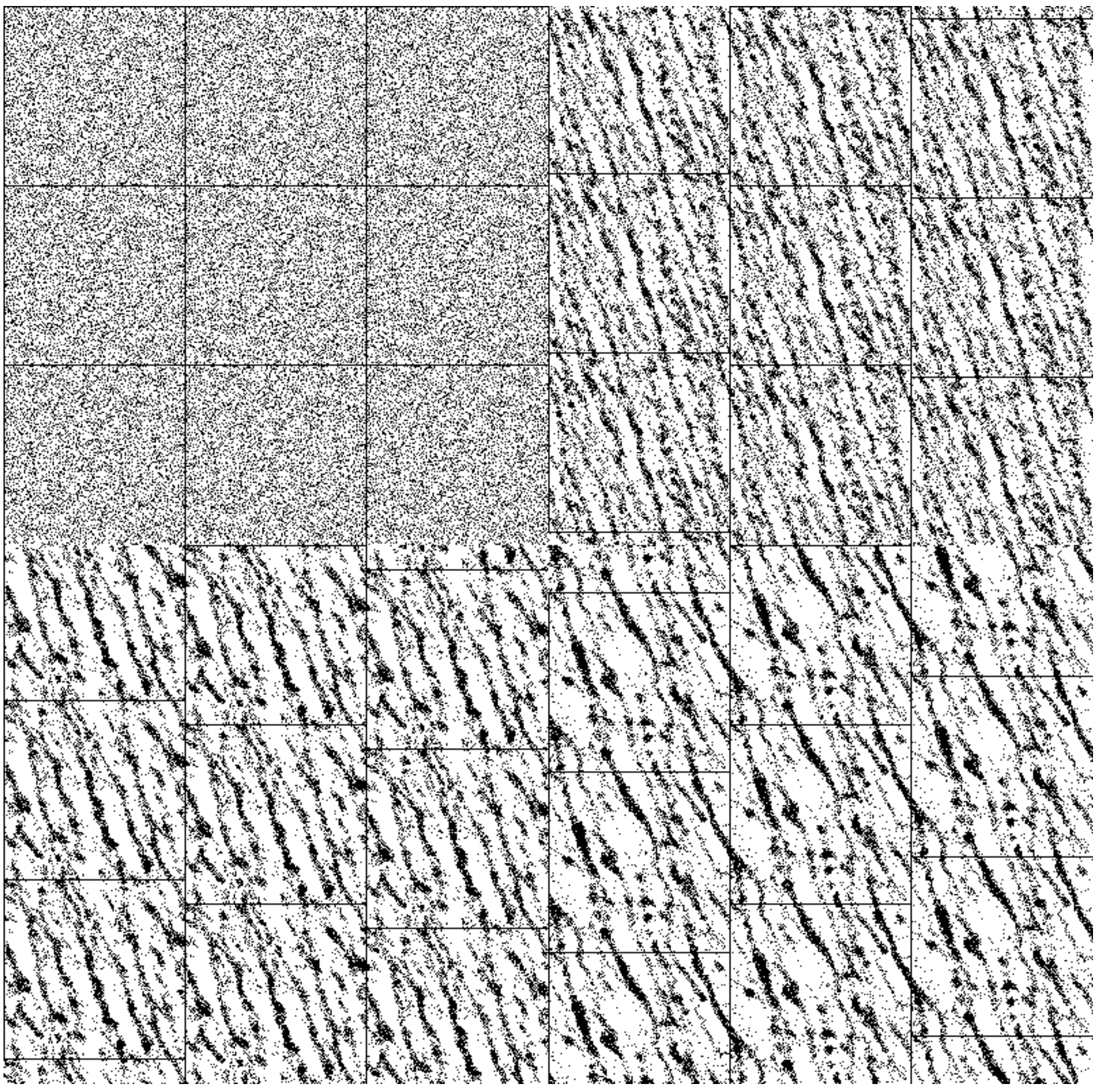


Figure 5.33: Snapshots at $t = 0$ (top left), 0.75 (top right), 1.5 (bottom left), and 2.2 (bottom right) of a system of 5 000 equal-size particles. The box size is 1 km and the particle radii are 5 m. The dynamical optical depth is 0.4.

for aggregate formation.

A disturbing aspect of some of the large-scale features reported here is that they can be comparable in size to the central box, and indeed may even extend beyond the box (see Fig. 5.32 for example, and Fig. 5.33 in particular). For a model that employs periodic boundary conditions, this means that such a structure may actually interact with itself, confusing the interpretation of the results. One possible consequence is that such structures break up and reform more often in these models than they would in reality, or at least are truncated in size. Another related problem is that the mean velocities are fairly large in model (vi), implying large radial excursions (which are in fact seen in animations), possibly invalidating the local nature of the model. These problems can only be addressed, however, with larger box sizes, necessitating much larger values of N to test models with comparable density. Longer integrations would also help address the question of the long term stability of the aggregation process.

Chapter 6

Other Applications & Future Work

Although `box_tree` was designed as a tool for modelling planetesimal dynamics, the code was written in a sufficiently general fashion for it to be applied successfully to other unrelated problems. Nonetheless, there are many avenues open for further development of `box_tree`. This chapter outlines work done to date with `box_tree` on gravitational microlensing and galaxy-galaxy collision simulations, and closes with a discussion of the most important improvements that could be made to the code.

6.1 Code Generality

The `box_tree` header file `params.h` and the run-time parameter file (described fully in the Appendix) allow the user to tailor the behaviour of `box_tree` to a particular problem. The most important parameter is the choice of reference frame, which includes, for example, the rotating frame described in Chapter 2, and the option of an inertial frame. In principle, any problem can be treated in the inertial frame, but many problems are better suited to specialized frames, especially if interparticle gravity is included only as a perturbation on a dominant force. In fact, `box_tree` uses an inertial frame by default, applying external potentials and coordinate transformations to change the reference frame as desired. This simplifies the procedure of adding a new frame of reference, as was done for the galaxy collision simulations described below.

Another important parameter that improves the code generality is the choice of boundary conditions. Currently choices include periodic and unbounded systems. The periodic boundary condition option can be used in either the rotating or inertial frame. In the rotating frame, shear is taken into account during a boundary crossing, while in the inertial frame the particle velocity remains unchanged. Ghost boxes may even be used in the inertial frame, though they remain stationary with respect to the central box. For the unbounded case, the concept of a "box" no longer applies, and particles are free to move anywhere. This is particularly useful when an entire complex system (such as two galaxies) is being modelled.

The tree code can be used in any frame and with any boundary conditions. Recall however that the tree is never rebuilt in its entirety to keep it up to date; rather, the tree is repaired on a node-by-node basis as needed (§3.4.1). This poses a problem in the unbounded case, since `box_tree` would normally complain if a particle tried to move outside the tree. Consequently, the user may specify a tree expansion factor in the parameter file at run time: if a particle exits the tree at any time, the root size is expanded by the supplied factor as many times as necessary to encompass all the particles. Once the new size has been determined, the old tree is destroyed and the new tree is constructed. Fortunately, this relatively expensive procedure is needed only infrequently during a typical unbounded simulation.

There are other useful run-time parameters that extend the code generality. These include: an option to read initial conditions from a file; an option to use softened potentials, thereby disabling collisions; an option to turn off interparticle gravity; an option to disable mergers; and an option to enable or disable the tree. Options that can be changed at compile time include the tree dimension and the maximum number of particles allowed in the simulation. It is felt that this level of flexibility is essential for `box_tree` to be of maximum benefit.

6.2 Gravitational Microlensing

In a recent study of gravitational microlensing (Lewis et al. 1993), a stripped-down 2D version of `box_tree` was used to reduce the time required to evaluate the microlensing equation. Since there was no integration as such to perform, only the components of `box_tree` directly related to the tree code were used. In simplistic terms, the study of gravitational microlensing involves tracing changes in the observed flux from a distant source as the light passes through an intervening mass distribution consisting of discrete gravitational potentials, such as the stars and other compact objects that make up a galaxy. The intervening potentials split the source light into many images (a phenomenon called microlensing); changes in the summed flux of these images caused by relative motions of the source and lenses can be measured.

6.2.1 New Multipole Expansion

The microlensing equation describes the deflection suffered by a ray of light traveling through the mass distribution. The equation involves a sum over all lenses of the form:

$$\sum_i \frac{m_i\,(\boldsymbol{x} - \boldsymbol{x}_i)}{|\boldsymbol{x} - \boldsymbol{x}_i|^2},$$

where the subscript i labels stars in the image plane. This sum is reminiscent of the expression for the net gravitational acceleration due to a distribution of point masses, except that the sum above varies as $1/r$ instead of $1/r^2$. To use the tree code to approximate these contributions to the light curve, a formula for the multipole expansion of a $1/r$ law is required. Appendix A of Lewis et al. (1993) describes the derivation in detail. Implementation of the new expansion was straightforward and provided a considerable reduction in the computation time required to generate a given light curve.

6.2.2 Discontinuity Problem

Part of the technique used in generating light curves involves numerically tracing roots (zero points) on a plane, evaluating the microlensing equation at each point. However, the deflection angle may change discontinuously from one point to the next if a particular node expansion is performed at one point and not at the other. Recall that the expansion criterion depends only on the angle subtended by the node at the point in question. Discontinuities can be removed by using a weighted average of the contribution to the deflection angle from the node and from its immediate children if the opening angle is within a small fixed interval of θ_C. The actual formula used can be found in Appendix A of Lewis et al. (1993). This simple yet important technique for eliminating discontinuities may prove useful in other applications that incorporate tree code.

6.3 Galaxy Collisions

An ongoing project (Thomson & Richardson, in preparation) uses `box_tree` to model grazing collisions between a small galaxy in a near parabolic encounter with a galaxy ten times more massive. The large galaxy is left relatively undisturbed by such an encounter, but recent test particle simulations suggest the disk of the small galaxy is disrupted violently into three main parts (Thomson 1992): an accretion ring about the massive galaxy, a large "shred" that escapes the system, and a bulge remnant. A key question is whether the shred, or a significant portion of it, can hold together under self-gravity. If so, then this model may explain observed features of Centaurus A and some of its immediate neighbours (namely the prominent dust lane seen in Centaurus A, and the configuration of a nearby dwarf elliptical galaxy and an irregular bar-shaped galaxy). To test the properties of the shred thoroughly, however, requires $\sim 10^4$ or more self-gravitating particles, which is why `box_tree` has been adapted for this problem.

Since the large galaxy remains relatively intact following the collision, it is modelled by a single large Plummer potential. Initially the smaller galaxy was modelled by a fixed bulge component (also a Plummer potential) and a disk component of self-gravitating particles with a fixed softening parameter. Later a halo component of particles with larger softening was added to help stabilize the disk. Later still a gas component was introduced to see how it would be affected by the galaxy collision (it is believed that the gas may collect at the tip of the shred, becoming a bright star-forming region — see Schweizer 1978; Mirabel, Dottori, Lutz 1992). The gas is actually modelled by particles of fixed size that are allowed to undergo dissipative collisions with each other (Noguchi 1988). The evolution of these components is best seen in animations, using different colours for each component. Initial conditions are supplied through a data file generated by an external program and are based on Barnes (1992). A 3D tree is used since the initial collision trajectory may be inclined to the disk and the particles are typically scattered in all directions after the encounter, making a 2D tree inefficient.

6.3.1 Accelerated Frame of Reference

Since the tree currently assumes the system is more or less stationary, it is convenient to use a frame centred on the bulge of the smaller galaxy. This allows the bulge component to simply be added in as an external potential for the other particles. The massive galaxy is then treated as a single particle with large softening and, for maximum efficiency, it is *excluded* from the tree structure. The new frame is non-inertial due to the mutual acceleration between the two galaxies; this effect must be accounted for in the equations of motion of the particles. The derivation of the frame acceleration is similar to the derivation of the equations of motion of protoplanets in the heliocentric frame (cf. "Cowell's Method" in Brouwer & Clemence, 1961). If $\boldsymbol{\mathcal{R}}_i$ is the position of the ith particle in the inertial frame (where $i = 0$ denotes the large galaxy), then the position in the new frame is given by $\boldsymbol{r}_i = \boldsymbol{\mathcal{R}}_i - \boldsymbol{\mathcal{R}}_b$, where $\boldsymbol{\mathcal{R}}_b$ is the position vector of the bulge of the small galaxy in the inertial frame. The accelerations of the ith particle and the bulge in the inertial frame are given by:

$$\ddot{\boldsymbol{\mathcal{R}}}_i = -\frac{Gm_b\boldsymbol{r}_i}{r_i^3} - G\sum_{j\neq i}\frac{m_j\left(\boldsymbol{r}_i - \boldsymbol{r}_j\right)}{\left|\boldsymbol{r}_i - \boldsymbol{r}_j\right|^3} = \boldsymbol{\mathcal{F}} - \frac{Gm_b\boldsymbol{r}_i}{r_i^3},$$

and,

$$\ddot{\boldsymbol{\mathcal{R}}}_b = -G\sum_j \frac{m_j\boldsymbol{r}_j}{r_j^3} \simeq -\frac{Gm_0\boldsymbol{r}_0}{r_0^3},$$

where $\boldsymbol{r}_0$ is the position vector of the massive galaxy in the new frame, and where it has been assumed m_0 is large compared to the m_j's of the small galaxy, and that the $\boldsymbol{r}_j$'s of the small galaxy are symmetrically distributed about the bulge and therefore largely cancel. Hence the acceleration of a particle in the new frame is given by:

$$\ddot{\boldsymbol{r}}_i = \boldsymbol{\mathcal{F}} - \frac{Gm_b\boldsymbol{r}_i}{r_i^3} - \frac{Gm_0\boldsymbol{r}_0}{r_0^3}. \tag{6.1}$$

The first term is calculated by the tree code, the remaining terms are added afterwards. Recall that the massive galaxy is treated just like any other particle, so it is included in $\boldsymbol{\mathcal{F}}$ and is subject to equation (6.1).

6.3.2 Softening

The `box_tree` code was originally designed to remove potential singularities by enforcing particle collisions since the evolution of discrete particles was of chief interest. For galaxy simulations, however, it is completely beyond the means of present-day computers to follow each star individually. Instead softened forces of the form:

$$\boldsymbol{\mathcal{F}}_{ij} = -\frac{Gm_j\boldsymbol{r}_{ij}}{\left(r_{ij}^2 + \epsilon^2\right)^{3/2}}$$

are used to smooth over the discreteness. In fact, if softening is being used, `box_tree` takes the radius of each particle as it's softening parameter, and uses the *maximum* value $\epsilon = \max[\epsilon_i, \epsilon_j]$ for the new potential. This is intended for use with only a small number of different classes of object. In the present case, there are three classes: the disk and halo particles, and the massive galaxy. A more consistent treatment of softening is discussed in Dyer & Ip (1993).

To ensure correct behaviour, it is necessary to include the softening when performing multipole expansions. Only the softening of the particle for which the new force is being calculated is used, i.e. ϵ_i^2 is added to r^2 in equation (2.9).

6.3.3 Early Results

Figure 6.1 shows a top and side view of a typical post-encounter configuration. The collision occurred at time $t = 10$; the views are for time $t = 50$. The plane of the orbit is inclined 15° to the disk of the small galaxy. Both halo and disk particles are shown (1 000 of each); there is no gas component in this simulation. The halo is very diffuse after the collision and tends to fill the space between the heavier concentrations of disk particles. A tenuous shred is seen in both views, near the top of the first and at the right of the second. Also clearly seen is the bulge remnant and the larger accretion ring about the big galaxy. Note the clumping seen in the shred, suggesting that portions are being held together by self-gravity (also see Barnes & Hernquist 1992; Elmegreen, Kaufman & Thomasson 1993). The number of particles is too small in this simulation to form a well-defined shred. For reference, the small galaxy was originally $\sim$ 10 kpc in radius; in the views shown, the large galaxy is $\sim$ 160 kpc away from the bulge remnant. Since the orbit was originally parabolic and energy was lost to the shred and halo, the bulge is now bound to the large galaxy. At this post-encounter stage the expansion is basically self-similar, so that the configuration at any future time can be inferred by simple scaling. In this way, and by rotating positions in 3D, it is possible to reproduce a configuration similar to that seen in the sky for the Centaurus A system.

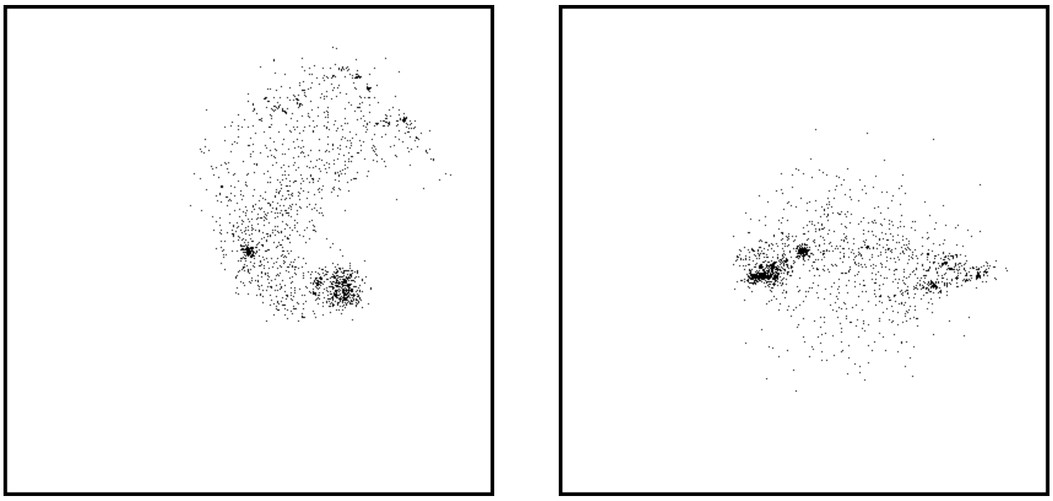


Figure 6.1: Top (left) and side (right) view of a galaxy-galaxy encounter with 15° inclination. In the top view, the large galaxy looped around the left side of the small galaxy, from the top right (20 kpc away) to the current position in the bottom right (160 kpc away). In the side view, the large galaxy is moving to the left and towards the observer. The shred, bulge remnant, and accretion ring primarily consist of disk particles, while the halo is spread out diffusely in between.

Simulations are currently in progress to examine the effects of varying the inclination angle of the orbit and of adding a gaseous component. Investigation of hyperbolic encounters will also be performed. To study the shred more fully, large N simulations will be carried out as well.

6.4 Future Work

There are several major aspects of `box_tree` that could be improved. Some of the most important are discussed in this section.

6.4.1 Fragmentation

Currently in `box_tree` when two particles collide they either bounce or merge. The entire merging process is assumed to occur instantaneously, and all the details are ignored. Such details probably include melting and cooling and a certain amount of fragmentation, especially at high rotation rates. Any fragments are assumed to fall back onto the new mass before they can interact with any other bodies. However, at very high impact energies, the colliding bodies may shatter completely, resulting in many new particles. Such fragmentation may provide a balance to runaway accretion, slowing the formation of the largest bodies but at the same time repopulating the planetesimal swarm.

Beaugé & Aarseth (1990) used a model based on work by Greenberg et al. (1978) that divided the collision outcome into three cases depending on the magnitude of the relative velocity and the impact energy of the colliders: (1) if the relative velocity is less than a critical velocity that depends on the material strength of the colliders, the bodies either bounce or merge; (2) if the relative velocity exceeds the critical velocity, but the impact energy is low enough, cratering occurs, resulting in a transfer of mass between the colliders (the ejecta are assumed to remain bound to the two-body system); (3) if the impact energy is sufficiently high, one or both of the bodies shatter, forming new particles.

A model based on experiment is used to approximate the mass and velocity distribution of the fragments that result from case (3). In Beaugé & Aarseth (1990), each fragmenting body was divided into four smaller bodies plus the remaining core. A conservative number of fragments was chosen to prevent the simulation from becoming unmanageable.

It would be straightforward to accommodate fragmentation in `box_tree`. Each new body is easily placed in the tree, and the code is already set up to handle particles of different mass. A model would be required, however, for the spin distribution, such that the total angular momentum of the system is conserved. In addition, a minimum mass would need to be defined to prevent the formation of infinitesimally small bodies.

6.4.2 Gas Drag

Gas drag has the effect of removing both energy and angular momentum from solid bodies, causing even circular orbits to spiral inwards (Adachi, Hayashi & Nakazawa 1976; Weidenschilling 1977; Beaugé, Aarseth & Ferraz-Mello, in preparation). Though `box_tree` makes provision for a simple treatment of gas drag, it has never been tested in a proper simulation. One difficulty is that by definition gas drag will increase the radial velocity dispersion, which may invalidate the box model assumptions. However, with careful monitoring, gas drag may add an important component to planetesimal simulations. In particular, since the drag is inversely proportional to the particle radius, it would have a systematically greater effect on low mass particles, sweeping them between the larger particles and possibly enhancing the accretion rate.

6.4.3 Hermite Integrator & Block Steps

As described in §3.1, `box_tree` uses a standard individual time-step scheme for integrating the equations of motion. One drawback to this method is the fact that the integrator requires a history of the previous few time-steps to form its correction to particle positions and velocities. In the Hermite scheme (Makino 1991; Makino & Aarseth 1992), the first time derivative of the force is calculated explicitly, eliminating the need for memory of previous time-steps. This is possible because the second and third order derivatives of the force can be interpolated from the force itself and its first derivative using a third-order Hermite polynomial. Explicit calculation of the first derivative of the force increases the CPU expense, but, according to Makino & Aarseth (1992), the improved stability of the technique makes it possible to use times-steps that are nearly twice as long. From these considerations alone, on balance, the Hermite scheme should give slightly better performance.

There is another advantage, however. As pointed out in §3.6, the force polynomials of particles involved in boundary crossings, collisions, and mergers must be reinitialized under the current scheme. Much of this process could be eliminated under the Hermite scheme since the second and third force derivatives are not required in advance. This would necessitate, however, a change in the time-step formula for initialization, since equation (3.7) makes use of the higher order derivatives. In this case a simpler expression, say $\eta\mathcal{F}/\dot{\mathcal{F}}$ or equation (3.6) could be used, possibly with a smaller η, and only for initialization. An added benefit is that external potentials would be much simpler to program, as there would be no need to explicitly code the higher order derivatives (see Appendix).

In order to obtain both the force and its first derivative, the tree code would need to be modified. In particular, the multipole expansion would become considerably more complex. For example, the first derivative of equation (2.9) is given by:

$$\begin{aligned}\dot{\boldsymbol{\mathcal{F}}} &= -M\left[\frac{r^2\dot{\boldsymbol{r}} - 3\,(\boldsymbol{r}\cdot\dot{\boldsymbol{r}})\,\boldsymbol{r}}{r^5}\right] + \frac{r^2\left(\dot{\mathbf{Q}}\cdot\boldsymbol{r} + \mathbf{Q}\cdot\dot{\boldsymbol{r}}\right) - 5\,(\boldsymbol{r}\cdot\dot{\boldsymbol{r}})\,(\mathbf{Q}\cdot\boldsymbol{r})}{r^7} \\ &\quad + \frac{5}{2}\left\{\frac{r^2\left[\left(\dot{\boldsymbol{r}}\cdot\mathbf{Q}\cdot\boldsymbol{r} + \boldsymbol{r}\cdot\dot{\mathbf{Q}}\cdot\boldsymbol{r} + \boldsymbol{r}\cdot\mathbf{Q}\cdot\dot{\boldsymbol{r}}\right)\boldsymbol{r} + (\boldsymbol{r}\cdot\mathbf{Q}\cdot\boldsymbol{r})\,\dot{\boldsymbol{r}}\right] - 7\,(\boldsymbol{r}\cdot\dot{\boldsymbol{r}})\,(\boldsymbol{r}\cdot\mathbf{Q}\cdot\boldsymbol{r})\,\boldsymbol{r}}{r^9}\right\},\end{aligned}$$

where $\dot{\mathbf{Q}}$ is given in equation (3.2). Each term in this equation is straightforward to calculate, but there will certainly be a large penalty for the extra work. Furthermore, $\dot{\mathbf{Q}}$ would need to be calculated for each node as well. It may be that with the Hermite scheme it would only be advantageous to perform a multipole expansion over a node if there is a minimum number of children in the node, say four or more.

The efficiency of a code employing the Hermite scheme can be further improved through the use of block time-steps (Aarseth, private communication). Particles retain individual time-steps with this method, but the steps are allowed to take on only discrete values, typically in powers of 2. This simplifies predictions considerably, and also eliminates round-off error problems that plague timing routines (see Appendix). Although block time-steps could be used in the existing integration scheme, they are found to be of greater advantage in the Hermite scheme. Note that block time-steps are also used in tree codes (MA), but mostly in the context of vectorization (see following section).

6.4.4 Parallelization

Tree code does not lend itself well to parallelization due to its inherently recursive and unbalanced nature. A version of the code that incorporates limited *vectorization* (where simple arithmetic is performed on an array of numbers using dedicated hardware) was successfully implemented by Hernquist (1987) for use on CRAY supercomputers equipped with FORTRAN compilers that support recursive function calls. In this case the tree walk for determining the force on a particle is vectorized by first obtaining a list of all nodes with $s/d < \theta_C$, then performing multipole expansions over these nodes in a vector loop.

True parallelization, however, involves farming out complex but balanced tasks to multiple processors. In principle, a factor N_p improvement in speed is possible, where N_p is the number of (identical) processors available. In practice, a collisional N-body code would be unlikely to ever attain this limit, since only parts of such codes are suitable for parallel execution (namely those that involve loops over all the particles, or all the nodes in the tree); there would still be many sequential steps in between.

The simplest parallelization of `box_tree` would probably involve modifying just the tree walk, which is currently the most expensive routine (see §4.2). Each processor would be given a starting node, and would return the force contribution from that node. Starting nodes would be chosen in sequence from level $(1/n)\log_2 N_p$ in the tree, where n is the tree dimension (for simplicity it is assumed N_p is a power of 2). Unless the tree was uniformly populated, however, the processors would not be balanced; to do better would probably require fundamental changes to the code. Also, since just the tree walk is being parallelized, only a modest improvement (a few factors of 2) would be expected. Nonetheless, a parallel implementation of the tree walk would probably do better than vectorized versions, especially if the tree were stored in a global memory that could be accessed by all the processors, to minimize software data passing.

6.4.5 Miscellaneous

There are other miscellaneous items on the `box_tree` "to-do" list. These include: improvement of the initial conditions code, adding more options and a friendlier interface; streamlining of the time-step formula routines; an option to rebuild the tree at user-specified intervals; an option to display the closest binaries at a given time; and full ANSI C function prototyping. As they are fairly straightforward in nature, many of these improvements are likely to be implemented in the near future.

Chapter 7

Conclusions

A new tree code for fast simulation of planetesimal dynamics in a thin disk has been presented. The `box_tree` code combines elements of existing techniques, namely the box code method of WT, which employs a self-similarity argument to confine the region of interest to a small orbiting patch with periodic boundary conditions and uniform Keplerian shear, and the tree code method given by BH, which considerably reduces the expense of force calculations in an N-body system by recursively expanding hierarchical groups of particles into corresponding gravitational moments. Several new techniques have been introduced to eliminate special problems and to minimize both the computation time and the force errors relative to a direct summation method: (1) tree repair to update parts of the tree as needed, rather than rebuilding the tree every time-step or in average block steps; (2) node updates and prediction to give a more accurate representation of the force; (3) node packing to deal with overlapping particle projections in a 2D tree; (4) stretchable nodes to allow for the rapid shearing motion in the system as well as to give a realistic node size in a 2D tree for nodes containing children that are extended perpendicular to the plane; and (5) position and velocity corrections to compensate for the fact that collisions are detected only after penetration has occurred.

With all these elements in place, a considerable gain in efficiency and accuracy has been achieved. The `box_tree` code is several times faster than the recent N-body box code of ALP for moderate values of N (a few hundred), while maintaining force errors close to the theoretical limit for quadrupole expansions. The code becomes increasingly advantageous for larger values of N, in a manner consistent with an $\mathcal{O}(N \log N)$ algorithm. Although fairly optimized already, there are several ways in which the code efficiency could be improved further. One possibility is the incorporation of a fast square root algorithm, as the square root function is currently the third most expensive routine in `box_tree`. Other possibilities include the use of a Hermite integrator or even the use of parallel processors.

Two major simulations were presented in this thesis, one for early planet-forming planetesimals and the other for planetary rings. In the former, a large N model in 2D was shown to develop runaway accretion, forming several large planetesimals many hundreds of times larger than the starting mass in a short period of time. Equipartition of energy was evident, with the largest masses having the smallest velocity and spin dispersions in general. An explanation for oscillations in the z velocity dispersion seen in an ALP model was also given, where it was argued that a fixed 1 yr sampling interval coupled with a slight perturbation from the mean 1 yr orbital period due to interparticle gravity could give rise to an apparent long-term oscillation. Also, the origin of initial oscillations was explained to arise from the finite-number statistics of a random velocity distribution.

The second major simulation took advantage of the provisions for interparticle gravity, size distributions, and spin in `box_tree` to generate new models of the dynamics in

Saturn's B ring, extending the original work of WT and others. Comparisons with WT models were presented and showed excellent agreement overall. Improved detail allowed a closer look at layering phenomena. Extensions into the self-gravity and small size distribution regimes showed behaviour similar to the earlier models, and there was still no evidence for overall viscous instability. Larger size ranges at moderate optical depth gave rise to aggregate formation and gravitational wakes, also seen in similar simulations by Salo. The most realistic simulation so far (ΔR = 0.2–5 m, $\tau = 1$) was presented, and also formed density transients. Since *all* size range models showed the rapid development of some form of association, it must be concluded that such systems strongly favour aggregate formation on very short time scales (less than one orbital revolution), and this may help explain the non-uniformities seen in Saturn's outer rings. It should be noted that the wake lengths in these local simulations may be limited by the choice of box size. Particle spins in these later models were found to lie inside a rotational energy equipartition envelope at equilibrium and were retrograde on average in the local frame, though the particles generally had a large spread in obliquities.

During the development of `box_tree`, an effort was made to keep the code as general as possible. This allowed the adaptation of the code for use with gravitational microlensing and galaxy collision simulations, as well as other simulations that required different reference frames. Equally important, the general nature of the code allows relatively straightforward implementation of such refinements as particle fragmentation, an important dynamical effect in planetesimal evolution that has not yet been considered with `box_tree`.

There are other possible refinements that have not been discussed. For example, frost layers on centimetre-sized particles have been shown by experiment to increase dramatically the chance of sticking (Hatzes et al. 1991; Bridges, Supulver & Lin 1993), a possibility that may apply to ring particles. In addition, there are improved elasticity models, also based on actual experiment. Recent results suggest collisions are more elastic than previously thought (thereby increasing the mean equilibrium thickness of the disk and possibly reducing the compactness of aggregates), and it has been found that the *transverse* coefficient of restitution may also be velocity-dependent (Supulver, Bridges & Lin 1993). Other enhancements that need to be made in future include the introduction of non-spherical particles (e.g. ellipsoids with a distribution in axis ratio), although keeping track of position angles and evaluating torque effects correctly will be challenging.

The Cassini mission to Saturn offers the exciting possibility of testing recent numerical studies of planetary rings directly. To make the most of this opportunity, it is important that as many improvements as possible be made to existing techniques. Some basic refinements have already been mentioned, but there are also many aspects of existing models that need to be explored further in order to determine the best approach towards future development. Particle aggregates in particular merit much further study: the maximum aggregate size needs to be determined, requiring larger scale simulations; a better picture of how the aggregates form and dissolve needs to be obtained; and a determination of the role aggregates may play in reducing the rate of angular momentum transport is also needed. As computing facilities improve, these problems will become increasingly easier to address.

The source code of `box_tree` and other major related programs is freely available to researchers (see the Preface for details). The code includes a package for generating and viewing movies of the planetesimal and tree dynamics. Movie generation has not only proved invaluable as a diagnostic tool but has also turned out to be a very instructive way of examining the non-intuitive particle motions that occur in the rotating frame. Appendix A is a guide for those who wish to use the `box_tree` package on their own. Full source listings of `box_tree` with introductory comments by file are given in Appendix B to simplify the process of modifying the code. Users that register with the author will be notified of any future updates to `box_tree`.

It is hoped that `box_tree` will prove a useful tool for future research in planetesimal dynamics and other areas of study.

REFERENCES

Aarseth S. J., 1985, in Brackill J. U., Cohen B. I., eds, Multiple Time Scales, Academic Press, New York, p. 377

Aarseth S. J., 1994, in Benz W., Barnes J., Müller E., Norman M., eds, Computational Astrophysics: Gas Dynamics and Particle Methods. Springer-Verlag, New York, in press

Aarseth S. J., Lin D. N. C., Palmer P. L., 1993, ApJ, 403, 351 (ALP)

Adachi I., Hayashi C., Nakazawa K., 1976, Prog. Theor. Phys., 56, 1756

Araki S., 1991, Icarus, 90, 139

Araki S., Tremaine S., 1986, Icarus, 65, 83

Barnes J., Hut P., 1986, Nat, 324, 446 (BH)

Barnes J., Hut P., 1989, ApJS, 70, 389

Barnes J. E., 1992, ApJ, 393, 484

Barnes J. E., Hernquist L., 1992, Nat, 360, 715

Beaugé C., Aarseth S. J., 1990, MNRAS, 245, 30

Binney J., Tremaine S., 1987, Galactic Dynamics. Princeton University Press, Princeton, NJ

Bridges F. G., Hatzes A., Lin D. N. C., 1984, Nat, 309, 333

Bridges F. G., Supulver K. D., Lin D. N. C., 1993, Icarus, submitted

Brouwer D., Clemence G. M., 1961, Methods of Celestial Mechanics, Academic Press, New York

Burns J. A., Showalter M. R., Morfill G. E., 1984, in Greenberg R., Brahic A., eds, Planetary Rings. University of Arizona Press, Tucson, Arizona, p. 200

Cuzzi J. N., Lissauer J. J., Esposito L. W., Holberg J. B., Marouf E. A., Tyler G. L., Boischot A., 1984, in Greenberg R., Brahic A., eds, Planetary Rings. University of Arizona Press, Tucson, Arizona, p. 73

Dyer C. C., Ip P. S. S., 1993, ApJ, 409, 60

Elmegreen B. G., Kaufman M., Thomasson M., 1993, ApJ, 412, 90

Emori H., Ida S., Nakazawa K., 1993, PASJ, 45, 321

Goldreich P., Ward W. R., 1973, ApJ, 183, 1051

Goldreich P., Tremaine S., 1978, Icarus, 34, 227

Goldstein H., 1980, Classical Mechanics, 2nd ed, Addison-Wesley, Reading, MA

Greenberg R., Brahic A., eds, 1984, Planetary Rings. University of Arizona Press, Tucson, Arizona

Greenberg R., Wacker J. F., Hartmann W. K., Chapman C. R., 1978, Icarus, 35, 1

Hatzes A. P., Bridges F., Lin D. N. C., Sachtjen S., 1991, Icarus, 89, 113

Hernquist L., 1987, ApJS, 64, 715

Hernquist L., 1990, J. Comp. Phys., 87, 137

Julian W. H., Toomre A., 1966, ApJ, 146, 810

Lewis G. F., Miralda-Escudé J., Richardson D. C., Wambsganss J., 1993, MNRAS, 261, 647

McMillan S. L. W., Aarseth S. J., 1993, ApJ, 414, 200 (MA)

Makino J., 1990, J. Comp. Phys., 87, 148

Makino J., 1991, ApJ, 369, 200

Makino J., Aarseth S. J., 1992, PASJ, 44, 141

Marion J. B., Heald M. A., 1980, Classical Electromagnetic Radiation, 2nd ed. Academic Press, Orlando, Florida

Mirabel I. F., Dottori H., Lutz D., 1992, A&A, 256, L19

Nakagawa Y., Hayashi C., Nagazawa K., 1983, Icarus, 54, 361

Palmer P. L., Lin D. N. C., Aarseth S. J., 1993, ApJ, 403, 336

Petit J. M., Hénon M., 1987, A&A, 173, 389

Press W. H., Flannery B. P., Teukolsky S. A., Vetterling W. T., 1988, Numerical Recipes in C (The Art of Scientific Computing). Cambridge University Press, Cambridge

Press W. H., Spergel D. N., 1988, ApJ, 325, 715

Richardson D. C., 1993a, MNRAS, 261, 396

Richardson D. C., 1993b, MNRAS, submitted

Safronov V. S., 1969, Evolution of the Protoplanetary Cloud and the Formation of the Earth and Planets. Nauka Press, Moscow

Salo H., 1991, Icarus, 90, 254 (also Erratum, Icarus, 92, 367)

Salo H., 1992a, Icarus, 96, 85

Salo H., 1992b, Nat (Let), 359, 619

Schweizer F., 1978, in Berkhuijsen E. M., Wielebinski R., eds, Proc. IAU Symp. 77, Structure and Properties of Nearby Galaxies. Reidel, Dordrecht, p. 279

Supulver K. D., Bridges F. G., Lin D. N. C., 1993, Icarus, submitted

Szebehely V., Peters C. F., 1967, AJ, 72, 876

Thomson R. C., 1992, MNRAS, 257, 689

Ward W. R., 1984, in Greenberg R., Brahic A., eds, Planetary Rings. University of Arizona Press, Tucson, Arizona, p. 660

Weidenschilling S. J., 1977, MNRAS, 180, 56

Wetherill G. W., 1980, ARA&A, 18, 77

Wetherill G. W., Stewart G. R., 1989, Icarus, 77, 330

Wisdom J., Tremaine S., 1988, AJ, 95, 925 (WT)

Appendix A

User Manual

This appendix is a guide to using `box_tree` in a workstation environment. For simplicity, it is assumed the workstation is running `SunOS 4.1.x`, though the code can be compiled on other systems (§A.2). Some familiarity with Unix and C is assumed. Reference will be made to the main thesis text in order to minimize duplication of material. The source code for `box_tree` and the parameter file parser `rdpar` are given in Appendix B. Other programs and macros designed for use with `box_tree` (particularly for examining the output), are included in the source distribution. Unix-style "man pages" for `box_tree` are included as well. Instructions on how to obtain the source distribution are given in the Preface.

A.1 Overview

A typical `box_tree` simulation can be broken down into several steps. First a "run" directory is created and the `box_tree` executable and a default parameter file are copied into it. Next the parameter file is edited as appropriate for the desired simulation. The code is then run for a specified length of time, or until interrupted by a user request. After termination the output is analyzed and one or more restarts are carried out if desired. Often the output is monitored as the code is running in order to verify that the simulation is evolving correctly or to check whether it has reached some equilibrium state and may be terminated. All these steps will be considered in greater detail in the following sections.

A.2 Compiling

Each major C program distributed with `box_tree` comes with its own "makefile" (a file found in the source directory and usually called `Makefile`). A makefile consists of a list of commands and dependency rules for compiling source code. The Unix command `make` reads the makefile and performs the necessary operations. Only the `box_tree` makefile will be discussed here; the other makefiles are similar but less complex.

A.2.1 Makefile Options

By default, when the user types `make` in the `box_tree` source directory, an unoptimized executable is created. This is the fastest compilation option. Other options require a single argument to be given to the `make` command. The options and their effects are summarized in Table A.1.

Table A.1: Makefile arguments for `box_tree`.

Argument	Effect
opt	uses "`cc -O`" and strips executable
debug	uses "`cc -g`" to include symbol info suitable for `dbx`
profile	uses "`cc -pg -Bstatic`" for profiling
gcc	uses `gcc` with maximum optimization and source checking
lint	runs `lint` to check the source
cflow	generates flowchart from code
clean	removes redundant files, including object files
backup	makes backup of code and important subdirectories

The command "`make opt`" uses the `SunOS` C compiler to generate optimized and symbol-stripped code. However, if the GNU compiler `gcc` is available, it should be used in preference to `cc` (i.e. type "`make gcc`"). It is widely recognized that `gcc` is a better optimizer than `cc`, and the GNU compiler also carries out a number of useful checks on the source code that `cc` omits. All GNU software is freely available, so there should be no difficulty in obtaining a copy of `gcc`. For debugging, however ("`make debug`"), the `cc` compiler is still used, as it is more compatible with the `dbx` and `dbxtool` debugging packages available with `SunOS 4.1.x` and `OpenWindows`. The user is of course free to change this so that the GNU versions are used instead. Debugging packages such as `dbxtool` have proved invaluable during the development of `box_tree`. Note that executables created for debugging will generally run slower than the default or optimized versions.

The "`make profile`" option is used to generate an executable suitable for profiling. This option adds instructions to the executable to automatically generate a file called `gmon.out` at termination that contains the profiling information. To read the file, the `SunOS` utility `gprof` must be used. Note that it is not possible to optimize and profile at the same time, so the profiling information must be considered only a rough guide to the best code performance. A typical profile run is discussed in §4.2.

The remaining makefile options do not create a `box_tree` executable. The command "`make lint`" runs the `SunOS` utility `lint` on the code to check for potential problems that the compiler may have missed. The "`make cflow`" command writes a simple program flowchart to a file called `cflow.out` using the `SunOS cflow` utility. The "`make clean`" option removes all extraneous files from the source directory, including object files. The `box_tree` executable is *not* deleted however, contrary to the usual practice. Finally, the command "`make backup`" uses the `SunOS` utilities `tar` and `compress` to create a compressed archive of the source code and important subdirectories containing utilities, plotting macros, and test data. The resulting file is called `box_tree.tar.Z` and should be copied somewhere else for safety.

A.2.2 Recompiling

The `box_tree` makefile uses the special "`.KEEP_STATE:`" directive to create dependency files in the source directory when compiling. These files contain information that `make` uses on subsequent compilation requests to determine which source files need to be recompiled. If only one source file is modified, for example, only that file is recompiled. The resulting object file is linked with the others to form the updated executable. This saves considerable time when making small changes to the code. In fact, this behaviour occurs automatically even without the "`KEEP_STATE:`" directive. With it, however, `make` can detect changes to header files or even the makefile itself and determine which source files need to be recompiled as a consequence (usually all of them). For this reason, there is no need to delete object files or the executable between invocations of, for example, "`make debug`" and "`make gcc`"; the code will automatically be recompiled as required. Note that "`KEEP_STATE:`" may not be available on other platforms.

One typical reason for recompiling is to put into effect changes made to one or more of the parameters in the `params.h` header file. These parameters include the maximum number of particles allowed and the tree dimension. See §B.1 for more information.

A.2.3 Other Platforms

The `box_tree` code has been compiled successfully in the past on System V (e.g. `Solaris 2.1`) and DEC Alpha platforms. Recent changes to the code may have introduced a few incompatibilities, but due to difficulties arranging access to foreign platforms there has been no opportunity to identify any new problems. However, such porting difficulties are easily dealt with using preprocessor macros. An examination of the source code (§B.1) will reveal that defining the empty macros `SYSV` and `ALPHA` will eliminate most incompatibilities for these systems. Examples of how to set these macros are given in the makefile. Note that some of the special features used in `box_tree`, such as built-in movie generation, are not supported on foreign platforms.

A.3 Running

A typical invocation of `box_tree` on a Unix platform running the `csh` shell might look something like this:

```
nice +19 box_tree >&! output &
```

Here the program has been put in the background at reduced priority and the output (`stdout` and `stderr`) has been redirected to the file `output`. The exclamation mark ("!") instructs the shell to overwrite the output file if it already exists. It is recommended that each `box_tree` run be carried out in a new directory because, in addition to the `stdout` and `stderr` output, `box_tree` can generate a large number of sequentially numbered data files for later analysis (§A.5). Although `box_tree` will automatically make backups of existing files before writing new ones, a run directory can quickly become cluttered.

Only two files are needed to start a run: the parameter file (described in §A.4.2) and the executable itself, both of which should be copied into the run directory. It is preferable to copy the executable (rather than link to it, for example) because depending on the circumstance, the operating system may not load the executable in its entirety into memory while running. Consequently, any changes made to the executable may cause any current runs to crash. Also, with a copy of the executable, it will always be possible to restart the run from the save file (§A.5.6); new executables may have incompatible save files.

A `box_tree` run will terminate when: (1) the simulation clock reaches a user-supplied termination time; (2) the run reaches a user-supplied CPU limit; (3) the empty file `STOP` is created by the user (e.g. "`touch STOP`" in the run directory); (4) an interactive session is interrupted by pressing `<CTRL><C>`; (5) the user executes a `kill` command; (6) the machine crashes; or (7) the code crashes with an error. Simulation and CPU time limits are specified in the parameter file (§A.4.2). However, if the user does not know in advance when to terminate the simulation, large limits may be specified to allow the program to

Table A.2: Command line arguments for `box_tree`.

Argument	Effect
-b	disables auto backup of existing output files
-l logfile	sets "logfile" as name of file for logging
-p parfile	sets "parfile" as name of parameter file
-r	attempts a restart from most recent save file
-s savfile	sets "savfile" as name of save file
-x	disables logging

run indefinitely. To stop the run cleanly in this case and generate an up-to-date save file (cf. §A.5.6), the "`touch STOP`" facility should be used. If restarts are not important, the job can simply be killed, using the `SunOS kill` command for background jobs, or by pressing `<CTRL><C>` for foreground (interactive) jobs. The run can still be restarted, but only from the last save file. Nothing can be done about a machine crash except to restart from the last save file. In the event of a code crash, see §A.6.

A.4 Input

The parameter file, and optionally a file of preset initial conditions, uniquely define a `box_tree` run. It is possible to *restart* a run without a parameter file, but this is not recommended practice; the parameter file should always be kept with the executable and the output. If nothing else, it will help identify the run for later analysis. The parameter file and the format for supplied initial conditions are discussed in this section. There are also a few command line arguments available which are discussed first.

A.4.1 Command Line Arguments

Table A.2 summarizes the command line arguments available with `box_tree`. These options must be read before the parameter file, which is why they are specified on the command line.

The "-b" option suppresses the default behaviour of backing up existing data files before writing new ones. For example, if the statistics file `box_tree.stats` already exists at the beginning of a new run, it is moved to the file `box_tree.stats%` in case the user did not mean to overwrite it. Turning off this feature may be advantageous for short runs that are I/O-intensive. The "-b" option also has the effect of disabling the check for existing save files when starting a new run. Normally `box_tree` assumes that if a save file exists, the user would want to restart from it (option "-r"); if no restart is specified, `box_tree` halts with an error message. To start a new run in a directory containing a save file without using the "-b" option, simply delete the save file first.

The "-l", "-p", and "-s" options alter the default log, parameter, and save file names, respectively. The default values are specified in the source file `params.h`: the default log file name is `box_tree.log`, the parameter file is `box_tree.par`, and the save file is `box_tree.sav`. Note that a small number of the most recent save files may be set aside temporarily if desired (cf. §A.4.2). By default, the current save file is always `box_tree.sav`; this file is backed up to `box_tree.sav0`, `box_tree.sav1`, etc., as appropriate before each subsequent dump. These backups are gradually overwritten as the run progresses.

The "-r" command instructs `box_tree` to attempt a restart from the current save file (assumed to be `box_tree.sav` unless overridden by the "-s" option). Section A.5.6 discusses save files and restarts in greater detail. The "-x" command disables logging (§A.5.2).

The following example illustrates the use of `box_tree` command line arguments:

```
nice +19 box_tree -p test.par -r -s old_run -b >&! test &
```

In this example, a low-priority job is submitted that restarts from the file `old_run.sav` using new parameters specified in `test.par`. File backups are disabled for this run. The log will be kept in `box_tree.log`. Note that the command line arguments need not be specified in any particular order, although any filename arguments must appear beside their flags.

A.4.2 The Parameter File

Perhaps the easiest way to familiarize the user with the `box_tree` parameter file is to examine a typical example in detail. In this section, the default parameter file supplied with `box_tree` is presented line-by-line from start to finish, with detailed explanations after each major block of related parameters. The actual parameter file contains short descriptive comments between these blocks to help the user when editing.

The parameter file consists of lines of the form:

```
keyword value ! comment
```

The keywords are labels for the data value. The comment is ignored; it serves only as an aid to the user. The "#" symbol may also be used to denote comments. Note that "!" and "#" must not appear in string constants, as they will still be construed as comment markers (cf. `rdpar` source code in §B.2). The keyword and value must be separated by whitespace, generally `<TAB>`s or `<space>`s. The lines containing the keywords and associated data may appear in any order, but currently they are read by `box_tree` in almost the same order they are given in the sample parameter file. There should be no need to change this. Generally the "read-once" parameters are at the beginning of the file, while those parameters that may be altered each restart make up the rest of the file. Real values (as opposed to integer values) may be given in floating point or exponential notation. Double quotes ("") around string constants are optional. Some parameters are multi-valued; the proper syntax will be described as needed. Other details about the parameter file format can be found in §B.2.

Many parameters require data that have dimension, such as length, velocity, etc. In general, `box_tree` assumes such supplied data are in scaled units (cf. §5.1). However, mks units can be used in most cases by giving a negative value. For example, a box size of 0.01 could mean 0.01 AU while -100 would mean 100 m). Any exceptions will be noted below.

The `box_tree` parameters will now be described in order, with one or more lines of the sample file being given first, followed by the detailed description.

```
Comment line            "box_tree test: 3D N 100 rot per w/ghosts w/tree"
```

The first parameter is an optional descriptive comment for file headers. The comment line appears in several output files, as well as the general output. It is a good idea to use the comment line facility in case the output from multiple runs gets confused. The comment line may be changed for a restart if desired.

```
Reference frame         1       ! 1=rotating,2=inertial,3=galaxy
```

Currently there are three choices of reference frame. The choice of reference frame may place limits on which other parameters may be used, or the range of values certain parameters may take. Any such restrictions will be noted as appropriate below. The inertial frame (option 2) is described briefly in §6.1. The rotating frame (option 1) is the subject of much of the main thesis text. The galaxy frame (option 3) is for use with the simulations described in §6.3. The choice of frame largely determines which external potentials (if any) will be applied to the particles during integration. Attempts to change the reference frame for a restart are ignored. In fact, all of the following parameters, up to but not including the verbosity level, are read only once, at the start of a new run.

```
Length scale            1.49597892e11   ! In metres
Mass scale              1.989e30        ! In kilograms
Time scale              3.1558149984e7  ! In seconds
```

In order to convert from mks units to the scaled units, `box_tree` needs to know the length, mass, and time scales for the simulation. The example shown here is for 1 AU, 1 $M_\odot$, and 1 sidereal yr. See §5.1 for other scales. In the rotating frame, times are multiplied by 2π *internally* (i.e. the factor of 2π is dropped for outputs). The velocity scale in the rotating frame also includes a factor of 2π, so that for the current example, the velocity scale is 30 km/s (the rotation speed at 1 AU). This factor is *not* removed for output. These factors of 2π are included so that the mean angular velocity Ω can be set identically to 1 and hence be omitted from all calculations. Note that if only scaled units are used in the parameter file, and the particle density is not needed (to convert from radii to masses), there is no need to set the length, mass, and time scales.

```
Random number seed      0       ! Must be non-negative (0 for "random")
Init cond option        1       ! 1=align,2=unif,3=WT,4=packed,5=supplied
```

The initial conditions for the simulation are specified by the random number seed and the initial conditions option. If a positive number is given for the seed, that value will be used. This allows multiple runs to be performed on the same set of initial conditions. A random seed can be requested by setting the value to zero. In this case `box_tree` sets the seed to the process ID number of the job. The value is output in various places and saved in the restart file (as are all the other parameters), in case it is necessary to reproduce the exact initial conditions at a later date.

Currently there are five initial conditions options. Each one requires a certain number of other parameters to be specified (described below), and in many cases more than one option uses the same subset of parameters. Option 1 ("aligned c-o-m") places particles randomly in the central box and adjusts the positions and velocities so that the centre-of-mass position and velocity of the system are both zero. Option 2 ("uniform random") places particles in a grid superimposed on the box, with equal numbers of particles distributed in each subdivision in a uniformly random manner. The centre-of-mass position is not adjusted in this case, but should lie close to the origin if there are a sufficient number of particles and subdivisions. Option 3 ("WT") applies the initial conditions described in §5.3 for planetary ring simulations. Option 4 ("close-packed") places the particles in layers reminiscent of a cubic lattice. This requires that N/n_l be a perfect square, where n_l is the number of particle layers. The centre-of-mass velocity is also set to zero with this option. Option 5 ("supplied IC's") reads a separate data file for the initial conditions (§A.4.3). Only option 5 can currently be used with the inertial or galaxy frames.

```
Bdry cond option        1       ! 1=periodic,2=unbounded,3=disabled
```

There are three choices of boundary conditions. The first two—periodic and unbounded boundary conditions—are described in §6.1. The third option stipulates that particles not be allowed to leave the box; an error is generated if a particle attempts to do so. Note that only periodic boundary conditions can be used in the rotating frame.

```
Use ghost particles?    1       ! 0=NO,1=YES (NO if unbounded BC's)
Box size                0.04    ! Box size in length units
Initial clock time      0.0     ! Starting time in time units
Initial x vel disp      3.0e-4  ! Initial x vel disp in velocity units
Initial y vel disp      1.5e-4  ! Initial y velocity dispersion
Initial z vel disp      1.5e-4  ! Initial z velocity dispersion
Use small dispersions?  0       ! Toggle (may override x/y/z vel disp)
```

These parameters may be used with any initial conditions option. As with all toggles, the "Use ghost particles?" has two possible values: 0 (NO) and 1 (YES). Ghosts may not be used with unbounded simulations, but are otherwise allowed, even in the inertial frame (the ghost boxes remain stationary in this case). The box size must be specified if the simulation is bounded and the initial conditions are not WT-type (where the optical depth may be used to determine the box size). Note that the box size is expected to be supplied in length units, but mks units may be specified by prepending a minus sign. The initial clock time may be set if desired, but currently the only reason for setting it to a non-zero value is to obtain an "evolved" ghost box configuration (i.e. sheared from the initial 3×3 grid), usually for drawing movie frames.

Initial velocity dispersions may be specified in x, y, and z if desired. In the case of supplied initial conditions, the initial velocity dispersions are added as a perturbation to the supplied velocities. Currently WT simulations ignore the initial dispersions. When using initial dispersions, the x- and y-velocity (or the perturbation in the case of supplied velocities) of each particle is set to the corresponding initial dispersion times a random Gaussian deviate with zero mean and unit variance. For close-packed and supplied initial conditions, the z-velocity perturbation is determined in the same way. For the other initial condition options however (which are only used in the rotating frame), the z-position and velocity are constrained by the requirement of simple harmonic motion [cf. equation (2.7)]. If a vertical scale height is not specified (see below), the initial position and velocity in z of each particle is set according to:

$$\begin{aligned} z_i &= \sqrt{2}\,\sigma_z^0\,\gamma_i\cos(\phi_i), \\ \dot{z}_i &= -\sqrt{2}\,\sigma_z^0\,\gamma_i\sin(\phi_i), \end{aligned}$$

where σ_z^0 is the initial z velocity dispersion, γ_i is a Gaussian deviate, and ϕ_i is a phase angle chosen randomly from a uniform distribution between 0 and 2π. It can be shown from the formalism presented in §5.2.4 that for sufficiently large N, the velocity dispersion of the $\dot{z}_i$'s given above equals σ_z^0. In the case of a mass distribution, the z-components of position and velocity, as well as the x and y velocity components, are weighted by $(\overline{m}/m_i)$, where $\overline{m}$ is the mean mass (cf. §3.3.1). This ensures that the largest particles are not given excessively large initial velocities. Note that this factor was chosen so that it does not change the initial dispersions.

The "Use small dispersions?" flag currently only works with close-packed initial conditions. If set, the initial velocity dispersions are changed to ΩR, where R is the particle radius, overriding any other initial velocity dispersions.

```
Number of particles      100      ! Between 1 and MAX_NUM_PARTICLES
Smallest initial mass    8.0e-11  ! Smallest initial mass in mass units
Largest initial mass     8.0e-11  ! Largest initial mass in mass units
Particle density         1.4      ! Particle density in g/cm^3
Smallest initial radius  0.0      ! Overrides mass if not 0 (length units)
Largest initial radius   0.0      ! Overrides mass if not 0 (length units)
```

These parameters apply to the first four initial conditions options (i.e. everything except supplied initial conditions). First, the number of particles N must be specified, between 1 and the maximum number set in the `params.h` header file (currently 10 000). A range of masses may be set by specifying the upper and lower limits to the distribution. If the limits are equal, the particles will all have the same size. For the close-packed case, the limits must be equal. A particle density in cgs units (negative for mks units) must be specified for simulations that allow particle collisions. A range of radii can be specified, and will override the mass range if given. In this case the particle density *must* be specified in order to determine the particle masses.

```
Mass function exponent  0.0      ! Mass function exponent (0 ==> constant)
Seed mass               0.0      ! Optional init mass of particle 0
```

If a mass or radius range has been specified for the aligned, uniform, or WT initial conditions, the mass function exponent [α in equation (3.1)] must be given. A zero value implies a linear function between the mass or radius limits (i.e. equal numbers of small- and high-mass particles). Currently the initial mass function is sampled smoothly (cf. §3.3.1). A seed mass may also be specified if desired. This mass will be given to particle 0, which will be excluded from the initial mass or radius function. The seed mass is placed at the centre of the box with zero initial velocity.

```
Use softening?          0        ! 0=NO,1=YES
```

For the aligned, uniform, or supplied initial conditions, softening may be turned on. Section 6.3.2 describes the form of softening currently used in `box_tree`. Note that with softening turned on, particle collisions are disabled. The particle radii become the softening lengths.

```
Reject init. binaries?  1        ! 0=NO,1=YES (checks within 10 R_roche)
Vertical scale height   0.0      ! Scale height in R_roche (0.0 to ignore)
```

For the aligned and uniform initial conditions, potential initial binaries may be rejected if desired. Currently only particle pairs separated by less than 10 Roche radii (cf. §5.2) are considered for rejection. In the case of a mass distribution, the Roche radius of the largest mass is used for the limit. Overlapping particles are rejected automatically with this option (this is also done for WT initial conditions). Otherwise a pair is considered a binary if the semi-major axis [cf. equation (3.20)] is positive but smaller than the largest Roche radius. Note that this rejection algorithm is expensive to compute [$\mathcal{O}(N^2)$], since all pairs must be considered. Fortunately, it is only used to set the initial conditions at the beginning of the run.

A vertical scale height in maximum Roche radii may be specified if desired. This overrides the use of the initial z velocity dispersion for setting initial positions and velocities in z (see above).

```
Number of x divisions   0        ! No. unif. random div. in x direction
Number of y divisions   0        ! No. unif. random div. in y direction
```

For the uniform initial conditions, the number of divisions in x and y for the particle placement grid must be specified. If both values are set to 1, the entire box is used at once for particle placement. Note that the product of the number of divisions in x and y should not exceed the particle number N.

```
Optical depth           0.0      ! Overrides box size if not 0
Disk thickness          10.0     ! Initial disk thickness in radii
```

These parameters are used with the WT initial conditions. The dynamical optical depth τ [cf. equation (5.2)] may be specified, overriding the box size (since the particle radii are known). The initial disk thickness must be specified, and is given in (maximum) particle radii.

```
Number of layers        0        ! No. particle layers in z for packing
Expand radii?           0        ! 0=NO,1=YES (overrides density & radii)
Stagger in z?           0        ! 0=NO,1=YES
```

For close-packing, the number of particle layers (at least one) must be specified. The layers are distributed evenly above and below the $z = 0$ plane. If "Expand radii?" is set, the particle radii are expanded so that the particles are just touching, both within a layer and between layers. If "Stagger in z?" is true, each layer is alternately offset horizontally by a particle radius so that the particle centres of every second layer overlap the gaps between particles in the neighbouring layers. If both the latter flags are set, the layers are compressed in the z-direction so that the particles interlock (maximum packing).

```
Init cond filename      "dat000.out"
No. header lines        0        ! No. lines before first line of data
Add shear?              1        ! 0=NO,1=YES
```

The supplied initial conditions option has three parameters. First the filename of the data file must be specified (see §A.4.3 for the file format). Next the number of header lines (if any) to skip in the file must be given. Finally, there is an option to add shear to the supplied velocities. The shear parameter is only recognized in the rotating frame.

```
Verbosity level         1        ! 0=NONE,1=VERBOSE,2=VERY_VERBOSE
Debug level             1        ! 0=NONE,1=ERROR_CHECK,2=MONITOR,3=TRACK
```

These options and most of the remaining parameters may be changed for restarts. The verbosity level controls how much output is sent to `stdout`. The nature of this output is described in greater detail in §A.5.1. The debug level controls how much internal checking is performed during a run. When set to 0, only mandatory checks are performed. At level 1, various simple diagnostics are enabled, and some warnings will be printed if necessary. Level 2 performs detailed and often costly checks, particularly regarding the tree code accuracy. Level 3 is used primarily to track specific particles (see below) one step at a time, displaying information such as the current position and velocity, and the nearest neighbour. Note that the higher levels include all the lower levels, so a verbosity level of 2 and debug level of 3 will generate maximum output and perform all built-in tests. Often both the debug and verbosity levels are consulted to decide whether certain warnings should be printed. A brief search of the source code is perhaps the best way to see how these parameters are used (cf. §B.1).

```
Stop check               1000    ! Check for STOP every n steps (or 0)
CPU check                0       ! Check for time limit expiration
Safety dump              100000  ! Safety dump every n steps
Log time stamp           10000   ! Output time stamp every n steps
```

These parameters control the frequency at which `box_tree` performs various book-keeping functions. The parameter values are in time-steps, with 0 indicating that the corresponding task should not be performed. Note that the time-steps in `box_tree` are irregular, so the frequency at which these tasks are performed bears no relation to the simulation time. Rather, the parameter values are more closely tied with the elapsed CPU time of the process.

The first parameter controls the rate at which checks for the `STOP` file are performed (cf. §A.3). Each check involves an I/O operation so it is prudent not to perform these too often. However, fast response is desirable if it becomes necessary to stop a job quickly and cleanly. An interval of 1 000 steps generally gives a detection time of about one minute.

Checks for CPU expiry are performed only periodically as they involve system calls. The "CPU check" parameter controls the frequency. The "Safety dump" parameter controls the rate at which restart files are written to disk (cf. §A.5.6). A value of 100 000 typically results in one dump per hour. The last parameter controls the frequency of time stamps in the log file (cf. §A.5.2).

```
Output interval          0.1     ! Main output int. in time units (or 0)
Stats interval           0.1     ! Stats summary output interval (or 0)
Dat interval             0.1     ! Particle data output interval (or 0)
Evol par interval        0.1     ! Interval to recalc evol. params (or 0)
Movie interval           0.01    ! Movie frame output interval (or 0)
Debug/check interval     0.0     ! Debug/check interval (or 0)
Termination time         1.0     ! Final termination time in time units
Run time                 0.0     ! Max duration for this run (in CPU mins)
```

These parameters control the major simulation data output and diagnostic intervals as well as the duration of the simulation. The intervals are given in simulation time units, as is the termination time. The output and diagnostic intervals are checked in the order they are given here. For example, at $t = 0.2$ the stats file will be updated before the next particle data file is created. The clocks are checked each time-step to ensure that all output occurs at the correct time and in the correct order. Particle positions and velocities are predicted to high (third) order for output. Details regarding the various forms of output may be found in §A.5. Particular output can be disabled by setting the corresponding interval to 0. Note that output is generated at $t = 0$ as well.

The termination time is the simulation time at which the run is to terminate. A final output check is performed at termination. A restart file is always created at termination, regardless of whether periodic safety dumps are enabled. If the final parameter—the run time in CPU minutes—is non-zero and CPU checks are enabled, the run will be terminated if the elapsed CPU exceeds the given run time. A save file will also be generated in this case.

```
No. backup save files    3       ! Specify between 0 and 9
TSF option               2       ! 1=RV only,2=RV and F,3=F only
Time-step coefficient    0.02    ! Coefficient in time-step formula (TSF)
Minimum time-step        0.0     ! Min time-step in time units (0 ignores)
Maximum time-step        0.005   ! Max time-step in time units (0=no lim)
```

```
Include self-gravity?   1         ! 0=NO,1=YES
Z grav enhance factor   1.0       ! Rotating frame only, must be >= 1
CP sum of radii factor  0.99      ! For use in time-step formula
Radial restitut'n coef. 0.5       ! Between 0 and 1 (or -9.9 for vel-dep)
Trans. restitut'n coef. 0.5       ! Between -1 and 1 (or -9.9 for vel-dep)
Inhibit sliding phase?  1         ! 0=NO,1=YES
Apply coll'n velo adj?  0         ! 0=NO,1=YES
Include gas drag?       0         ! 0=NO,1=YES
Drag coef in x          0.0
Drag coef in y          0.0
Drag coef in z          0.0
Constant drag in y      0.0       ! (scaled by 10^-3)
Allow mergers?          1         ! 0=NO,1=YES
Use tree?               1         ! 0=NO,1=YES
```

These parameters control various aspects of the simulation. Most of these quantities are self-explanatory, so they will only be described briefly. The first parameter controls the number of backup save files `box_tree` is to maintain, up to a maximum of nine (so that only a single digit is required to differentiate between the files). After this number of backup save files has been generated, the oldest is overwritten at the next dump, and so on. The "TSF option" controls which time-step formula to use. Option 1 (RV only) uses equation (3.6), option 3 (F only) uses equation (3.7), and option 2 (RV and F) uses the first equation if a particle's closest neighbour is approaching, and the second if it is receding. Note that only option 3 can be used with simulations that include softening. The time-step coefficient is the value η in these equations. The minimum and maximum time-steps to use (if any) may also be specified.

The "Include self-gravity?" flag controls whether or not interparticle gravity is included in the simulation. This can be switched on and off for subsequent restarts if desired. Note that for simulations in the inertial frame where there is no external potential, the first time-step formula option (RV only) must be used if there is no interparticle gravity (since this implies there are no forces at all acting on the particles). The "Z grav enhance factor" is the quantity g defined for model (iii) of the WT simulations (cf. §5.3); this may only be used in the rotating frame. The "CP sum of radii factor" is the value ε in equation (3.6). This quantity must be less than one for the "RV only" TSF option; it is ignored for the "F only" option.

The radial and transverse coefficients of restitution are the values ϵ_n and ϵ_t, respectively, in equation (3.13). If either value is given as -9.9, the Bridges et al. (1984) velocity-dependent formula is used instead (cf. §5.3). Note that this formula is empirical in nature and applies only to small ice spheres. If "Inhibit sliding phase?" is true, particles that collide with a very small relative radial velocity are forced to bounce elastically (cf. §5.3). If "Apply coll'n velo adj?" is true, the velocity corrections described in §3.5.4 are performed before a collision.

The gas drag parameters will not be discussed here, as they have not been used in any proper `box_tree` simulations to date. For further details, refer to the source listings (§B.1).

The final two parameters in this set control whether particle mergers are allowed, and whether or not to use the tree code to speed up interparticle force calculations. Note that currently the tree cannot be switched on for a restart; the tree code must be used from the beginning.

```
Stats filename          "box_tree.stats"        ! For statistics summary
```

```
Dat file basename       "box_tree"                ! For particle data
Starting dat file no.   -1                        ! Use -1 for default
NLV output filename     ""                        ! For non-local viscosity
```

These are miscellaneous output parameters. First the name of the statistics file (cf. §A.5.3), then the base name for particle data files and the starting file number (cf. §A.5.4). A file number of -1 may be specified for the "default" value, namely 0 at the start of the run or one more than the number of the last data file generated before a restart. Specifying any other non-negative number will instruct `box_tree` to begin sequential numbering at that value, regardless of what has gone before. Currently 999 is the largest file number allowed (cf. `params.h`). For future reference, a numbered file has the following components: the base name (e.g. `box_tree`), the file number (e.g. `999`), and the extension (e.g. `.dat` in the case of particle data files). Thus by default the first particle data file generated is `box_tree000.dat`.

The filename for output of non-local viscosity data [cf. equation (5.4)] is the final parameter in this set. Currently the data is generated every collision, so the file can grow quickly and much time can be spent on I/O operations. If no filename is specified, NLV data is not calculated. This option is included only for comparison with the WT simulations and may not be supported in future versions of the code. See the source listings (§B.1) for the file format.

```
Tree size               0       ! In length units (0=box size)
Expansion factor        1       ! For resizing, use > 1 (1=no expansion)
Maximum opening angle   0.6     ! Max. subtended angle for mult. exp.
Use quadrupole?         1       ! 0=NO,1=YES
Use minimum repair?     1       ! 0=use RemoveFromTree(),1=MoveInTree()
Use hi-ord. prediction? 0       ! 0=NO,1=YES
Predict monopole?       1       ! 0=NO,1=YES
Predict quadrupole?     1       ! 0=NO,1=YES
Check update times?     1       ! 0=NO,1=YES
Mono time-step coef     0.001   ! Coeff. for node monopole time-steps
Quad time-step coef     0.01    ! Coeff. for node quadrupole time-steps
Exclude particle        NULL    ! For initial conditions only
```

These parameters control the behaviour of the tree code. First the tree size must be specified. If the size is 0, the tree size will be set to the box size. Recall however that an unbounded simulation is not constrained to a box so in this case the tree size must be set explicitly. The expansion factor controls how quickly the tree will resize if a particle moves outside the root node in an unbounded simulation (cf. §6.1). A value of 1 disables expansions.

The opening angle θ_C is the next parameter to be specified. The value is given in radians and may be zero. A warning will be generated if θ_C exceeds $n^{-1/2}$, where n is the tree dimension (cf. §3.4.5). The next six parameters are all toggles that control the multipole expansions and node predictions. If "Use quadrupole?" is set, both the monopole and quadrupole will be used for expansions; otherwise only the monopole will be used. Note that currently the quadrupole cannot be turned on for a restart. If "Use minimum repair?" is set, tree repair following particle updates will be as described in §3.4.1; otherwise each updated particle will be completely removed then replaced in the tree. The former option is much faster, but the latter gives better accuracy since all the ancestors of the particle in the tree are updated, rather than the minimum number required for efficient tree repair. If "Use hi-ord. prediction" is true, leaf particles are predicted to high order when updating nodes. This is expensive and offers only a modest improvement in accuracy. The "Predict monopole?" and "Predict quadrupole?" flags are self-explanatory. Note however that monopole prediction must be enabled for quadrupole prediction to take place. If "Check update times?" is true, nodes are given time-steps according to equations (3.3) and (3.4) using the time-step coefficients specified by "Mono time-step coef" ($\epsilon_{\rm M}$) and "Quad time-step coef" ($\epsilon_{\rm Q}$). If the monopole or quadrupole of a node is found to be out of date, it is updated immediately before being used.

The last item in this set is the first example of a multi-valued parameter. Here a list of particles to be excluded from the tree at the start of the simulation may be given (the list is currently ignored for restarts). Each particle to be excluded must appear on a separate line, preceded by the "Exclude particle" keyword. The list must be terminated by the value "NULL" as shown. The massive galaxy (particle 0) described in §6.3 was excluded from the tree in this way.

```
File basenames          "movie" ! Basename for movie files
Starting frame number   -1      ! Use -1 for default
Frame size              400     ! In pixels (square frame)
View size               0       ! In length units
View centre             0 0     ! Coords of view centre (length units)
Draw tree?              1       ! 0=NO,1=YES
Particle shape          5       ! 0=DOT,1=CIRC,2=SQ,3=DIAM,4=DISK,5=SPH
Radius magnification    500.0   ! Particle size magnification
Viewing distance        0.1     ! Viewing distance in length units
Z magnification         25.0    ! Exaggeration of z dispersion
Hide blocked objects?   1       ! 0=NO,1=YES
Draw velocity vectors?  1       ! 0=NO,1=YES
Default color           255     ! 255=WHITE (other colors given below)
```

These parameters control `box_tree` movie generation (cf. §A.5.5). The parameters begin with the base name for the movie frames and the starting frame number. Next the size of each frame in pixels must be specified. Currently the frames are square. For reference, a typical Sun workstation has a screen resolution of $\sim 1\,150 \times 900$ pixels. The view size is given in the same units as the box and tree sizes. If the view size is zero, the initial tree size is used if applicable, otherwise the box size is used (assuming a bounded simulation). If the view size is negative, the absolute value is taken as a multiple of the box size. For example, a view size of -3 would include the surrounding ghost boxes if applicable. The centre of view in x and y may also be specified (note "View centre" is a different kind of multi-valued parameter: one keyword with two values).

If the tree is enabled, it will be drawn if "Draw tree?" is set. Both 2D and 3D trees can be accommodated, although currently the 3D version is rotated to match the 2D view (such that the xy-plane is seen face-on — see source listing for `draw.c` in §B.1). Section A.5.5 describes the various features drawn.

Currently there are six choices of particle shape for drawing: single dot, open circle, solid square, solid diamond, solid disk, and shaded sphere. The sizes of the shapes are scaled to the particle size, with an optional radius magnification. The single dot should be used when softening is enabled, as the softening lengths tend to be large. A viewing distance and magnification in z can also be specified to exaggerate the sizes of "nearby" foreground objects (large z). If "Hide blocked objects?" is true, the particles are sorted in z before drawing; particles with small z are drawn first so that those with larger z (closer to the viewer) are drawn overtop. Otherwise the particles are drawn in their index order. Velocity vectors will be drawn if "Draw velocity vectors?" is true. The vector

Table A.3: Drawing colours supported by `box_tree`.

Index	Color	Index	Color
0	black	6	purple
1	red	7	cyan
2	pink	8	blue
3	yellow	9–254	grey scale
4	green	255	white
5	orange		

lengths correspond to the distance that would be traveled by each particle in 0.001 time units if all gravity was switched off. The vectors are correctly projected onto the viewing surface. The last movie parameter is the default drawing colour for particles. The colours currently defined in `box_tree` are given in Table A.3.

```
Track particle          NULL    NULL
```

This multi-valued parameter allows the user to tag particles for tracking (see "Debug level" above) or for drawing in different colors. This is a null-terminated list with two values to be specified on each line, the first being the particle index and the second being the desired color. If the color is the default color (see above), `box_tree` just marks the particle for tracking.

```
Check tree?             0       ! 0=NO,1=YES
Check multipoles?       0       ! 0=NO,1=YES
Check force?            0       ! 0=NO,1=YES
```

The last parameters in the file are toggles for various checking routines, currently all associated with the tree. These are only used if the debug/check interval (see above) is non-zero. The first option, if set, enables a self-consistency check for the tree. The second option instructs `box_tree` to check the multipoles after initial tree construction, and to check the accuracy of multipole predictions at subsequent check intervals. The final option enables checking of the tree force on a periodic basis. Error statistics are accumulated with this option and displayed at the end of the run. For more details, see §A.5.1 and §B.1.

A.4.3 Supplied Initial Conditions

Since it is impractical for `box_tree` to generate all conceivable initial conditions, the user has the option of supplying a text (ASCII) file containing all the data necessary to describe the initial state of each particle in the simulation. This feature can also be used to "restart" a simulation if there is no save file. Note that precision will be lost in such a restart so the actual evolution may differ from its original course.

The format of the data file is the same as the format of the long output from `dat_read` (§A.5.4). Each line of the file consists of the data for one particle:

$$i\ i_0\ m\ R\ x\ y\ z\ \dot{x}\ \dot{y}\ \dot{z}\ \dot{y}_{\rm r}\ \omega_x\ \omega_y\ \omega_z\ c,$$

where: i is the index of the particle; i_0 is the *original* index of the particle before any merger events; m is the mass; R is the radius; x, y, and z are the position components; $\dot{x}$, $\dot{y}$, and $\dot{z}$ are the velocity components, including any shear in the y-direction; $\dot{y}_{\rm r}$ is the y-velocity with respect to any mean shear (cf. §5.3); ω_x, ω_y, and ω_z are the spin components; and c is the particle colour (a value of 0 is taken to indicate the default colour). Any number of particles may be read in, up to the maximum limit set in the `params.h` header file. All units are in the scaled units, and positions, velocities, and spins are with respect to the chosen coordinate system. The quantity $\dot{y}_{\mathrm{r}}$ is actually redundant, since an option exists to add shear to the initial supplied velocities. The column is included to be compatible with the output format of `dat_read`. Note that i_0 should be the same as i at the start. If merging is not allowed, these indices will not change. Also note that both the mass and radius of each particle are specified, allowing particles with different densities to be simulated. Finally note that the positions are expected to lie within the central box as well as the root node of the tree if applicable. Particles found to lie outside the box will have their positions adjusted automatically through application of the boundary conditions. However, if an initial particle position is outside the tree, an error is generated. Currently it is up to the user to ensure that the initial tree size accommodates all the supplied positions.

A.5 Output

For a numerical simulation to be useful, it must generate relevant output organized in such a way that the user can examine it quickly and deduce any important results. There are many forms of output generated by `box_tree`, each with a different level of detail. The "standard" output is what would normally be directed to the screen, or to a file using shell redirection commands. This output can be very verbose at times, and often the user may choose to discard it (by redirecting it to `/dev/null` for example). A log file is generated by default; it includes important parameter information and time stamps written out by `box_tree` at fixed intervals. A summary of various statistical quantities (given in full in the standard output) is optionally accumulated in the "stats" file. Data files containing all the information describing each particle in the simulation can be output at fixed intervals. Movie frames can be generated as a graphical summary of the particle and tree data. Finally, save files containing all the information needed to perform an exact restart can be output periodically. The details regarding all these forms of `box_tree` output, and how they may best be analyzed if applicable, will be presented in the following sections.

A.5.1 Standard Output

Depending on the verbosity level and the various output options, the standard output includes a complete account of all the parameters passed to `box_tree`, the main periodic output, the status of "evolving" parameters, collision data, results from various error checks, and any warnings and error messages. Most of the standard output is self-explanatory, so only important aspects of the main periodic output and the evolving parameters will be discussed here. Information regarding warnings and errors can be found in §A.6.

The frequency of the main periodic output is controlled by the parameter "Output interval" described in §A.4.2. Figure A.1 shows a sample output for reference that was generated using the parameter file discussed in §A.4.2. The output contains: the current clock time, elapsed CPU, and time-step count; the centre-of-mass position and velocity of the system and their initial values; the centre-of-mass velocity with respect to any mean shear, divided by Ωs (cf. §4.4.1) — this is labeled as "P.v. err"; the velocity dispersion with respect to any shear; data concerning the particle with the maximum mass; the mean mass and corresponding radius and Roche radius (using an average density in the

```
Centre box stats at t = 1 (total CPU = 6.13e+00 min) after 43777 steps:
   Com pos:  x = -1.42907e-06 y = -1.59531e-03 z = -6.44560e-09 (mag 1.595e-03)
   >initial: x = -3.93799e-18 y = -1.60000e-03 z = +3.10614e-20 (mag 1.600e-03)
   Com vel:  x = +3.64943e-07 y = +2.67208e-06 z = +2.97831e-08 (mag 2.697e-06)
   >initial: x = +6.15250e-20 y = +5.75553e-18 z = -7.88861e-21 (mag 5.756e-18)
   P.v. err: x =  9.12357e-06 y =  1.32118e-05 z =  7.44578e-07 (mag 1.607e-05)
   Vel disp: x =  4.28555e-04 y =  2.41602e-04 z =  1.81255e-04 (mag 5.243e-04)
   Max mass: 7 (7) mass 1.60e-10 (frac. of tot. 2.0e-02) max z 1.57855e-04
   Mean mass: 8.08e-11 (radius: 2.02e-06 Roche radius: 3.00e-04)
   Mean dist: 2.00e-02 (66.6 RR, 0.50 box_size) Mean min dist 3.65e-03 (12.2 RR)
   Max z: 51 (52) mass 8.00e-11 z -3.34247e-05 max z 1.12361e-04
   tzam/M: init -7.9770e-18 adj cur 1.6851e-03 diff 1.69e-03 rms 1.26e-03
   energy: init 0.0000e+00 adj cur -8.6507e-16 diff -8.65e-16 rms 6.54e-16
           (KE 0.00000e+00 (coll -8.65057e-16) RE 0.00000e+00 GPE 0.00000e+00)
   Other: <ecc> 3.66941e-02 hgt 1.66716e-04 <osc> 1.7414e-04
          FF(0) 0.00e+00 <c/p/o> 7.071e-02 mfp 2.335e+02 lv 1.716e-08
   Counters:
      Time-steps:         43777
      Min time-steps:     0
      Max time-steps:     6540
      Collisions:         7
      First-time col´ns:  1
      Ghost collisions:   0
      Mergers:            1
      Forced mergers:     1
      Box boundary xings: 206
      L-R boundary xings: 0
      Ghost box xings:    9
      Total mono updates: 52265
      Recur mono updates: 15874
      Total quad updates: 52160
      Recur quad updates: 19379
      Node packings:      0
      Force errors:       0
      Warnings:           2
      I/O errors:         0

   Collision statistics:

       Particle   Last collider  Nc  M/min z_max/<hgt>
      ----------- ------------- ---- ----- -----------
        7 (    7)    -1 (   31)    7 2.000    0.947

[END OUTPUT]
```

Figure A.1: Sample main periodic output from a `box_tree` run.

```
Status of evolving parameters, t = 1:
    Mag of vel disp       = 5.242939e-04
    Mean mass             = 8.080808e-11
    Median mass           = 8.000000e-11
    Mean radius           = 2.015420e-06
    Mean Roche radius     = 2.997629e-04
    CP min radial vel     = 5.242939e-06
    CP check zone         = 2.997629e-02
    Current tree size     = 4.000000e-02
    Self-grav check zone = 4.000000e-04
    Total CPU (min)       = 6.133333e+00
```

Figure A.2: Sample output of evolving `box_tree` parameters.

case of supplied initial conditions); the mean distance and mean minimum distance between particles in the central box; data regarding the particle with the current maximum absolute z (in the rotating frame, each particle also has a maximum z associated with its simple harmonic motion: for an oscillator, $\dot{z}^2 + \Omega^2 z^2 = E_{\text{osc}} = \Omega^2 z^2_{\text{max}}$); the initial total z angular momentum per unit mass, its adjusted current value, the difference between the adjusted value and the initial value, and the RMS error — see §4.4.2; the initial total energy, the adjusted current value, the difference, and the RMS error, followed by the total kinetic energy, the net loss in KE due to collisions, the total rotational energy, and the total gravitational potential energy — see §4.4.3; the mean eccentricity, scale height, mean z oscillation energy, filling factor at the midplane, mean collisions per particle per orbit, mean free path, and the local viscosity (many of these are undefined outside of the rotational frame); all the main counter variables; and any collision statistics.

In the example shown, particles 7 and 31 collided seven times before merging to form a new particle 7 (particle 31 was deleted). Note that particles are referred to first by their current index and second by their original index in parentheses. Thus the particle with the largest z value was originally particle 52 but is now particle 51 as a result of the merger. Also note that the run was performed in the rotating frame so the total energy is undefined, although the collisional loss in KE is still shown. The single warning referred to the fact that the initial TZAM was too close to zero to allow normalization of the TZAM errors.

Figure A.2 shows the status of the evolving parameters for the same run. Evolving parameters are simply statistical quantities that `box_tree` may use to make certain decisions (currently only a few of the parameters are actually used). For example, the detection zone for closest particle checks ("CP check zone") is currently set to 100 times the mean Roche radius, which will change as particles merge to form larger particles. Similarly, the minimum radial velocity before invoking the sliding phase ("CP min radial vel") is 0.01 times the mean velocity dispersion. The self-gravity check zone is 0.01 times the tree size (cf. §3.4.5). These quantities are updated at a rate determined by "Evol par interval" in the parameter file (cf. §A.4.2).

A.5.2 The Log File

Figure A.3 shows the log file generated for the sample run mentioned in the previous section. The most important compile- and run-time parameters are echoed to this file, along with the date the run was started, the machine used, and any major warnings. In this example, time stamps were generated every 10 000 steps, showing the real time, the simulation time, and the elapsed CPU. At the end, the completion date of the run is shown. Any fatal errors would also be recorded here. The log file is intended as a backup

```
Session begun  Sun Sep  5 14:06:48 1993 on theory0.ast.cam.ac.uk
Code compiled for optimization
Code compiled with gcc Sep  4 1993 15:34:40
Phys dim = 3, tree dim = 2
Parameters read from box_tree.par
box_tree rotating frame test: 3D N 100 w/ghosts w/tree
Reference frame = ROTATING
Length units = 1.49598e+11 m
Mass units = 1.989e+30 kg
Time units = 3.15581e+07 s
Random number seed = 7248
@box_tree -- major warning in calc_data(): tzam ~ 0; error unnormalized.
Sun Sep  5 14:06:49 1993 TIME = 0.000e+00 CPU min = 0.000e+00 Nsteps = 0
Sun Sep  5 14:08:42 1993 TIME = 2.881e-01 CPU min = 1.700e+00 Nsteps = 10000
Sun Sep  5 14:09:45 1993 TIME = 4.015e-01 CPU min = 2.533e+00 Nsteps = 20000
Sun Sep  5 14:11:07 1993 TIME = 6.092e-01 CPU min = 3.783e+00 Nsteps = 30000
Sun Sep  5 14:13:09 1993 TIME = 8.976e-01 CPU min = 5.500e+00 Nsteps = 40000
Run completed Sun Sep  5 14:13:53 1993.
```

Figure A.3: Sample box_tree log file.

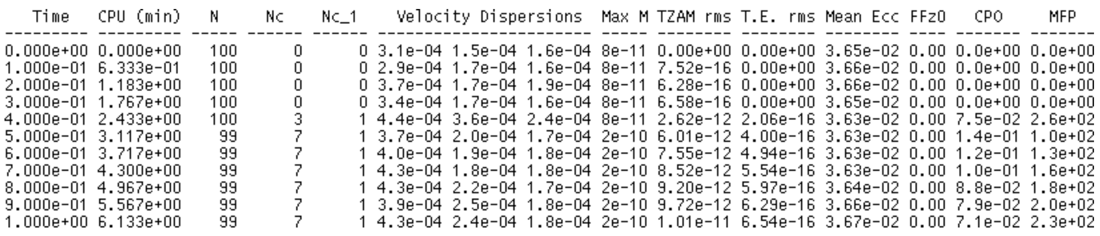
Table A.4: Stats file format.

Data	Format	Data	Format
simulation time*	double	TZAM	double
elapsed CPU*	double	TZAM error (diff)	double
no. of time-steps	int	TZAM RMS error*	double
no. particles*	int	total energy	double
no. collisions*	int	collisional ΔT	double
no. 1st-time coll's*	int	total energy error (diff)	double
c-o-m position	$3 \times$ double	total energy RMS error*	double
c-o-m velocity	$3 \times$ double	mean eccentricity*	double
velocity dispersion*	$3 \times$ double	scale height	double
total mass	double	midplane filling factor*	double
maximum mass	double	collisions/particle/orbit*	double
max. mass position	$3 \times$ double	mean free path*	double
max. mass velocity	$3 \times$ double	local viscosity	double
max. mass spin	$3 \times$ double		

in case the standard output is lost or deliberately discarded.

A.5.3 The Stats File (stats_read)

The stats file contains a summary of information displayed in the main periodic output (§A.5.1). New information is appended to the end of the file as the run progresses. For maximum efficiency, the file is in binary; Table A.4 shows the format (read down the first column then down the second column for the correct byte ordering). Note that the beginning of the file contains the first MAX_STR_LEN (usually 256 — see params.h header file) bytes of the comment string. In order to access the data for plotting, the file needs to be converted to ASCII text format. The auxiliary program stats_read (included in the box_tree distribution) was written for this purpose. Given the name of a parameter file and optionally the name of the statistics file (default box_tree.stats), stats_read reads the binary data and generates ASCII output in a format specified by the supplied parameters. Two parameter files, stats_read_long.par and stats_read_short.par, are included with the distribution. The former causes stats_read to extract all the infor-



```
   Time    CPU (min)   N     Nc     Nc_1     Velocity Dispersions   Max M TZAM rms T.E. rms Mean Ecc FFz0   CPO     MFP
--------- --------- ----- ------ ------ ---------------------- ----- -------- -------- -------- ---- ------- -------
0.000e+00 0.000e+00  100      0      0 3.1e-04 1.5e-04 1.6e-04 8e-11 0.00e+00 0.00e+00 3.65e-02 0.00 0.0e+00 0.0e+00
1.000e-01 6.333e-01  100      0      0 2.9e-04 1.7e-04 1.6e-04 8e-11 7.52e-16 0.00e+00 3.66e-02 0.00 0.0e+00 0.0e+00
2.000e-01 1.183e+00  100      0      0 3.7e-04 1.7e-04 1.9e-04 8e-11 6.28e-16 0.00e+00 3.66e-02 0.00 0.0e+00 0.0e+00
3.000e-01 1.767e+00  100      0      0 3.4e-04 1.7e-04 1.6e-04 8e-11 6.58e-16 0.00e+00 3.65e-02 0.00 0.0e+00 0.0e+00
4.000e-01 2.433e+00  100      3      1 4.4e-04 3.6e-04 2.4e-04 8e-11 2.62e-12 2.06e-16 3.63e-02 0.00 7.5e-02 2.6e+02
5.000e-01 3.117e+00   99      7      1 3.7e-04 2.0e-04 1.7e-04 2e-10 6.01e-12 4.00e-16 3.63e-02 0.00 1.4e-01 1.0e+02
6.000e-01 3.717e+00   99      7      1 4.0e-04 1.9e-04 1.8e-04 2e-10 7.55e-12 4.94e-16 3.63e-02 0.00 1.2e-01 1.3e+02
7.000e-01 4.300e+00   99      7      1 4.3e-04 1.8e-04 1.8e-04 2e-10 8.52e-12 5.54e-16 3.63e-02 0.00 1.0e-01 1.6e+02
8.000e-01 4.967e+00   99      7      1 4.3e-04 2.2e-04 1.7e-04 2e-10 9.20e-12 5.97e-16 3.64e-02 0.00 8.8e-02 1.8e+02
9.000e-01 5.567e+00   99      7      1 3.9e-04 2.5e-04 1.8e-04 2e-10 9.72e-12 6.29e-16 3.66e-02 0.00 7.9e-02 2.0e+02
1.000e+00 6.133e+00   99      7      1 4.3e-04 2.4e-04 1.8e-04 2e-10 1.01e-11 6.54e-16 3.67e-02 0.00 7.1e-02 2.3e+02
```

Figure A.4: Sample short form output from `stats_read`.

mation from the statistics file and dump it in long integer and exponential text format to the screen (`stdout`). With the latter file, `stats_read` extracts a subset of the fields (indicated by asterixes in Table A.4) and prints the information in formatted columns with a header line to the screen. Figure A.4 is an example of the short form generated from the run discussed in the previous two sections. Note that there are extra switches in these parameter files to do things like print only every nth line, etc. In addition, the ability to exclude certain columns when reading is provided for backwards compatibility with older versions of `box_tree`. See the source code in the distribution for more details.

It is convenient to define the following `csh` aliases to facilitate the use of `stats_read`:

```
alias srs '$bt_dir/Util/Stats_Read/stats_read{,_short.par}'
alias srl '$bt_dir/Util/Stats_Read/stats_read{,_long.par}
```

where in this example `bt_dir` is a shell variable containing the path name of the `box_tree` source directory. Typing `srs` at any time during a run will generate the short form output to the screen. Use `srl` to generate the long form. It may be helpful to append "`>! stats.out`" at the end of the `srl` alias to automatically redirect output to a file called `stats.out`.

The long form of output is recommended for use with plotting packages. Included in the `box_tree` distribution is a set of `sm` macros specifically designed to plot the stats data in an X11 window (and optionally generate hardcopies or postscript versions of the graphs). A `csh` executable script called `stats_plot` is provided to automatically invoke `sm` and plot the graphs. There are three files containing the `sm` commands and macros: `stats_plot.sm`, which is read in by `sm` and executed a line at a time; `stats_plot.sm.init`, which contains macros for reading in the stats data; and `stats_plot.sm.macros`, which contains the actual plotting macros. The stats file is assumed to have the name `stats.out` and to reside in the plot directory (it is good practice to copy—rather than move—the stats file for this purpose). The plots are self-explanatory, although the user may find it helpful to examine the macro files for details. Note that the macros also generate helpful equilibrium data in text form, complete with error bars (standard deviation of the mean) for the most important quantities. The time range to be used may be specified in `stats_plot.sm.init`. This is useful for generating equilibrium statistics (cf. §5.3). Finally, the data fields to be used and the maximum range of the various quantities can also be set in `stats_plot.sm.init`.

A.5.4 Data Files (`dat_read`)

The particle data files contain binary information regarding each particle in the simulation at a specific time. The files are output periodically at a rate determined by "Dat interval" in the parameter file (cf. §A.4.2). The files are numbered sequentially, starting with `box_tree000.dat` by default. The first `MAX_STR_LEN` bytes of each file contain the header string. This is followed by a double precision value containing the simulation time at output. The remainder of the file contains N lines of binary data of the form described in §A.4.3 (for the supplied initial conditions file), using double or int format as appropriate.

Since the data files are in binary to save space (and to preserve the data precision), the program `dat_read` was written to convert them to ASCII in various formats. This program works much the same way as `stats_read` described above, but there are important differences. Since there are usually many data files, `dat_read` accepts a list of files after the parameter file argument. In addition, the ASCII output is written to individual files rather than to the screen, starting with `dat000.out` by default. There are four parameter files supplied with `dat_read`: `dat_read_long.par` to generate long form output; `dat_read_movie.par` to create files suitable for reading by `make_movie` (cf. §A.5.5); `dat_read_short.par` for short form output; and `dat_read_stats.par` to perform various statistical operations on the data files and to generate binned data.

As for `stats_read`, it is convenient to define aliases to simplify the use of these parameter files:

```
alias cpd 'cp $bt_dir/Util/Dat_Read/dat_read_stats.par .'
alias drd '$bt_dir/Util/Dat_Read/dat_read dat_read_stats.par'
alias drl '$bt_dir/Util/Dat_Read/dat_read{,_long.par}'
alias drm '$bt_dir/Util/Dat_Read/dat_read{,_movie.par}'
alias drs '$bt_dir/Util/Dat_Read/dat_read{,_short.par}'
```

The first alias copies the `dat_read_stats.par` file into the current directory. This is useful because when generating statistics there are a few extra parameters that must be specified that depend on the run in question. Rather than editing the original parameter file each time `dat_read` is used on the same data set, it makes sense to make a copy of the file and use the copy instead. Hence the `drd` alias assumes the `dat_read_stats.par` file resides in the current run directory. The remaining aliases are of the same form as those defined for `stats_read`. It may be convenient to add the fragment:

```
\!*; if ($status == 0) more dat???.out
```

to the end of the `drs` alias to display the short form output on the screen.

The long and short form output are self-explanatory. The movie output will be discussed in the following section. The statistics and binned data output, however, require more explanation. Basically, `dat_read` collects statistics over the range of files (and hence the range of time) supplied on the command line. A restricted range of masses to consider can be specified in `dat_read_stats.par`. The statistics include the mean velocity dispersions, the mean particle spin and obliquity, and the mean z excursion, complete with the usual error estimates. This output was used to generate much of Table 5.2 for example. In addition to these statistics, binned data is output in ASCII form to a file called `bin_stats.out`. This data was used to generate the various histograms seen in §5.3, such as the particle number density as a function of height above and below the midplane.

Again as for `stats_read`, there are `sm` macros included in the source distribution to automatically generate plots from the files output by `dat_read`. The `dat_plot` set takes a single long form file (`dat000.out` by default) and plots such things as the mass, velocity, and spin distributions. Useful phase space plots are also generated. The `bin_plot` set of macros takes the `bin_stats.out` file and plots the associated histograms. Finally, column labels and grouping data for use with the 3D data visualizer `xgobi` are also included (the data file is assumed to be `dat000.out` and must be in long format).

A.5.5 Movie Frames (make_movie, xrastool)

One of the best ways of visualizing the evolution of complex dynamical systems is to use graphical animation, or movies. To this end, a considerable amount of effort was devoted to the development of `xrastool` (originally `rastool`), a utility for loading movie frames into memory and displaying them rapidly in sequence. Currently `xrastool` supports full-colour Sun rasterfiles of arbitrary size, and can achieve a display rate in excess of 50 frames per second for 400×400 pixel 8 bit images on a Sparc IPX or Sparc 10 running an X window manager such as `twm`. The advantage of loading entire frames is that the display rate is limited only by the size of frame, whereas an animation tool that displays particles one at a time is limited by the total particle number. In addition, the frames can be made as complex as desired, using shaded spheres for particles and including the infrastructure of the tree for example. The other major advantage is that `xrastool` can be used as a stand-alone animator for other applications; it is not specialized for use with `box_tree`. Full details of `xrastool` are provided with the source distribution. In fact, `xrastool` has already been released to the public domain, so the distribution includes a separate man page for the utility.

Movie frames for use with `xrastool` may be generated by `box_tree`, in much the same way as particle data files (i.e. , one file per frame, starting with `movie000.ras` by default). The parameters for customizing the movie frames have already been discussed in §A.4.2. A typical default frame shows the entirety of the central box seen from directly above the xy-plane. Particles are represented by shaded spheres of varying size depending on the particle radii and the apparent distance from the observer. Individual particles may be assigned specific colours if desired. If the tree is drawn, each cell is represented by an open square bordered in white. Centre-of-mass positions are shown by blue diamonds, with blue lines connecting the centres of mass of parent cells with their children. Red lines connect leaf particles to their parents. The centre-of-mass position of the root node is drawn in pink to distinguish it from the rest. The choice of structures to be drawn and their colours currently can only be changed in the source code itself (cf. `draw.c`). Note that if ghost boxes are included in the picture (open squares bordered in yellow), the tree is only drawn in the central box to minimize the drawing complexity, but would look the same in each ghost box.

In certain cases it is desirable to view the particles from a different direction (especially in 3D), or to use a larger view size if many tree expansions have occurred in an unbounded simulation (for example). It would be completely impractical to re-run `box_tree` simply to generate new movie frames, so the program `make_movie` may be used instead. This program reads ASCII particle data files in the format output by `dat_read` using the `dat_read_movie.par` parameter file (see previous section). The file format is simply $x\ y\ z\ R\ c$ (in the notation of §A.4.3), so any application that generates particle data, not just `box_tree`, could make use of `make_movie`. Indeed this was the original motivation for writing the utility.

As with all of the auxiliary programs discussed so far, `make_movie` has its own parameter file for adjusting the default behaviour. The file is called `make_movie.par` and it is good practice to copy the file into the run directory and modify it there, rather than editing the original. Since there is just the one parameter file, there is no need to specify it as a command line argument (although the "-p" argument option is included in case a name other than `make_movie.par` is used). The parameters themselves are fairly self-explanatory. First a frame size in pixels is specified. Next the minimum and maximum values in all three Cartesian directions must be given (it is too time consuming to search all the files to determine the data range automatically). The remaining options include the particle shapes to use, the colormap template (use option 3 for the colours listed in Table A.3), and the viewing direction (along any of the three axes). There are also options to disable reading of the radius and/or colour fields if they are not present in the data files.

The make_movie program takes a list of files on the command line, and generates the movie frames sequentially, starting with mm000.ras by default. As an example, the following sequence of commands might be used to generate movie frames from the binary particle data files created during a typical box_tree run:

```
drm box_tree*.dat
make_movie dat*.out
```

The movie frames could then be loaded for viewing with:

```
xrastool -fast mm*.ras
```

It is assumed that all the necessary data and parameter files reside in the run directory in this example. Also, the make_movie and xrastool commands are assumed to be in the execution path.

A.5.6 Save Files and Restarts

As explained previously, save files may be dumped to disk periodically if desired to guard against the unexpected termination of a run. In addition, a save file is generated automatically when box_tree terminates without error at the end of a run. In order for runs to be reproducible from restarts, it is necessary that *all* global program variables and parameters that may change from run to run be saved. To minimize the risk of round-off error, the data must be saved in binary format (this also reduces the file size considerably). Further, the parameters set in params.h are also saved to ensure that this file is not changed between restarts.

Fortunately, it is relatively straightforward to dump all of the relevant program variables. This is because the vast majority of the variables and parameters are contained in structures of known size that can be written out to disk with a single operation. This includes the N structures containing all the particle data. Saving the tree is slightly more complex, requiring a recursive procedure that writes out one node at a time starting from the root node. The procedures for reading the save file for restarts are analogous: simple read commands to retrieve the structures followed by a recursive procedure to load the tree. When reading, memory is allocated as required for the particle and node data. Further details can be obtained by examining the main header file box_tree.h, which contains the structure definitions, and the routines SaveRestartData() and ReadRestartData() in misc.c, which perform the save and read operations.

To restart a run from the last save file, use the command `box_tree -r`. Almost everything will proceed as before the restart, with the minor exception of the TZAM and total energy error statistics, which currently are reset following a restart for technical reasons. Also, those files to which data are appended (namely the stats and NLV files) may end up with redundant lines of data since the restart may begin some time before the last output to these files. The stats_read program automatically accounts for this by keeping track of the simulation time of each data line; if the current entry is found to be for the same time as, or for a time previous to the last entry, the current line and all subsequent lines are skipped until genuinely new data is detected. Currently there is no provision to do this with NLV files. Note that box_tree will generate an extra main output (§A.5.1) at the beginning of a restart so that the save file integrity can be verified. Periodic output will then continue at the same time multiples as before the restart.

A.6 Warnings and Error Conditions

There are currently seven categories of warnings and error conditions, namely major warnings, minor warnings, I/O errors, fatal I/O errors, fatal errors, halts, and system errors. Major warnings and fatal errors are echoed to `stderr` and the log file (cf. §A.5.2) while minor warnings and non-fatal I/O errors just appear on `stdout`. The "halt" error is not actually an error; it is generated when a run is legitimately interrupted by `<CTRL><C>` or the `STOP` file (cf. §A.3).

There are many types of warnings, varying from a message if θ_C is deemed too large, to notices that node packing has been invoked. Output of warnings can be controlled through the use of the verbosity and debug level parameters (cf. §A.4.2). Warnings do not terminate execution.

An I/O error is generated if `box_tree` is unable to open, read from, or write to a file. Some I/O errors, depending on the importance of the file concerned, are considered fatal, and result in program termination. Usually, however, the error is ignored and the attempted operation is skipped. Thus if disk space runs out for output, `box_tree` will not crash. However, some data will inevitably be lost. Of course, it is possible that because the disk is full, `box_tree` will be unable to advise the user of the error, so it is a good idea to check periodically to make sure there is no danger of running out of disk space during a run. Note that `box_tree` uses the `perror()` system call to tell the user the exact nature of the I/O error if known.

Fatal errors can occur as a result of internal `box_tree` checks (some of which can be controlled through the debug level). For example, an invalid setting in the `box_tree` parameter file will generate a fatal error, terminating the run. Most error messages are self-explanatory; in the case of parameter errors, for example, the error message contains a brief explanation of why the parameter is invalid. Note that the parameter file parser `rdpar` has its own built-in error traps for bad syntax or missing keywords; these will override the `box_tree` error handling routines (see source in §B.1).

A system error is also fatal. These can occur as the result of an arithmetic trap, a segmentation fault, or a BUS error. These generally mean that something is wrong with the code and any manifestations of these errors should be reported to the author. Note that buffering of `stdout` is disabled so that the output should be up to date at the time of the error, which will help in locating the problem. If possible, a debugger should be used to try to reproduce the error under controlled conditions. Often this will expose the offending line of code; the debugger can then be used to examine the values of any relevant variables.

In order to make warnings and errors easy to find and interpret, they all take on a similar form. For example, he following message is generated if an attempt is made to define a negative length scale in the parameter file:

```
Reading parameter file "box_tree.par"...

@box_tree -- fatal error in GetParams(): Invalid choice/bad syntax.
   (length scale must be positive).

*** FATAL ERROR in box_tree
Program halted at t = 0.00000e+00 (CPU 0.000e+00 min this run, 0 steps).
IOT trap (core dumped)
```

The main error message consists of the program name (`box_tree`), the error type (fatal), the routine in which the error occurred (`GetParams()`), and the nature of the error (in-

Table A.5: Miscellaneous supporting code, scripts, and macros.

Item	Description
`quad.c`	simple multipole tester used to generate Fig. 4.5.
`nlv_read.c`	code to calculate NLV data from from NLV file (cf. §5.3).
`error` (`csh` script)	script for automating tests of ϵ_M & ϵ_Q (cf. §4.3)
`error.awk`	`awk` file used in conjunction with `error`.
`timing` (`csh` script)	script for automating timing tests (cf. Fig. 4.1 & 4.2).
`timing.awk`	`awk` file used in conjunction with `timing`.
`wt` (`csh` script)	script for automating WT comparison tests (cf. §5.3).
`e_plot` (`csh` script)	script for using `sm` to generate plots from output of `error`.
`e_plot.sm`	`sm` file used in conjunction with `e_plot`.
`e_plot.sm.macros`	`sm` file used in conjunction with `e_plot`.
`t_plot` (`csh` script)	script for using `sm` to generate plots from output of `timing`.
`t_plot.sm`	`sm` file used in conjunction with `t_plot`.
`t_plot.sm.macros`	`sm` file used in conjunction with `t_plot`.
`results` (`csh` script)	script for using `sm` to generate WT comparison plots (cf. `wt`).
`results.sm`	`sm` file used in conjunction with `results`.
`results.sm.macros`	`sm` file used in conjunction with `results`.

valid parameter). An optional subsidiary message in parentheses gives more information ("length scale must be positive"). If the error was fatal, as it was in this case, a termination message is printed. A core dump will also be produced by `box_tree` if possible (to disable core dumps under `csh`, type "`limit coredumpsize 0`" at any shell prompt or in a startup file). Note the "@" symbol in the error message. This symbol is prepended to any error message sent to `stdout` or the log file to allow for easy detection of warnings or other non-fatal errors (e.g. "`grep @ output`" under `csh`). Also note that, in the case of major warnings or fatal errors, messages are sent to both `stdout` and `stderr`. Thus if these I/O streams are directed to the same place, the error message will be duplicated. Finally, major warnings and fatal errors have a "beep" code (`<CTRL><G>`) imbedded in them to attract attention.

A.7 Miscellaneous Code, Scripts, and Macros

There are other pieces of code and various scripts and macros in the `box_tree` source distribution that have not been described. Some of these are out of date and are incompatible with current output formats. They are included both for historical reasons and to serve as templates for building new `box_tree` interfaces. Table A.5 briefly describes these miscellaneous items.

A.8 Test Suite

It is helpful to maintain test data with a complex program so that any problems that develop during code development may be identified quickly. In addition to the default parameter file, which instructs `box_tree` to perform a simple planetesimal run for one orbit, there are compressed tar files in the source distribution containing parameter files and initial conditions for four separate tests. The files are called `collide`, `orbit`, `pool`, and `pythag` (omitting the "`.tar.Z`" extension). To perform one of these tests, first unpack the associated files. For example,

```
zcat pythag.tar.Z | tar xvf -
```

extracts the files pythag.dat, box_tree.par, and README and places them in a subdirectory called pythag. Next cd pythag and copy the box_tree executable into the new subdirectory. Run the test with:

```
nice +19 box_tree >&! output &
```

Note that some of the tests (pythag in particular) may generate a large number of files. Follow the instructions given in §A.5 for examining the various forms of output. The README file accompanying each test package describes the expected behaviour, and may include a TZAM and/or total energy conservation check for comparison. For reference, the test packages will be described briefly below.

The collide test was used extensively while developing the collision code. The simulation starts with seven equal-mass particles at rest in the $z = 0$ plane. The particles begin to accelerate towards the centre of mass and eventually bounce into each other. The bounces are nearly elastic ($\epsilon_n = 0.9$, $\epsilon_t = 1.0$), but nevertheless the particles converge rapidly to the centre. The final configuration should have no linear or angular momentum.

The orbit test is a very simple two-body simulation with a mass ratio of 1 000:1. The README file lists the various starting velocities for the smaller particle that will result in circular, elliptical, parabolic, or hyperbolic orbits. Refer to §A.4.3 for the initial conditions file format.

The pool test is another collision checker, this time with spin included but no interparticle gravity. For variety, stationary ghost boxes are drawn as well.

The final test, pythag, reproduces the classic three-body simulation known as the "Pythagorean Problem". The movie output should be compared with the trajectories drawn by Szebehely & Peters (1967). Since box_tree conserves the total energy in this run to better than 1 part in 10^8, the agreement should be essentially exact. Note that Szebehely & Peters used a form of two-body regularization to accurately integrate over the several very close encounters that occur in the simulation. Regularization is not supported in box_tree, so a very small time-step coefficient is needed instead. However the entire simulation still takes less than 1 CPU minute to complete on a Sparc 10/51.

Appendix B

Source Listings

This appendix contains the complete source listings for `box_tree` (§B.1) as well as `rdpar` (§B.2). The code is presented a file at a time with introductory comments as appropriate. The source listings of the other auxiliary programs used with `box_tree` are not presented here but are available in the `box_tree` distribution.

B.1 `box_tree`

The `box_tree` code is made up of 3 header files and 17 source ("`.c`") files. The header files will be presented first, in order of usage, followed by `box_tree.c`. The remaining files are listed in alphabetical order.

B.1.1 `box_tree.h`

This is the main `box_tree` header file. It loads in other header files (both system files and the remaining `box_tree` header files), defines many preprocessor constants (aliases), constructs new variable types, and declares all the global (external) variables and functions. This file is included by each "`.c`" file, so a change here requires recompilation of the entire source.

Most of the coding conventions used in `box_tree` are illustrated in `box_tree.h`. Preprocessor aliases and macros are all in upper case, with words separated by underscores (e.g. `NUM_COUNTERS` and `POW2()`). New type definitions are also in upper case, with "`_T`" appended (e.g. `NODE_T`). The `BOOLEAN` type is the only exception to this rule. Global variables and functions are capitalized; if more than one word makes up the name, all the words are capitalized and no separator is used (e.g. `RunPar` and `UpdateBoxPos()`). Automatic variables and functions (those that are most limited in scope) are all in lower case, with words separated by underscores (e.g. `ref_frame` and `box_tree()`). When declaring variables and functions, the following order of precedence is used: `void`, `int` (loop variables and other simple indices first), any defined type (e.g. `DATA_T` or `NODE_T`), `char`, `double`, and `BOOLEAN`. Other conventions (such as indentation and placement of comments) are easily seen from the file itself.

The header file is fairly self-explanatory. Note that some of the aliases defined in `params.h` (listed below) are used here (e.g. MAX_NUM_PARTICLES). Some use is made of conditional preprocessor statements for portability reasons.

```
/*
 * box_tree.h - DCR 91-04-30
 * ==========================
 * Main header file for box_tree: includes, definitions, and declarations.
 *
 */

/* Copyright notice... */

#include "COPYRIGHT"

/* Other header files to include... */

#include <stdio.h>                                            /* Standard I/O definitions */
#include <string.h>                               /* Definitions for string handling functions */
#include <math.h>                                                         /* Math definitions */
#include <malloc.h>                             /* Definitions for memory allocation functions */
#include "params.h"                                  /* Main box_tree compile-time parameters */
#include "macros.h"                                                /* Various box_tree macros */

/* Basic definitions... */

#define NUM_COUNTERS 19                          /* Miscellaneous counters (c.f. Counter[]) */

#define TIME_STEPS 0                                           /* Total number of time-steps */
#define MIN_TIME_STEPS 1                                    /* Number of minimum time-steps */
#define MAX_TIME_STEPS 2                                    /* Number of maximum time-steps */
#define COLLISIONS 3                                           /* Total number of collisions */
#define FIRST_TIME_COLLISIONS 4                           /* Number of first-time collisions */
#define GHOST_COLLISIONS 5                                     /* Number of ghost collisions */
#define MERGERS 6                                                 /* Total number of mergers */
#define FORCED_MERGERS 7                                         /* Number of forced mergers */
#define BNDRY_XINGS 8                                  /* Total number of boundary crossings */
#define LATERAL_BNDRY_XINGS 9                           /* Number of lateral (x) bndry crossings */
#define GHOST_BOX_BNDRY_XINGS 10                     /* Number of ghost box boundary crossings */
#define TOTAL_MONO_UPDATES 11                           /* Total number of monopole updates */
#define RECUR_MONO_UPDATES 12                       /* Number of recursive monopole updates */
#define TOTAL_QUAD_UPDATES 13                         /* Total number of quadrupole updates */
#define RECUR_QUAD_UPDATES 14                      /* Number of recursive quadrupole updates */
#define PACKINGS 15                                               /* Number of node packings */
#define FORCE_ERRORS 16                                       /* Number of large force errors */
#define WARNINGS 17                                                    /* Number of warnings */
#define IO_ERRORS 18                                        /* Number of non-fatal I/O errors */

#define NUM_TIMERS 6                           /* Timers for event processing (c.f. CLOCK_T) */

#define OUTPUT 0                                                              /* Main output */
#define STATS 1                                                        /* Statistics summary */
#define DAT 2                                                        /* Particle data output */
#define EVOL 3                                                        /* Evolving parameters */
#define MOVIE 4                                                        /* Movie frame output */
#define CHECK 5                                                        /* Debug/check output */

#define ROTATING 1                                          /* Labels for reference frames */
#define INERTIAL 2
#define GALAXY 3

#define ALIGNED_COM 1                                  /* Labels for initial conditions options */
#define UNIFORM_RAN 2
#define WT 3
#define CLOSE_PACKED 4
#define SUPPLIED 5

#define PERIODIC 1                                    /* Labels for boundary conditions options */
```

```
#define UNBOUNDED 2
#define DISABLED 3

#define RV_ONLY 1                               /* Labels for time-step formula options */
#define RV_AND_F 2
#define F_ONLY 3

#define BHL_FLAG -9.9                   /* Coef of rest flag ==> BHL formula should be used */

#define BC_NONE 0                           /* Flags for ApplyBndryCond() in bndry_cond.c */
#define BC_BOX 1
#define BC_TREE 2

#define DRAW_TREE POW2(0)                              /* Bit flags for Draw() in draw.c */
#define DRAW_BOXES POW2(1)
#define PLOT_POS POW2(2)
#define PLOT_VEL POW2(3)
#define PLOT_COM POW2(4)
#define COM_LINES POW2(5)

#define NO_UPDATE 0                           /* Flags for PlaceInTree() in make_tree.c */
#define UPDATE 1
#define UPDATE_CHILDREN 2

#define WARNING1 0                                      /* Flags for Error() in misc.c */
#define WARNING2 1
#define IO 2
#define FATAL_IO 3
#define FATAL 4
#define HALT 5
#define SYS_ERR 6

#define ALL_DONE 0                                  /* Flags for Terminate() in misc.c */
#define ERROR 1
#define USER_HALT 2

#define SUBM -1                 /* Directives for UpdateBranchMoments() in update_tree.c */
#define ADDM 1

#define NO_PRED 0                          /* Particle position and velocity prediction flags */
#define UN_PRED 1
#define LO_PRED 2
#define HI_PRED 3

#define CP_UNDEF -1               /* For cp structure (e.g. set_time_step() in integrate.c) */

#define RED 1                                              /* Bright colors for drawing */
#define PINK 2
#define YELLOW 3
#define GREEN 4
#define ORANGE 5
#define PURPLE 6
#define CYAN 7
#define BLUE 8

#define BLACK 0                                   /* Black should ALWAYS be defined as 0 */

#define FIRST_GRAY 9                              /* Fill rest of colormap with gray scale */
#define LAST_GRAY 255
```

```
#define WHITE LAST_GRAY                    /* White should ALWAYS be defined as 255 */

#define DENSITY_CGS_TO_MKS 1.0e3           /* Density conversion factor from cgs to mks */

#define PI 3.141592653589793                                /* Pi to 15 decimal places */
#define TWO_PI 6.283185307179586                                      /* Ditto for 2 Pi */

#ifdef ALPHA
# undef HUGE_VAL
# define HUGE_VAL MAXFLOAT                 /* Alphas don't define HUGE_VAL (infinity) */
#endif

/* Useful abbreviations... */

#define ROTATING_FRAME (RunPar.ref_frame == ROTATING)
#define INERTIAL_FRAME (RunPar.ref_frame == INERTIAL)
#define GALAXY_FRAME (RunPar.ref_frame == GALAXY)

#define GHOSTS (NumBoxes > 1)

#define BOX_SIZE RunPar.box_size
#define HALF_BOX_SIZE RunPar.half_box_size
#define SYS_SIZE RunPar.sys_size
#define HALF_SYS_SIZE RunPar.half_sys_size
#define SYS_CENTRE RunPar.sys_centre
#define TREE_SIZE TreePar.tree_size
#define HALF_TREE_SIZE TreePar.half_tree_size
#define TREE_CENTRE SYS_CENTRE
#define VIEW_SIZE MoviePar.view_size
#define HALF_VIEW_SIZE MoviePar.half_view_size
#define VIEW_CENTRE MoviePar.view_centre
#define BOX_X_OFFSET RunPar.box_x_offset
#define BOX_Y_OFFSET RunPar.box_y_offset
#define BOX_POS RunPar.box_pos
#define BOX_VEL RunPar.box_vel

/* Current clock time with 2 pi scaling removed for rotating frame */

#define TIME (ROTATING_FRAME ? Clock.time / TWO_PI : Clock.time)

/*
 * For maximum optimization, replace following macros with FALSE.
 * e.g. #define VERBOSE FALSE, #define ERROR_CHECK FALSE, etc.
 *
 */

#define VERBOSE (RunPar.verbosity_level > 0)
#define VERY_VERBOSE (RunPar.verbosity_level > 1)

#define ERROR_CHECK (RunPar.debug_level > 0)
#define MONITOR (RunPar.debug_level > 1)
#define TRACK (RunPar.debug_level > 2)

/* Simple type definitions... */

/* Boolean type */

#define BOOLEAN int                  /* Do it this way to avoid enumeration problems */

#define FALSE 0                                              /* e.g. result of (0 == 1) */
```

```
#define TRUE 1                                        /* e.g. result of (0 == 0) */

/* Definitions of NODE_T, LEAF_T, and BRANCH_T, used in struct node_s below */

typedef struct node_s NODE_T;

#define LEAF_T int
#define BRANCH_T NODE_T

/* Tree cell descriptors */

typedef enum {EMPTY, BRANCH, LEAF} CELL_T;

/* Object shapes for movies */

typedef enum {POINT, CIRCLE, SQUARE, DIAMOND, DISK, SPHERE} SHAPE_T;

/* Color type definition */

typedef unsigned char COLOR_T;

/* Complex (structure) type definitions... */

/* Def'n of random number generator data structure type (c.f. RunPar.ran) */

typedef struct {
    int seed;                                         /* For Ran() in recipes.c */
    long int ix1, ix2, ix3;
    double r[98];
    int iset;                                         /* For Gasdev() in recipes.c */
    double gset;
} RAN_T;

/* Def'n of coeff. of restitution data structure type (c.f. RunPar.rest_coef) */

typedef struct {
    double radial;
    double transverse;
} REST_COEF_T;

/* Definition of gas drag data structure type (c.f. RunPar.drag_coef) */

typedef struct {
    double x;
    double y;
    double z;
    double hdot;
} DRAG_COEF_T;

/*
 * Definition of run parameters structure type (one only, viz. RunPar).
 * The structure is loosely patterned on the run-time parameter file
 * (e.g. box_tree.par) and is therefore NOT in optimum alignment. Physical
 * parameters (e.g. RunPar.box_size) are in scaled units unless otherwise
 * indicated.
 *
 */

typedef struct {
```

```
char comment_line[MAX_STR_LEN];                    /* Simple user-supplied comment */

/*
 * The following parameters are read or calculated only once. Note that
 * some parameters may not be used, depending on the choice of reference
 * frame and initial conditions. Recall that some run parameters are
 * global variables for convenience (notably NumBoxes & NumParticles).
 *
 */

int ref_frame;                                                   /* Reference frame */
double length_scale;                                           /* Length scale in m */
double mass_scale;                                              /* Mass scale in kg */
double time_scale;                                               /* Time scale in s */
double velocity_scale;                  /* Velocity scale (may include factor of 2 pi) */
double density_conv;                   /* Density conversion from mks to scaled units */
RAN_T ran;                                             /* Random number data storage */
int ic_opt;                                               /* Initial conditions option */
int bc_opt;                                              /* Boundary conditions option */
double box_size;                                                        /* Box size */
double half_box_size;                                             /* Half width of box */
double sys_size;                                                     /* System size */
double half_sys_size;                                          /* Half width of system */
double sys_centre[NUM_BOX_DIM];              /* Centre of system (currently always 0) */
double init_clock_time;                                          /* Initial clock time */
double init_x_vel_disp;                                     /* Initial x-vel dispersion */
double init_y_vel_disp;                                     /* Initial y-vel dispersion */
double init_z_vel_disp;                                     /* Initial z-vel dispersion */
BOOLEAN small_disp;                          /* TRUE for "small" initial dispersions */
double init_min_mass;                                         /* Smallest initial mass */
double init_max_mass;                                          /* Largest initial mass */
double density;                                   /* Particle density (in scaled units) */
double mass_exponent;                                      /* Mass function exponent */
double seed_mass;                              /* Initial mass of particle 0 if different */
BOOLEAN use_softening;                    /* TRUE if radii are to be used as softenings */
BOOLEAN rej_init_bin;                            /* TRUE to reject initial binaries */
double init_scale_height;                                /* Initial vertical scale height */
int num_x_div;                          /* Num x divisions for unif. random init. cond. */
int num_y_div;                          /* Num y divisions for unif. random init. cond. */
double optical_depth;                                       /* Dynamic optical depth */
double init_disk_thickness;                  /* Initial disk thickness in particle radii */
int num_layers;                         /* Number of particle layers for close-packing */
BOOLEAN expand_radii;                       /* TRUE to expand radii for close-packing */
BOOLEAN stagger_in_z;                       /* TRUE to stagger close-packed planes */
char init_cond_filename[MAX_FILENAME_LEN];                   /* Filename for supplied IC */
int num_header_lines;                     /* Number of header lines to skip in IC file */
BOOLEAN add_shear;                          /* TRUE to add shear to supplied velocities */
double total_mass;                          /* Total mass of system (currently constant) */
double box_x_offset[MAX_NUM_BOXES];                          /* x offsets of ghost boxes */
double box_y_offset[MAX_NUM_BOXES];                          /* y offsets of ghost boxes */
double box_pos[MAX_NUM_BOXES][NUM_BOX_DIM];                  /* Position of ghost boxes */
double box_vel[MAX_NUM_BOXES][NUM_BOX_DIM];                  /* Velocity of ghost boxes */

/* Remaining parameters can be changed every restart */

int verbosity_level;                                                 /* Verbosity level */
int debug_level;                                                         /* Debug level */
long int stop_check;                            /* STOP check interval (in time-steps) */
long int cpu_check;                                               /* CPU check interval */
long int safety_dump;                                           /* Safety dump interval */
```

```
    long int time_stamp;                                         /* Time stamp interval */
    double interval[NUM_TIMERS];                                 /* Main timer intervals */
    double termination_time;                     /* Program termination time in scaled units */
    double run_time;                               /* Time to spend on run in CPU minutes */
    int num_save_files;                           /* Number of backup save files to maintain */
    int save_file_index;                                       /* Current save file index */
    int tsf_opt;                                               /* Time-step formula option */
    double time_step_coef;                               /* Coefficient in time-step formula */
    double min_time_step;                                     /* Minimum allowed time-step */
    double max_time_step;                                     /* Maximum allowed time-step */
    BOOLEAN self_grav;                               /* TRUE to include interparticle gravity */
    double g_factor_sq;                            /* Enhancement to z oscillation frequency */
    double cp_fac_sq;                                 /* Adjustment for finite sizes in tsf */
    REST_COEF_T rest_coef;                                  /* Restitution coefficient data */
    BOOLEAN no_slide;                                       /* TRUE to inhibit sliding phase */
    BOOLEAN conserve_total_energy;                   /* TRUE to make GPE constant after hit */
    BOOLEAN include_drag;                                       /* TRUE to include gas drag */
    DRAG_COEF_T drag_coef;                                                /* Gas drag data */
    BOOLEAN allow_mergers;                                 /* TRUE to allow particle mergers */
    BOOLEAN use_tree;                                                   /* TRUE to use tree */
    int num_to_track;                                  /* Number of particles for tracking */
    int track_list[MAX_NUM_TO_TRACK];                                  /* List of particles */
    COLOR_T track_colors[MAX_NUM_TO_TRACK];                               /* List of colors */
    char stats_filename[MAX_FILENAME_LEN];                           /* Name for stats file */
    char dat_basename[MAX_FILENAME_LEN - MAX_NUM_FILENUM_DIGITS - 4];
    int dat_number;                                /* Current data file no. (base def'd above) */
    char nlv_filename[MAX_FILENAME_LEN];                          /* Name for NLV data file */
} RUN_PAR_T;

/* Definition of evolving parameters structure type (one only, viz. EvolPar) */

typedef struct {
    double vel_disp;                                                  /* Velocity dispersion */
    double mean_mass;                                                           /* Mean mass */
    double median_mass;                                                       /* Median mass */
    double mean_radius;                                                       /* Mean radius */
    double mean_roche_radius;                                           /* Mean Roche radius */
    double min_rad_vel;                                               /* Sliding phase limit */
    double cp_zone_sq;                          /* Max dist squared for closest-particle check */
    double self_grav_r2;                         /* Max dist squared for node self-grav check */
    double total_cpu;                                                /* Total CPU used so far */
} EVOL_PAR_T;

/* Definition of tree parameters structure type (one only, viz. TreePar) */

typedef struct {
    double tree_size;                                                   /* Current tree size */
    double half_tree_size;                                              /* Half width of tree */
    double expansion;                                /* Expansion factor for boundary crossing */
    double theta_sq;                                        /* Square of maximum opening angle */
    BOOLEAN use_quad;                                  /* TRUE to include quadrupole moment */
    BOOLEAN use_move;                                           /* TRUE to use "minimum" repair */
    BOOLEAN use_high_order;                              /* TRUE for high-order leaf prediction */
    BOOLEAN pred_mono;                                              /* TRUE to predict monopole */
    BOOLEAN pred_quad;                                            /* TRUE to predict quadrupole */
    BOOLEAN check_update_times;                       /* TRUE to check mono/quad update times */
    double mtsc;                                              /* Monopole time-step coefficient */
    double qtsc;                                            /* Quadrupole time-step coefficient */
    int num_excluded;                                /* Number of particles to exclude from tree */
    int exclude_list[MAX_NUM_TO_EXCLUDE];                             /* List of excluded particles */
```

```
} TREE_PAR_T;

/* Definition of movie parameters structure type (one only, viz. MoviePar) */

typedef struct {
    char basename[MAX_FILENAME_LEN - MAX_NUM_FILENUM_DIGITS - 4];
    int frame_number;                         /* Current movie frame no. (base above) */
    int frame_size;                                             /* Frame size in pixels */
    double view_size;                                      /* View size in scaled units */
    double half_view_size;                                         /* Half width of view */
    double view_centre[NUM_BOX_DIM];                               /* Centre of view (2D) */
    BOOLEAN draw_tree;                                   /* TRUE to draw tree structure */
    SHAPE_T particle_shape;                               /* Particle shape for drawing */
    double radius_mag;                                     /* Particle size magnification */
    double distance;                                                  /* Viewing distance */
    double z_mag;                                          /* Exaggeration of z dispersion */
    BOOLEAN hide_blocked_objects;                          /* TRUE to hide blocked objects */
    BOOLEAN plot_vel;                                     /* TRUE to plot velocity vectors */
    COLOR_T dflt_color;                                         /* Default particle color */
} MOVIE_PAR_T;

/* Definition of debug/check parameters struct type (one only, viz. DebugPar) */

typedef struct {
    BOOLEAN check_tree;                                     /* TRUE to perform tree checks */
    BOOLEAN check_multipoles;                                  /* TRUE to check multipoles */
    BOOLEAN check_force;                                           /* TRUE to check forces */
    int num_force_checks;                                /* Current number of force checks */
    double avg_force;                                             /* Current average force */
    double max_force;                                             /* Current maximum force */
    double total_err;                           /* Total error (div. by #/checks for avg.) */
    double max_err;                                              /* Maximum recorded error */
    double com_pos[NUM_PHYS_DIM];                         /* Position of system centre of mass */
    double com_vel[NUM_PHYS_DIM];                         /* Velocity of system centre of mass */
    double tzam;                                             /* Total z angular momentum */
    double tzam_adj;                                         /* Current adjustment to tzam */
    double tzam_rms_err;                                     /* Current rms error in tzam */
    double total_energy;                                          /* Total energy of system */
    double collision_dke;                                     /* KE loss due to collisions */
    double total_energy_adj;                          /* Current adjustment to total energy */
    double total_energy_rms_err;                        /* Current rms error in total energy */
    int num_calls;                                        /* Number of tzam/TE check calls */
} DEBUG_PAR_T;

/* Defintion of closest particle structure type (used in DATA_T below) */

typedef struct {
    int index;                                /* Index of closest particle (or CP_UNDEF) */
    int box;                                   /* Index of box occupied by closest particle */
    double radius;                                           /* Radius of closest particle */
    double rel_pos_sq;                                               /* Square of separation */
    double pos[NUM_PHYS_DIM];                            /* Low order position, ghost corrected */
    double vel[NUM_PHYS_DIM];                            /* Low order velocity, ghost corrected */
} CP_T;

/* Definition of data structure type (one per particle, c.f. Data[]) */

typedef struct {
    double mass;                                                           /* Particle mass */
    double radius;                                                       /* Particle radius */
    double radius_sq;                                              /* Square of radius */
    double inertia;                                                /* Moment of inertia */
    double drag_fac;                                               /* Gas drag coefficient */
    double pos0[NUM_PHYS_DIM];                                     /* Position at start of step */
    double pos[NUM_PHYS_DIM];                                      /* Current particle position */
    int pos_status;                                                /* NO_, UN_, LO_, or HI_PRED */
    double vel0[NUM_PHYS_DIM];                                     /* Velocity at start of step */
    double vel[NUM_PHYS_DIM];                                      /* Current particle velocity */
    int vel_status;                                                /* NO_, UN_, LO_, or HI_PRED */
    double f[NUM_PHYS_DIM];                                        /* Force (acceleration) on particle */
    double f_dot[NUM_PHYS_DIM];                                    /* First derivative */
    double d1[NUM_PHYS_DIM];                                       /* First divided difference */
    double d2[NUM_PHYS_DIM];                                       /* Second divided difference */
    double d3[NUM_PHYS_DIM];                                       /* Third divided difference */
    double spin[NUM_PHYS_DIM];                                     /* Spin vector of particle */
    double t0, t1, t2, t3;                                         /* Last four update times */
    double time_step;                                              /* Current update time-step */
    BOOLEAN in_tree;                                               /* TRUE if particle is in tree */
    NODE_T *node;                                                  /* Pointer to particle's node */
    int node_index;                                                /* Leaf position in node */
    int orig_index;                                                /* Initial particle index */
    CP_T cp;                                                       /* Closest particle data */
    int last_collider;                                             /* Index of last collider */
    int num_collisions;                                            /* No. collisions with last collider */
    BOOLEAN monitor;                                               /* Flag for debug monitoring */
    COLOR_T color;                                                 /* Color for movie tracking */
} DATA_T;

/* Definition of clock structure type (one only, viz. Clock) */

typedef struct {
    double time;                                                   /* Current time */
    double tsl_time;                                               /* Update time for time-step list */
    double timer[NUM_TIMERS];                                      /* Various timers */
} CLOCK_T;

/* Definition of time-step list structure type (one only, viz. Tsl) */

typedef struct {
    int num_on_list;                                               /* Number of particles on tsl */
    int index;                                                     /* Current position in list */
    int list[MAX_NUM_ON_TSL];                                      /* List of particles */
    double times[MAX_NUM_ON_TSL];                                  /* List of particle update times */
    double update_interval;                                        /* Current update interval */
    double stab, stab1, stab2;                                     /* Update interval stabilizers */
    BOOLEAN short_step;                                            /* Used with stabilization procedure */
} TSL_T;

/* Definition of cell union used in node_s struct (see below) */

typedef union {
    LEAF_T leaf;                                                   /* Leaf (particle) index... */
    BRANCH_T *branch;                                              /* ...or branch (node) pointer */
} CHILD_T;

/*
 * Structure template for tree nodes (one per node, starting at Root).
 * (Template used to avoid self-reference conflict).
 *
 */
```

```
struct node_s {
    int tree_index;                                  /* Node position in tree (index) */
    NODE_T *parent;                                           /* Pointer to parent node */
    int node_index;                                  /* Node position in cell (index) */
    double size;                                             /* Physical size of node */
    double half_size;                                                     /* Half size */
    double max_ext;                                        /* Maximum node extension */
    double max_size;                              /* Effective (max) size for tree force */
    double max_size_sq;                                     /* Square of effective size */
    double centre[NUM_TREE_DIM];               /* Geometric centre of node in system */
    CELL_T child_type[MAX_NUM_CHILDREN];                                /* Child types */
    CHILD_T child[MAX_NUM_CHILDREN];                             /* Child data (union) */
    int num_leaves;                                     /* Cumulative number of leaves */
    double mass;                         /* Total (cumulative) mass (mono moment) */
    double pos0[NUM_PHYS_DIM];                /* Centre-of-mass position at last update */
    double pos[NUM_PHYS_DIM];                        /* Current centre-of-mass position */
    double vel[NUM_PHYS_DIM];                        /* Current centre-of-mass velocity */
    double f[NUM_PHYS_DIM];                          /* Acceleration of centre of mass */
    double f_dot[NUM_PHYS_DIM];                                    /* First derivative */
    double mt0;                                             /* Last monopole update time */
    double mts;                                                 /* Monopole time-step */
    double q_mom0[NUM_QUAD_ELEM];               /* Quadrupole moment at last update */
    double q_mom[NUM_QUAD_ELEM];                          /* Current quadrupole moment */
    double q_dot[NUM_QUAD_ELEM];                  /* First derivative (at last update) */
    double q_2dot[NUM_QUAD_ELEM];                                 /* Second derivative */
    double q_3dot[NUM_QUAD_ELEM];                                  /* Third derivative */
    double qt0;                                          /* Last quadrupole update time */
    double qts;                                                /* Quadrupole time-step */
    BOOLEAN extended;                                         /* TRUE if extended node */
    BOOLEAN packed;                                             /* TRUE if packed node */
};

/* Global variable referencing declarations (also see extern.c) */

extern FILE *Logfile;                                              /* Pointer to log file */
extern char SaveFilename[MAX_FILENAME_LEN];                              /* Save file name */
extern BOOLEAN BackupFiles;                              /* TRUE to backup existing files */
extern int NumParticles;                                          /* Number of particles */
extern int NumBoxes;                              /* Number of boxes (currently 0, 1, or 9) */
extern RUN_PAR_T RunPar;                                                /* Run parameters */
extern EVOL_PAR_T EvolPar;                                         /* Evolving parameters */
extern TREE_PAR_T TreePar;                                             /* Tree parameters */
extern MOVIE_PAR_T MoviePar;                                          /* Movie parameters */
extern DEBUG_PAR_T DebugPar;                                          /* Debug parameters */
extern CLOCK_T Clock;                                                  /* Clock structure */
extern int Counter[NUM_COUNTERS];                                             /* Counters */
extern DATA_T *Data[MAX_NUM_PARTICLES];                                  /* Particle data */
extern TSL_T Tsl;                                                       /* Time-step list */
extern NODE_T *Root;                                      /* Pointer to root node of tree */
extern int ChildIndexOffset[NUM_TREE_DIM];                         /* Child indices/pos'ns */
extern int ChildCoordOffset[NUM_TREE_DIM][MAX_NUM_CHILDREN];
extern char Workspace[WORKSPACE_SIZE];                          /* Work space for strings */
extern char ErrorStr[WORKSPACE_SIZE];                    /* Work space for error messages */

/* Handy fractions */

extern double OneThird, TwoThirds, FourThirds, OneSixth, OneNinth, OneTwelfth;

/* Non-ANSI C may not declare these... */
```

```
#ifndef _STDIO_H
# ifndef ALPHA

extern void setbuf();                         /* Note gcc free() is defined as (void) */

extern int fclose(), fflush(), fprintf(), fread(), fscanf(), fwrite(),
        gethostname(), getopt(), getpid(), getrusage(), gettimeofday(),
        malloc_debug(), malloc_verify(), perror(), printf(), rename(),
        system(), unlink();

# endif
#endif

/* Global function referencing declarations for box_tree code */

extern void
    UpdateBoxPos(),                                         /* in bndry_cond.c... */
    Bounce(),                                               /* in bounce.c... */
    GetAngMom(),
    CheckTree(),                                            /* in check.c... */
    CheckMultipolePrediction(),
    CheckForce(),
    Draw(),                                                 /* in draw.c... */
    CalcTreeForce(),                                        /* in force.c... */
    CalcDirectForce(),
    AddGhostForce(),
    TestTreeForce(),
    SetInitCond(),                                          /* in init_cond.c... */
    InitLoOrderPoly(),                                      /* in integrate.c... */
    InitHiOrderPoly(),
    InitTsl(),
    Integrate(),
    MakeTree(),                                             /* in make_tree.c... */
    PlaceInTree(),
    InitCp(),                                               /* in misc.c... */
    CheckForCp1(),
    CheckForCp2(),
    CheckForCp3(),
    PredictPosAndVelHi(),
    PredictPosAndVelHiAll(),
    PredictPosAndQMomAll(),
    TimeStamp(),
    SaveRestartData(),
    ReadRestartData(),
    Error(),
    Terminate(),
    LongOutput(),                                           /* in output.c... */
    OpenStatsFile(),
    OutputStats(),
    CalcEvolPar(),
    OutputDat(),
    MakeMovieFrame(),
    OutputNlvData(),
    GetParams(),                                            /* in params.c... */
    DisplayParams(),
    Locate(),                                               /* in recipes.c... */
    Sort(),
    Sort2(),
    MoveInTree(),                                           /* in repair_tree.c... */
    RemoveFromTree(),
    DeallocTree(),                                   /* in tree_util.c... */
    GetOffspring(),
    CalcTreeMoments(),                               /* in update_tree.c... */
    UpdateBranchMoments(),
    UpdateMonopole(),
    UpdateQuadrupole();

extern int
    ApplyBndryCond(),                                /* in bndry_cond.c... */
    GetIndex(),                                      /* in make_tree.c... */
    CurrentIndex(),                                  /* in misc.c... */
    BackupFile(),
    TreeLevel();                                     /* in tree_util.c... */

extern NODE_T *Node();                               /* in tree_util.c... */

extern char
    *GetDate(),                                      /* in misc.c... */
    *GetHost(),
    *Boolean(),
    *MakeFilename(),
    *NodeInfo();                                     /* in tree_util.c... */

extern double
    InitMassFunc(),                                  /* in misc.c... */
    EstMeanMass(),
    Radius(),
    Mass(),
    Density(),
    RocheRadius(),
    MomentOfInertia(),
    DragFactor(),
    Median(),
    Gpe(),
    ElapsedCpu(),
    TotalCpu(),
    Ran(),                                           /* in recipes.c... */
    Gasdev();

extern BOOLEAN
    CheckMultipoles(),                               /* in check.c... */
    NotOffspring();                                  /* in tree_util.c... */

/* box_tree.h */
```

B.1.2 params.h

This header file defines all of the compile-time parameters that a user is most likely to want to change. Since this file is included by `box_tree.h`, complete recompilation of the source is required for any changes made here to come into effect. Note that this file should not be changed between restarts. The comments are fairly extensive and should serve to adequately explain the details.

```
/*
 * params.h - DCR 91-05-01
 * ========================
 * Parameter include file for box_tree code. This file contains definitions
 * that users may want to change before compiling (such as the number of
 * tree dimensions to use). Note that SaveRestartData() records these values;
 * they must not be changed for restarts.
 *
 */

/*
 * Some default file names (c.f. read_cmd_line_args() in box_tree.c).
 * Note that these strings MUST be less than MAX_FILENAME_LEN in length.
 *
 */

#define DFLT_LOG_FILENAME "box_tree.log"                        /* (c.f. Logfile) */
#define DFLT_PAR_FILENAME "box_tree.par"                   /* (used in params.c) */
#define DFLT_SAV_FILENAME "box_tree.sav"                  /* (c.f. SaveFilename) */

/*
 * Physical (spatial), tree, and box dimensions to use for simulation.
 * RESTRICTIONS: NUM_PHYS_DIM currently MUST be 3; NUM_TREE_DIM can be
 * 2 or 3; NUM_BOX_DIM must always be 2. Note that NUM_PHYS_DIM = 3 is
 * implicitly assumed in many places.
 *
 */

#define NUM_PHYS_DIM 3                                          /* For particles */
#define NUM_TREE_DIM 2                                     /* For tree structure */
#define NUM_BOX_DIM 2                                          /* For ghost boxes */

/* Maximum number of particles (for assigning array storage) */

#define MAX_NUM_PARTICLES 10000               /* (also see MAX_NUM_ON_TSL below) */

/* Maximum number of ghost boxes (do not change) */

#define MAX_NUM_BOXES 9                   /* One "real" box plus eight "ghost" boxes */

/*
 * Some dimension-dependent quantities, hard-wired to optimize execution.
 * The formula versions of these quantities are given by:
 *     #define NUM_QUAD_ELEM (2 * NUM_PHYS_DIM - 1)
 *     #define MAX_NUM_CHILDREN POW2(NUM_TREE_DIM)
 *     #define MAX_TREE_LEVEL (31 / NUM_TREE_DIM)
 *
 */

#define NUM_QUAD_ELEM 5                   /* Number of unique elements in Q tensor */

#if (NUM_TREE_DIM == 2)
# define MAX_NUM_CHILDREN 4
# define MAX_TREE_LEVEL 15
# define CHILD_INDEX_OFFSET_ARRAY {1, 2}
# define CHILD_COORD_OFFSET_ARRAY {{-1, 1, -1, 1}, {-1, -1, 1, 1}}
#else
# define MAX_NUM_CHILDREN 8
# define MAX_TREE_LEVEL 99                          /* (should be 10, but too small) */
# define CHILD_INDEX_OFFSET_ARRAY {1, 2, 4}
# define CHILD_COORD_OFFSET_ARRAY {{-1, 1, -1, 1, -1, 1, -1, 1}, \
                                    {-1, -1, 1, 1, -1, -1, 1, 1}, \
                                    {-1, -1, -1, -1, 1, 1, 1, 1}}
#endif
```

```
/* Largest membership on N-body time-step list (see integrate.c) */

#define MAX_NUM_ON_TSL MAX_NUM_PARTICLES          /* Could be smaller in most cases */

/* Maximum number of particles allowed for simultaneous tracking */

#define MAX_NUM_TO_TRACK 10                                 /* Adjust as required */

/* Maximum number of particles allowed on tree exclusion list */

#define MAX_NUM_TO_EXCLUDE 10                               /* Adjust as required */

/* Maximum number of simultaneous save files to use */

#define MAX_NUM_SAVE_FILES 10                               /* Must not exceed 10 */

/* Maximum number of digits in output file filenumbers */

#define MAX_NUM_FILENUM_DIGITS 3                          /* Allows for 000 to 999 */

/* Maximum number of characters in a filename (should be <~ 100 chars) */

#define MAX_FILENAME_LEN 81          /* Remember to add one for NULL character at end */

/* Character appended to filenames for backups */

#define BACKUP_MARKER '%'                      /* Note this is a character, not a string */

/* Default maximum string length in bytes */

#define MAX_STR_LEN 256                                /* Don't make this any smaller */

/* Bytes allocated to workspace buffer (for string manipulation) */

#define WORKSPACE_SIZE MAX_STR_LEN              /* Don't make this any smaller either */

/* Precision used in "APPROX" macros when checking double-precision numbers */

#define PRECISION 1.0e-9                         /* This is the best value found so far... */

/* params.h */
```

B.1.3 `macros.h`

The third header file contains all of the global macros used in `box_tree`, that is, all preprocessor definitions that take one or more arguments. Simple mathematical functions are defined first, followed by macros for rough comparison of double-precision numbers. The latter half of the file contains macro functions for fast vector operations and other in-line optimizations. Most of these optimizations simply eliminate a loop variable, so that operations on each component of the argument are “unwound” and written out explicitly.

```
/*
 * macros.h - DCR 93-06-03
 * ========================
 * Useful macro definitions for box_tree.
 *
 */

/* Mathematical macro definitions */
```

```
#define POW2(n) (1 ≪ (n))                                          /* 2 to nth power  */
#define EXP10(n) (pow(10.0, (double) (n)))                         /* 10 to nth power */
#define SQ(x) ((x) * (x))                                             /* Square of x */
#define CUBE(x) ((x) * (x) * (x))                                       /* Cube of x */
#define ABS(x) ((x) < 0 ? (- (x)) : (x))                              /* Abs val of x */
#define SGN(x) ((x) == 0 ? 0 : ((x) < 0 ? (-1) : 1))                  /* Signum of x */
#define MIN(x,y) ((x) < (y) ? (x) : (y))                           /* Min of x and y */
#define MAX(x,y) ((x) > (y) ? (x) : (y))                           /* Max of x and y */

/* Macros for rough comparisons of double-precision numbers */

#define APPROX_EQ(x,y) (ABS((x) - (y)) ≤ PRECISION * MAX(ABS(x), ABS(y)))
#define APPROX_LT(x,y) ((y) - (x) > PRECISION * ABS(y))
#define APPROX_GT(x,y) ((x) - (y) > PRECISION * ABS(x))
#define APPROX_LE(x,y) (APPROX_EQ((x),(y)) || APPROX_LT((x),(y)))
#define APPROX_GE(x,y) (APPROX_EQ((x),(y)) || APPROX_GT((x),(y)))

/* Following macro returns TRUE if its argument is an empty string */

#define EMPTY_STR(str) (str[0] == '\0')

/* Definitions for box macros */

#define CENTRE 0
#define LOWER_LEFT 1
#define LOWER_CENTRE 2
#define LOWER_RIGHT 3
#define CENTRE_LEFT 4
#define CENTRE_RIGHT 5
#define UPPER_LEFT 6
#define UPPER_CENTRE 7
#define UPPER_RIGHT 8

/* Macros for locating particles in zones */

#define IN_FAST_ZONE(b) \
    ((b) == LOWER_LEFT || (b) == CENTRE_LEFT || (b) == UPPER_LEFT)

#define IN_CENTRE_ZONE(b) \
    ((b) == LOWER_CENTRE || (b) == CENTRE || (b) == UPPER_CENTRE)

#define IN_SLOW_ZONE(b) \
    ((b) == LOWER_RIGHT || (b) == CENTRE_RIGHT || (b) == UPPER_RIGHT)

#define IN_LOWER_ROW(b) \
    ((b) == LOWER_LEFT || (b) == LOWER_CENTRE || (b) == LOWER_RIGHT)

#define IN_CENTRE_ROW(b) \
    ((b) == CENTRE_LEFT || (b) == CENTRE || (b) == CENTRE_RIGHT)

#define IN_UPPER_ROW(b) \
    ((b) == UPPER_LEFT || (b) == UPPER_CENTRE || (b) == UPPER_RIGHT)

#define IN_CORNER(b) \
    ((b) == LOWER_LEFT || (b) == UPPER_LEFT || \
     (b) == LOWER_RIGHT || (b) == UPPER_RIGHT)

/* Returns TRUE if "pos" outside centre box (assumes SYS_CENTRE = (0,0)) */
```

```
#define OUTSIDE_CENTRE(pos) \
    (ABS(pos[0]) > HALF_BOX_SIZE || ABS(pos[1]) > HALF_BOX_SIZE)

/* Returns TRUE if position "pos" is inside "box" */

#define INSIDE_BOX(pos, box) \
    (ABS(pos[0] - BOX_POS[box][0]) < HALF_BOX_SIZE && \
     ABS(pos[1] - BOX_POS[box][1]) < HALF_BOX_SIZE)

/* Returns TRUE if position "pos" is outside tree (assumes origin = 0) */

#if (NUM_TREE_DIM == 2)
# define OUTSIDE_TREE(pos) \
    (ABS(pos[0]) > HALF_TREE_SIZE ||\
     ABS(pos[1]) > HALF_TREE_SIZE)
#else
# define OUTSIDE_TREE(pos) \
    (ABS(pos[0]) > HALF_TREE_SIZE ||\
     ABS(pos[1]) > HALF_TREE_SIZE ||\
     ABS(pos[2]) > HALF_TREE_SIZE)
#endif

/* Returns TRUE if position "pos" is outside "node" */

#if (NUM_TREE_DIM == 2)
# define OUTSIDE_NODE(pos,node) \
    (ABS(pos[0] - node->centre[0]) > node->half_size || \
     ABS(pos[1] - node->centre[1]) > node->half_size)
#else
# define OUTSIDE_NODE(pos,node) \
    (ABS(pos[0] - node->centre[0]) > node->half_size || \
     ABS(pos[1] - node->centre[1]) > node->half_size || \
     ABS(pos[2] - node->centre[2]) > node->half_size)
#endif

/* Optimization macros, starting with simple vector operations */

#define ZERO(v) {\
    v[0] = v[1] = v[2] = 0;\
}

#define COPY(v1, v2) {\
    v2[0] = v1[0];\
    v2[1] = v1[1];\
    v2[2] = v1[2];\
}

#define NORM(v, c) {\
    double _n = 1.0 / (c);\
    v[0] *= _n;\
    v[1] *= _n;\
    v[2] *= _n;\
}

#define ADD(v1, v2, v) {\
    v[0] = v1[0] + v2[0];\
    v[1] = v1[1] + v2[1];\
    v[2] = v1[2] + v2[2];\
}
```

```
#define SUB(v1, v2, v) {\
    v[0] = v1[0] - v2[0];\
    v[1] = v1[1] - v2[1];\
    v[2] = v1[2] - v2[2];\
}

#define CROSS(v1, v2, v) {\
    v[0] = v1[1] * v2[2] - v1[2] * v2[1];\
    v[1] = v1[2] * v2[0] - v1[0] * v2[2];\
    v[2] = v1[0] * v2[1] - v1[1] * v2[0];\
}

#define DOT(v1, v2)\
    (v1[0] * v2[0] + v1[1] * v2[1] + v1[2] * v2[2])

#define MAG(v)\
    (sqrt(DOT(v, v)))

#define CROSS_Z(v1, v2)\
    (v1[0] * v2[1] - v1[1] * v2[0])

/* Macros for quickly applying ghost correction to supplied centre position */

#define ADD_BOX_OFFSET(pos, box) {\
    pos[0] += BOX_POS[box][0];\
    pos[1] += BOX_POS[box][1];\
}

#define SUB_BOX_OFFSET(pos, box) {\
    pos[0] -= BOX_POS[box][0];\
    pos[1] -= BOX_POS[box][1];\
}

/* Macros for adding or subtracting shear to supplied velocity */

#define ADD_SHEAR(ptr) {\
    ptr→vel[1] -= (ROTATING_FRAME ? 1.5 * ptr→pos[0] : 0);\
}

#define SUB_SHEAR(ptr) {\
    ptr→vel[1] += (ROTATING_FRAME ? 1.5 * ptr→pos[0] : 0);\
}

#define ADD_BOX_SHEAR(vel, box) {\
    vel[1] += BOX_VEL[box][1];\
}

#define SUB_BOX_SHEAR(vel, box) {\
    vel[1] -= BOX_VEL[box][1];\
}

/* Macro for applying bndry adj to y component of "pos" (SYS_CENTRE[1] = 0) */

#define WRAP(pos) {\
    while (ABS((pos)[1]) > HALF_SYS_SIZE)\
        (pos)[1] -= SGN((pos)[1]) * SYS_SIZE;\
}

/* Macro for applying bndry adj to both x and y (assumes SYS_CENTRE = (0,0)) */
```

```
#define REDUCE(pos) {\
    while (ABS((pos)[0]) > HALF_SYS_SIZE)\
        (pos)[0] -= SGN((pos)[0]) * SYS_SIZE;\
    while (ABS((pos)[1]) > HALF_SYS_SIZE)\
        pos[1] -= SGN((pos)[1]) * SYS_SIZE;\
}

/*
 * Macro for predicting position of particle pointed to by "ptr" to low order
 * if position has not already been predicted this time-step. Note that
 * boundary adjustments are NOT applied.
 *
 */

#define PREDICT_POS_LO(ptr) {\
    if ((ptr)->pos_status == UN_PRED) {\
        double _dt = Clock.time - (ptr)->t0;\
        (ptr)->pos[0] = (((ptr)->f_dot[0] * _dt + (ptr)->f[0]) * _dt +\
            (ptr)->vel0[0]) * _dt + (ptr)->pos0[0];\
        (ptr)->pos[1] = (((ptr)->f_dot[1] * _dt + (ptr)->f[1]) * _dt +\
            (ptr)->vel0[1]) * _dt + (ptr)->pos0[1];\
        (ptr)->pos[2] = (((ptr)->f_dot[2] * _dt + (ptr)->f[2]) * _dt +\
            (ptr)->vel0[2]) * _dt + (ptr)->pos0[2];\
        (ptr)->pos_status = LO_PRED;\
    }\
}

/* Ditto for velocity */

#define PREDICT_VEL_LO(ptr) {\
    if ((ptr)->vel_status == UN_PRED) {\
        double _dt = Clock.time - (ptr)->t0;\
        (ptr)->vel[0] = (3 * (ptr)->f_dot[0] * _dt +\
            2 * (ptr)->f[0]) * _dt + (ptr)->vel0[0];\
        (ptr)->vel[1] = (3 * (ptr)->f_dot[1] * _dt +\
            2 * (ptr)->f[1]) * _dt + (ptr)->vel0[1];\
        (ptr)->vel[2] = (3 * (ptr)->f_dot[2] * _dt +\
            2 * (ptr)->f[2]) * _dt + (ptr)->vel0[2];\
        (ptr)->vel_status = LO_PRED;\
    }\
}

/* Macro for predicting c-o-m position of "node" */

#define PREDICT_COM_POS(node) {\
    double _dt = Clock.time - (node)->mt0;\
    (node)->pos[0] = (((node)->f_dot[0] * _dt + (node)->f[0]) * _dt +\
        (node)->vel[0]) * _dt + (node)->pos0[0];\
    (node)->pos[1] = (((node)->f_dot[1] * _dt + (node)->f[1]) * _dt +\
        (node)->vel[1]) * _dt + (node)->pos0[1];\
    (node)->pos[2] = (((node)->f_dot[2] * _dt + (node)->f[2]) * _dt +\
        (node)->vel[2]) * _dt + (node)->pos0[2];\
}

/* Ditto for quadrupole moment (assumes NUM_QUAD_ELEM = 5) */

#define PREDICT_Q_MOM(node) {\
    double _dt = Clock.time - (node)->qt0;\
    (node)->q_mom[0] = (((node)->q_3dot[0] * _dt + (node)->q_2dot[0]) * _dt +\
        (node)->q_dot[0]) * _dt + (node)->q_mom0[0];\
    (node)→q_mom[1] = (((node)→q_3dot[1] * _dt + (node)→q_2dot[1]) * _dt +\
        (node)→q_dot[1]) * _dt + (node)→q_mom0[1];\
    (node)→q_mom[2] = (((node)→q_3dot[2] * _dt + (node)→q_2dot[2]) * _dt +\
        (node)→q_dot[2]) * _dt + (node)→q_mom0[2];\
    (node)→q_mom[3] = (((node)→q_3dot[3] * _dt + (node)→q_2dot[3]) * _dt +\
        (node)→q_dot[3]) * _dt + (node)→q_mom0[3];\
    (node)→q_mom[4] = (((node)→q_3dot[4] * _dt + (node)→q_2dot[4]) * _dt +\
        (node)→q_dot[4]) * _dt + (node)→q_mom0[4];\
}

/* macros.h */
```

B.1.4 `box_tree.c`

This source file is listed first because it contains `main()`, which is the entry point to all C programs. Note the convention used for the preamble at the beginning of each "`.c`" file: the header file include statements are given first, followed by any additional preprocessor definitions local to the file, all local (static) variable definitions, and all local function declarations. Global functions are listed in the comment at the very beginning. The function definitions themselves, after the preamble, are generally arranged in order of usage with the global functions first and the local functions last, although some exceptions exist.

The `box_tree.c` file, in addition to `main()`, contains arithmetic, segmentation fault, and BUS error trapping functions, as well as the routine for reading the command line arguments (`read_cmd_line_args()`), a function to check the validity of the `params.h` parameters (`check_params()`), a routine for displaying important start-up information, much of which is echoed to the log file (`display_session_info()`), and the main routine that initiates the simulation (`box_tree()`). Note that although the error trapping functions are defined locally, they are registered with system routines `ieee_handler()` and `signal()` so that they can be accessed from anywhere. The other local functions can only be invoked by `main()`. The routine to read the parameter file `GetParams()` (see `params.c` below) is also called by `main()`, as is, optionally, the routine to echo the parameters to `stdout` (`DisplayParams()`).

Note the use of the local `BOOLEAN` variable `restart` in this file, which is set to `TRUE` if the user requests a restart. This variable is checked in several places. It is also passed to `GetParams()` (along with the name of the parameter file itself).

The `box_tree()` function is responsible for initializing various global variables, including the simulation clocks and timers. The routine also calls certain global functions in order to set the initial conditions, initialize the force polynomials and time-step list, and construct the tree if applicable. It's main purpose, however, is to call `Integrate()` (cf. `integrate.c`), where control remains until the end of the simulation.

```
/*
 * box_tree.c - DCR 91-04-25
 * ========================
 * Startup functions and event handlers for box_tree.
 *
 * Global functions: main().
 *
 */

/* Include files */

#include "box_tree.h"
```

```
#ifndef SYSV
# ifndef ALPHA
# include <floatingpoint.h>                          /* For IEEE error trapping */
# endif
#endif

#include <signal.h>                               /* For bus and segv error trapping */

/* Local variables */

static char *par_filename;                    /* Name of user-supplied parameter file */
static BOOLEAN restart;                                   /* TRUE if this is a restart */

/* Local functions */

static void
    interrupt(),
    trap_division_by_zero(),
    trap_invalid_result(),
    trap_overflow(),
    trap_underflow(),
    trap_bus(),
    trap_segv(),
    read_cmd_line_args(),
    check_params(),
    display_session_info(),
    box_tree();

/* End of preamble */

int main(argc, argv)
int argc;
char *argv[];
{
    /*
     * Execution begins here.
     *
     * Return values:
     *
     *     0 - execution terminated normally
     *     1 - execution terminated as a result of an error condition
     *
     */

    /* Trap all Ctrl-C's if foreground job */

    if (signal(SIGINT, SIG_IGN) ≠ SIG_IGN)
        (void) signal(SIGINT, interrupt);

    /* Disable buffering to stdout */

    setbuf(stdout, (char *) NULL);

    /* Title */

    (void) printf("\n");
    (void) printf("box_tree ver 1.0.0 -- DCR 93-09-30\n");
    (void) printf("==================================\n");
    (void) printf("\n");
    (void) printf("See source code for license and copyright details.\n");

    /* Assume no log file initially */

    Logfile = (FILE *) NULL;

    /* Get command line arguments and/or set defaults */

    read_cmd_line_args(argc, argv);

    /*
     * Check options in params.h (done AFTER reading command line args
     * to ensure logfile open for error messages if desired).
     *
     */

    check_params();

    /*
     * Read restart file if requested (may change counters and time).
     * Otherwise abort if a save file exists (must be explicitly deleted).
     *
     */

    if (restart) {
        (void) printf("\nUser RESTART requested -- reading data...\n");
        ReadRestartData();
        (void) printf("Done.\n");
    }
    else if (BackupFiles && fopen(SaveFilename, "r"))
        Error(FATAL, "main():  Save file detected.", "");

    /* Now read parameter file, if it exists */

    GetParams(par_filename, restart);

    /* Trap FPEs, bus errors, and segmentation violations if debugging */

    if (ERROR_CHECK) {

#ifndef SYSV
# ifndef ALPHA
        (void) ieee_handler("set", "division", trap_division_by_zero);
        (void) ieee_handler("set", "invalid", trap_invalid_result);
        (void) ieee_handler("set", "overflow", trap_overflow);
        (void) ieee_handler("set", "underflow", trap_underflow);
# endif
#endif
        (void) signal(SIGBUS, trap_bus);
        (void) signal(SIGSEGV, trap_segv);
    }

    /*
     * If first run, initialize random number generator. If requested, a
     * "random" seed is constructed from the process ID number.
     *
     */

    if (!restart) {
        if (!RunPar.ran.seed)
```

```
            RunPar.ran.seed = getpid();
        RunPar.ran.seed = - RunPar.ran.seed;                    /* Neg. seed req'd to init. */
        (void) Ran();
        RunPar.ran.iset = 0;                                              /* For Gasdev() */
    }

    /* Display session info (always) */

    display_session_info();

    /* Display program parameters if desired */

    if (VERBOSE)
        DisplayParams();

    /* Call main routine (does not return) */

    box_tree();

    /* Following line should never be reached */

    return 0;
}

/* Event handlers */

static void interrupt()
{
    Error(HALT, "User INTERRUPT detected", "");
}

static void trap_division_by_zero()
{
    Error(SYS_ERR, "Floating division by zero has occured", "");
}

static void trap_invalid_result()
{
    Error(SYS_ERR, "Floating invalid result has occured", "");
}

static void trap_overflow()
{
    Error(SYS_ERR, "Floating overflow has occured", "");
}

static void trap_underflow()
{
    Error(SYS_ERR, "Floating underflow has occured", "");
}

static void trap_bus()
{
    Error(SYS_ERR, "BUS error has occured", "");
}

static void trap_segv()
{
    Error(SYS_ERR, "Segmentation violation (SEGV) has occured", "");
}
```

```
/* Remaining local functions */

#define ERROR_MSG(msg) {\
    (void) sprintf(ErrorStr, "%s%s", msg0, msg);\
    Error(FATAL, ErrorStr, usage);\
}

static void read_cmd_line_args(argc, argv)
int argc;
char *argv[];
{
    /*
     * Reads command-line arguments and overrides some defaults if
     * applicable. The getopt() function and the associated externals
     * optarg and optind are described on page 1002 of the SunOS 4.1
     * reference manual.
     *
     * Currently the following command line arguments are recognized:
     *
     *     "-b" - disables auto backup of existing output files
     *     "-l" - sets argument as name of file for logging
     *     "-p" - sets argument as name of parameter file
     *     "-r" - attempts a restart from a .sav file
     *     "-s" - sets argument as base name of save files
     *     "-x" - disables logging
     *
     */

    extern char *optarg;
    extern int optind;

    int c;
    char *msg0 = "read_cmd_line_args():  ", usage[MAX_STR_LEN], *log_filename;
    BOOLEAN use_logfile;

    /* Set defaults */

    log_filename = DFLT_LOG_FILENAME;
    par_filename = DFLT_PAR_FILENAME;

    (void) strcpy(SaveFilename, DFLT_SAV_FILENAME);

    use_logfile = TRUE;
    BackupFiles = TRUE;
    restart = FALSE;

    /* Construct usage message */

    (void) sprintf(usage, "usage:  %s %s", argv[0],
        "[-b] [-l log-file] [-p par-file] [-r] [-s sav-file] [-x]");

    /* Read options if any */

    while ((c = getopt(argc, argv, "bl:p:rs:x")) ≠ -1)
        switch (c) {
            case 'b':
                BackupFiles = FALSE;
                break;
            case 'l':
                log_filename = optarg;
                if (EMPTY_STR(log_filename))
                    ERROR_MSG("-l requires filename.");
                if (strlen(log_filename) ≥ MAX_FILENAME_LEN)
                    ERROR_MSG("-l filename too long.");
                use_logfile = TRUE;
                break;
            case 'p':
                par_filename = optarg;
                if (EMPTY_STR(par_filename))
                    ERROR_MSG("-p requires filename.");
                if (strlen(par_filename) ≥ MAX_FILENAME_LEN)
                    ERROR_MSG("-p filename too long.");
                break;
            case 'r':
                restart = TRUE;
                break;
            case 's':
                (void) strcpy(SaveFilename, optarg);
                if (EMPTY_STR(SaveFilename))
                    ERROR_MSG("-s requires filename.");
                if (strlen(SaveFilename) ≥ MAX_FILENAME_LEN - 1)
                    ERROR_MSG("-s filename too long.");
                break;
            case 'x':
                use_logfile = FALSE;
                break;
            case '?':
                ERROR_MSG("Syntax error.");
        }

    /* Final check */

    if (optind ≠ argc)
        ERROR_MSG("Missing or invalid argument.");

    /* Open log file if requested */

    if (use_logfile) {
        if (BackupFiles)
            (void) BackupFile(log_filename, BACKUP_MARKER);

        if ((Logfile = fopen(log_filename, "w")) == NULL)
            Error(IO, "read_cmd_line_args()", log_filename);
        else
            setbuf(Logfile, (char *) NULL);
    }
}

#undef ERROR_MSG

#define ERROR_MSG(msg) {\
    (void) sprintf(ErrorStr, "%s%s", msg0, msg);\
    Error(FATAL, ErrorStr, "");\
}

static void check_params()
{
    /* Checks validity of definitions in params.h */
    char *msg0 = "params.h:  ";

    if (NUM_PHYS_DIM ≠ 3)
        ERROR_MSG("This version requires NUM_PHYS_DIM = 3.");

    if (NUM_BOX_DIM ≠ 2)
        ERROR_MSG("Invalid box dimension.");

    if (NUM_TREE_DIM ≠ 2 && NUM_TREE_DIM ≠ 3)
        ERROR_MSG("Invalid tree dimension.");

    if (NUM_TREE_DIM > NUM_PHYS_DIM)                          /* (currently redundant) */
        ERROR_MSG("Tree dimension > physical dimension.");

    if (MAX_TREE_LEVEL > 31 / NUM_TREE_DIM)
        Error(WARNING1, "check_params():  Tree indices may be incorrect.", "");
}

#undef ERROR_MSG

static void display_session_info()
{
    /* Writes out some session info, starting with the date and host */

    (void) sprintf(Workspace, " %s on %s\n", GetDate(), GetHost());

    (void) printf("\nExecution begins %s", Workspace);

    if (Logfile)
        (void) fprintf(Logfile, "Session begun %s", Workspace);

    /* Message if non-standard architecture */

#ifdef SYSV
    (void) printf("\nWARNING: System V architecture...\n");
    (void) printf(" -- self-timing features disabled\n");
    (void) printf(" -- existing file backup disabled\n");
    (void) printf(" -- ieee error trapping disabled\n");
    (void) printf(" -- movie generation disabled\n");
    if (Logfile)
        (void) fprintf(Logfile, "System V architecture\n");
#endif
#ifdef ALPHA
    (void) printf("\nWARNING: Alpha architecture...\n");
    (void) printf(" -- movie generation disabled\n");
    if (Logfile)
        (void) fprintf(Logfile, "Alpha architecture\n");
#endif

    /* Display compiler options */

#if defined(OPTIMIZE)
    (void) printf("\n[Code compiled for optimization]\n");
    if (Logfile)
        (void) fprintf(Logfile, "Code compiled for optimization\n");
#elif defined(DEBUG)
    (void) printf("\n[Code compiled for debugging]\n");
    if (Logfile)
        (void) fprintf(Logfile, "Code compiled for debugging\n");
#elif defined(PROFILE)
```

```
    (void) printf("\n[Code compiled for profiling]\n");
    if (Logfile)
        (void) fprintf(Logfile, "Code compiled for profiling\n");
#endif

    /* Display compile time if possible */

#ifdef __GNUC__
    (void) printf("\n[Code compiled with gcc %s %s]\n", __DATE__, __TIME__);
    if (Logfile)
        (void) fprintf(Logfile, "Code compiled with gcc %s %s\n",
            __DATE__, __TIME__);
#endif

    /* Display physical and tree dimensions */

    (void) printf("\nPhysical dimension = %i tree dimension = %i\n",
            NUM_PHYS_DIM, NUM_TREE_DIM);

    if (Logfile)
        (void) fprintf(Logfile, "Phys dim = %i, tree dim = %i\n",
            NUM_PHYS_DIM, NUM_TREE_DIM);

    /* Message if restart */

    if (restart) {
        (void) printf("\nRESTART (restarting from %s)\n", SaveFilename);
        if (Logfile)
            (void) fprintf(Logfile, "Restart from %s\n", SaveFilename);
    }

    /* Parameter file */

    if (Logfile)
        (void) fprintf(Logfile, "Parameters read from %s\n", par_filename);

    /* Comment line */

    if (Logfile)
        (void) fprintf(Logfile, "%s\n", RunPar.comment_line);

    /* Reference frame and unit conversions */

    if (Logfile) {
        (void) fprintf(Logfile, "Reference frame = %s\n",
            (ROTATING_FRAME ? "ROTATING" : (INERTIAL_FRAME ?
            "INERTIAL" : (GALAXY_FRAME ? "GALAXY" : "UNKNOWN"))));
        (void) fprintf(Logfile, "Length units = %g m\n", RunPar.length_scale);
        (void) fprintf(Logfile, "Mass units = %g kg\n", RunPar.mass_scale);
        (void) fprintf(Logfile, "Time units = %g s\n", RunPar.time_scale);
    }

    /* Random number seed */

    if (Logfile)
        (void) fprintf(Logfile, "Random number seed = %i\n", RunPar.ran.seed);
}

static void box_tree()
{
```

```
/* Performs various initializations and calls Integrate() */

int i;

/* Following initializations not required if restart */

if (!restart) {

    /* Set system centre at origin */

    ZERO(SYS_CENTRE);

    /* Initialize "box_offset" arrays */

    for (i = 0; i < MAX_NUM_BOXES; i++) {          /* (MAX_NUM_BOXES must be 9) */
        if (IN_FAST_ZONE(i))
            BOX_X_OFFSET[i] = - BOX_SIZE;
        else if (IN_CENTRE_ZONE(i))
            BOX_X_OFFSET[i] = 0;
        else if (IN_SLOW_ZONE(i))
            BOX_X_OFFSET[i] = BOX_SIZE;
        else {
            (void) sprintf(ErrorStr, "box %i", i);
            Error(FATAL, "box_tree():  Invalid box index.", ErrorStr);
        }

        if (IN_LOWER_ROW(i))
            BOX_Y_OFFSET[i] = - BOX_SIZE;
        else if (IN_CENTRE_ROW(i))
            BOX_Y_OFFSET[i] = 0;
        else if (IN_UPPER_ROW(i))
            BOX_Y_OFFSET[i] = BOX_SIZE;
        else {
            (void) sprintf(ErrorStr, "box %i", i);
            Error(FATAL, "box_tree():  Invalid box index.", ErrorStr);
        }

        BOX_POS[i][0] = BOX_X_OFFSET[i];
        BOX_POS[i][1] = BOX_Y_OFFSET[i];

        BOX_VEL[i][0] = 0;
        BOX_VEL[i][1] = (ROTATING_FRAME ? - 1.5 * BOX_X_OFFSET[i] : 0);
    }

    /* Reset all clocks and timers */

    Clock.time = Clock.tsl_time = RunPar.init_clock_time;

    for (i = 0; i < NUM_TIMERS; i++)
        Clock.timer[i] = RunPar.init_clock_time;

    /* If init. clock time not 0, update ghost box pos'ns if applicable */

    if (ROTATING_FRAME && GHOSTS && RunPar.init_clock_time)
        UpdateBoxPos();

    /* Set or read particle initial conditions as appropriate */

    if (VERBOSE)
        (void) printf("Initializing particle data...\n");
```

```
SetInitCond();

if (VERBOSE)
    (void) printf("Done.\n");

/* Initialize all particle force polynomials and set time-steps */

if (VERBOSE)
    (void) printf("Initializing force polynomials...\n");

for (i = 0; i < NumParticles; i++)                              /* Low order first */
    InitLoOrderPoly(i);

for (i = 0; i < NumParticles; i++)                              /* Then high order */
    InitHiOrderPoly(i);

if (VERBOSE)
    (void) printf("Done.\n");

/* Initialize time-step list */

(void) printf("Initializing time-step list...\n");

if (Counter[MAX_TIME_STEPS] > MAX_NUM_ON_TSL) {
    (void) sprintf(ErrorStr, "max steps %i > max on tsl %i",
        Counter[MAX_TIME_STEPS], MAX_NUM_ON_TSL);
    Error(FATAL, "box_tree():  Too many maximum time-steps.", ErrorStr);
}

InitTsl();

if (VERBOSE)
    (void) printf("Done.\n");

/* Now construct tree if desired */

if (RunPar.use_tree) {
    if (VERBOSE)
        (void) printf("Constructing tree...\n");
    MakeTree(TREE_SIZE, TREE_CENTRE);
    if (VERBOSE)
        (void) printf("Done.\n");
}

/* Check multipoles if desired */

if (DebugPar.check_multipoles) {
    if (VERBOSE)
        (void) printf("Checking multipoles from root...\n");
    if (CheckMultipoles(Root) && VERBOSE)
        (void) printf("No errors.\n");
}

/* Backup output files if necessary */

if (BackupFiles) {
    (void) BackupFile(RunPar.stats_filename, BACKUP_MARKER);
    (void) BackupFile(RunPar.nlv_filename, BACKUP_MARKER);
}
```

```
        /* Prepare statistics file */

        if (RunPar.interval[STATS])
            OpenStatsFile();

    }  /* if not restart */
    else {
        /* Perform output and check tree before resuming integration */

        if (RunPar.interval[OUTPUT] && Clock.timer[OUTPUT] > Clock.time)
            LongOutput();

        if (DebugPar.check_tree && Clock.timer[CHECK] > Clock.time) {
            if (VERBOSE)
                (void) printf("Checking tree on restart...\n");
            CheckTree(Root);
            if (VERBOSE)
                (void) printf("No errors detected.\n");
        }
    }

    /* Output some info */

    (void) printf("\nIntegration begins at CPU time = ");
    (void) printf("%.2e min (current run)...\n", ElapsedCpu());

    /* Stamp log file if it exists */

    if (Logfile)
        TimeStamp();

    /* Begin integration */

    Integrate();

    /* All done */

    (void) printf("\nIntegration completed!\n");
    if (MONITOR && DebugPar.num_force_checks)
        (void) printf("Force error status:  avg abs err %.5e max err %.5e\n",
            DebugPar.total_err / DebugPar.num_force_checks, DebugPar.max_err);
    (void) printf("Elapsed CPU this run = %.2e min.\n", ElapsedCpu());
    (void) printf("Total CPU (all runs) = %.2e min.  (%i time-steps).\n",
        TotalCpu(), Counter[TIME_STEPS]);

    Terminate(ALL_DONE);
}

/* box_tree.c */
```

B.1.5 `bndry_cond.c`

The remaining source files are listed in alphabetical order. The `bndry_cond.c` file contains two global functions: `UpdateBoxPos()` for adjusting the y-displacement of the ghost boxes, and `ApplyBndryCond()` for applying boundary conditions to a given particle. The ghost box positions are used to determine ghost particle positions, so `UpdateBoxPos()` is called immediately after the simulation clock has been set at the beginning of each pass through the main integration loop. A ghost box boundary crossing counter is used to ensure the ghost box centres stay within one box half-width in the y-direction of the system origin. The routine takes a negligible amount of CPU so it is not particularly optimized. If ghost particles are not being included explicitly, `UpdateBoxPos()` need only be called when there is a particle boundary crossing.

The function `ApplyBndryCond()` is called to check whether a particle has crossed a box or tree boundary, and to apply the appropriate boundary conditions. The function returns `BC_NONE` if there was no boundary crossing, `BC_BOX` is there was a box crossing, or `BC_TREE` if there was a tree crossing. In the case of a box crossing, various corrections are made to any conservation variables affected by the change in particle position and velocity (cf. §4.4). The routine makes use of the local function `get_box()` to determine the index of the ghost box the central particle is entering. In the event of an allowed tree boundary crossing, the tree is expanded and completely rebuilt.

```
/*
 * bndry_cond.c - DCR 92-03-13
 * ===========================
 * Code for handling box_tree boundary conditions.
 *
 * Global functions: UpdateBoxPos(), ApplyBndryCond().
 *
 */

/* Include files */

#include "box_tree.h"

/* Local functions */

static int get_box();

/* End of preamble */

void UpdateBoxPos()
{
    /*
     * Updates y-position of ghost boxes to current time for rotating
     * frame. This routine should be called after Clock.time has been
     * updated if ghost particles are being used. (If there are NO ghost
     * particles, this function need only be called when determining which
     * ghost would have replaced the current particle in the event of a
     * central box boundary crossing). Note that this routine assumes
     * Clock.time always increases.
     *
     */

    static double last_time = 0;

    int i;
    double shear;

    /* Return if clock hasn't changed */

    if (last_time == Clock.time)
        return;

    /* Calculate shear */
    shear = 1.5 * Clock.time - Counter[GHOST_BOX_BNDRY_XINGS];

    /* Adjust for box boundary crossings */

    while (shear > 0.5) {
        ++Counter[GHOST_BOX_BNDRY_XINGS];
        --shear;
    }

    /* Apply shear */

    for (i = 0; i < NumBoxes; i++)
        BOX_POS[i][1] = BOX_Y_OFFSET[i] - shear * BOX_X_OFFSET[i];

    /* Store time */

    last_time = Clock.time;
}

#define MAX_NUM_EXPANSIONS 100                              /* For tree resizing */

int ApplyBndryCond(particle)
int particle;
{
    /*
     * Determines whether "particle" has crossed a box or tree boundary
     * (depending on boundary condition option). If so, the appropriate
     * boundary conditions are applied. In the case of periodic boundary
     * conditions in the central box, it is up to the calling function to
     * ensure the particle is correctly reinitialized.
     *
     * Return values:
     *
     *     BC_NONE - particle is within box and tree boundaries
     *     BC_BOX  - box boundary condition was applied to particle
     *     BC_TREE - tree boundary condition was applied to particle
     *
     */

    DATA_T *ptr = Data[particle];

    /* First check if box is bounded and particle outside box */

    if (RunPar.bc_opt ≠ UNBOUNDED && OUTSIDE_CENTRE(ptr→pos)) {
        int k, box;
        double old_gpe = 0;

        /* Error if no BC's allowed */

        if (RunPar.bc_opt == DISABLED) {
            (void) sprintf(ErrorStr, "particle %i (%i) x %g y %g t %g",
                particle, ptr→orig_index, ptr→pos[0], ptr→pos[1], TIME);
            Error(FATAL,
                "ApplyBndryCond():  Particle outside boundary but BCs disabled.",
                ErrorStr);
        }

        /* Update ghost boxes for case of no ghost particles (if applicable) */

        if (ROTATING_FRAME && !GHOSTS)
    UpdateBoxPos();

/* Determine index of box that particle is ENTERING */

box = get_box(ptr->pos);

/* Mandatory error check */

if (box == -1) {
    (void) sprintf(ErrorStr, "particle %i (%i) x %g y %g",
        particle, ptr->orig_index, ptr->pos[0], ptr->pos[1]);
    Error(FATAL, "ApplyBndryCond():  Destination box not found.",
        ErrorStr);
}

if (box == CENTRE) {
   (void) sprintf(ErrorStr, "particle %i (%i) x %g y %g",
        particle, ptr->orig_index, ptr->pos[0], ptr->pos[1]);
    Error(FATAL, "ApplyBndryCond():  Particle still in centre!",
        ErrorStr);
}

/* Get current GPE if required before applying boundary condition */

if (ERROR_CHECK || RunPar.conserve_total_energy)
   old_gpe = Gpe();                  /* (assumes all pos'ns predicted to low order...) */

/* Replace particle with ghost image, now in centre box */

for (k = 0; k < NUM_BOX_DIM; k++) {
   ptr->pos[k] -= BOX_POS[box][k];
   ptr->vel[k] -= BOX_VEL[box][k];
}

/*
 * Adjust centre-of-mass position and velocity if applicable.
 * Note that this correction applies only at the time of boundary
 * crossing: until the system becomes symmetric again there will
 * be a systematic motion of the centre-of-mass position.
 *
 */

if (ERROR_CHECK) {
    double r = ptr->mass / RunPar.total_mass;

    for (k = 0; k < NUM_BOX_DIM; k++) {        /* (assume SYS_CENTRE = (0,0) */
        DebugPar.com_pos[k] -= r * BOX_POS[box][k];
        DebugPar.com_vel[k] -= r * BOX_VEL[box][k];
    }
}

/* Correct total z angular momentum if applicable */

if (ERROR_CHECK) {
    if (ROTATING_FRAME)
        DebugPar.tzam_adj +=
            ptr->mass * (BOX_VEL[box][1] + 2 * BOX_POS[box][0]);
    else if (INERTIAL_FRAME)
        DebugPar.tzam_adj +=
            ptr->mass * CROSS_Z(BOX_POS[box], ptr->vel);
        }

        /* Adjust GPE if applicable */

        if (ERROR_CHECK || RunPar.conserve_total_energy)
            DebugPar.total_energy_adj -= Gpe() - old_gpe;

        /* Increment counters */

        ++Counter[BNDRY_XINGS];

        if (IN_FAST_ZONE(box) || IN_SLOW_ZONE(box))
            ++Counter[LATERAL_BNDRY_XINGS];

        return BC_BOX;
    }

    /* Otherwise check if box is unbounded but particle outside tree */

    if (RunPar.bc_opt == UNBOUNDED && ptr→in_tree && OUTSIDE_TREE(ptr→pos)) {
        int i, n;

        /* Error if not allowed to expand tree */

        if (Root == NULL || TreePar.expansion ≤ 1) {
            (void) sprintf(ErrorStr, "particle %i (%i) x %g y %g z %g t %g",
                particle, ptr→orig_index, ptr→pos[0], ptr→pos[1],
                ptr→pos[2], TIME);
            Error(FATAL, "ApplyBndryCond():  Particle outside tree.", ErrorStr);
        }

        /* Get new tree size, warning if it's taking too long... */

        n = 0;

        while (OUTSIDE_TREE(ptr→pos)) {
            if (++n == MAX_NUM_EXPANSIONS)
                Error(WARNING1, "ApplyBndryCond():  Infinite loop?", "");
            TREE_SIZE ∗= TreePar.expansion;
            HALF_TREE_SIZE ∗= TreePar.expansion;
        }

        /* Reconstruct tree */

        DeallocTree(Root);
        PredictPosAndVelHiAll();
        if (TreePar.pred_mono) {
            for (i = 0; i < NumParticles; i++)
                InitLoOrderPoly(i);
            for (i = 0; i < NumParticles; i++)
                InitHiOrderPoly(i);
            InitTsl();
        }
        MakeTree(TREE_SIZE, TREE_CENTRE);
        (void) sprintf(ErrorStr, "%i expansion(s), time %g, new size %e", n,
            TIME, TREE_SIZE);
        Error(WARNING1, "ApplyBndryCond():  Tree resized.", ErrorStr);

        return BC_TREE;
    }
    return BC_NONE;
}

#undef MAX_NUM_EXPANSIONS

static int get_box(pos)
double *pos;
{
    /* Returns index of box containing "pos" (-1 if not found) */

    int box = -1;
    double dx = pos[0], dy = pos[1];                    /* (assume SYS_CENTRE = (0,0)) */

    /* Error check */

    if (ERROR_CHECK && (dx > BOX_SIZE || dy > BOX_SIZE)) {
        (void) sprintf(ErrorStr, "dx %g dy %g", dx, dy);
        Error(FATAL, "get_box():  Position too far out.", ErrorStr);
    }

    /* Determine box index */

    if (dx < - HALF_BOX_SIZE) {
        if (INSIDE_BOX(pos, LOWER_LEFT))
            box = LOWER_LEFT;
        else if (INSIDE_BOX(pos, CENTRE_LEFT))
            box = CENTRE_LEFT;
        else if (INSIDE_BOX(pos, UPPER_LEFT))
            box = UPPER_LEFT;
    }
    else if (dx > HALF_BOX_SIZE) {
        if (INSIDE_BOX(pos, LOWER_RIGHT))
            box = LOWER_RIGHT;
        else if (INSIDE_BOX(pos, CENTRE_RIGHT))
            box = CENTRE_RIGHT;
        else if (INSIDE_BOX(pos, UPPER_RIGHT))
            box = UPPER_RIGHT;
    }
    else {
        if (dy < - HALF_BOX_SIZE)
            box = LOWER_CENTRE;
        else if (dy > HALF_BOX_SIZE)
            box = UPPER_CENTRE;
        else
            box = CENTRE;
    }

    return box;
}

/* bndry_cond.c */
```

B.1.6 bounce.c

This file contains code for determining new particle velocities once a two-body collision has been established and for checking the consistency of the calculations. The global routine `Bounce()` is called only by `collision()` (cf. `integrate.c`). The routine takes as arguments pointers to the colliders' data structures, the initial positions, velocities, and spins, and addresses for storing the final velocities and spins. Many local variables are defined to simplify the calculation process and much use is made of the vector macros defined in `params.h`. The formulae used for calculating the final velocities and spins are given by equations (3.14)–(3.17). An optional calculation of the change in kinetic energy is also included [equation (3.18)]. In addition, the local function `check_bounce()` may be called to verify the conservation properties of the collision equations. If velocity-dependent coefficients of restitution are in use, the local routine `bhl_formula()` is called to determine the appropriate values. The `check_bounce()` routine makes use of `GetAngMom()` and `get_lin_mom()` to calculate the angular and linear momentum, respectively, of a two-body system. The former routine is global because it is also called from the routine `merge()` in `integrate.c`.

```
/*
 * bounce.c - DCR 92-02-03
 * ========================
 * Code for calculating two-body collision dynamics.
 *
 * Global functions: Bounce(), GetAngMom().
 *
 */

/* Include files */

#include "box_tree.h"

/* Local functions */

static double bhl_formula();
static void check_bounce(), get_lin_mom();

/* End of preamble */

#define MAX_SEP_ERR 0.01                    /* Warning if colliders exceed this fract. penet. */

void Bounce(ptr1, ptr2, pos1, pos2, v1_i, v2_i, w1_i, w2_i, v1_f, v2_f,
            w1_f, w2_f)
DATA_T *ptr1, *ptr2;
double *pos1, *pos2, *v1_i, *v2_i, *w1_i, *w2_i, *v1_f, *v2_f, *w1_f, *w2_f;
{
    /*
     * Calculates linear velocities v1_f & v2_f and angular velocities
     * w1_f & w2_f following collision between particles ptr1 & ptr2 at
     * positions pos1 & pos2, with initial linear velocities v1_i & v2_i
     * and initial angular velocities w1_i & w2_i.
     *
     * NOTE: Memory economy has been sacrificed in favour of readability
     * in this routine.
     *
     */

    int k;

    /* Working variables... */

    double
        r1,                                                   /* Radius of particle 1 */
        r2,                                                   /* Radius of particle 2 */
        r,                                        /* Particle separation (sum of radii) */
    m1,                                              /* Mass of particle 1 */
    m2,                                              /* Mass of particle 2 */
    m,                                           /* Total mass (m1 + m2) */
    i1,                                   /* Moment of inertia of particle 1 */
    i2,                                   /* Moment of inertia of particle 2 */
    alpha,                                     /* Useful quantity in formulae */
    mu,                                                    /* Reduced mass */
    beta,                                      /* Reciprocal of 1 + alpha mu */
    n_hat[NUM_PHYS_DIM],           /* Unit vector in normal (radial) direction */
    v[NUM_PHYS_DIM],                         /* Initial relative linear velocity */
    s1[NUM_PHYS_DIM],                     /* Initial spin velocity of 1st particle */
    s2[NUM_PHYS_DIM],                     /* Initial spin velocity of 2nd particle */
    s[NUM_PHYS_DIM],                           /* Initial relative spin velocity */
    u[NUM_PHYS_DIM],                           /* Initial relative total velocity */
    u_n[NUM_PHYS_DIM],                             /* Initial n component of u */
    u_t[NUM_PHYS_DIM],                             /* Initial t component of u */
    eps_n,                            /* Radial component of coef. of restitution */
    eps_t,                       /* Transverse component of coef. of restitution */
    nxu[NUM_PHYS_DIM],                           /* Cross product of n_hat and u */
    p[NUM_PHYS_DIM],                                   /* Useful working vector */
    q[NUM_PHYS_DIM],                                                 /* Ditto */
    w[NUM_PHYS_DIM],               /* Radius-weighted sum of angular velocities */
    dum_dbl;                                            /* Useful dummy variable */

/* Load working variables */

r1 = ptr1→radius;
r2 = ptr2→radius;
r = r1 + r2;
m1 = ptr1→mass;
m2 = ptr2→mass;
m = m1 + m2;
i1 = ptr1→inertia;
i2 = ptr2→inertia;
alpha = ptr1→radius_sq / i1 + ptr2→radius_sq / i2;
mu = m1 * m2 / m;
beta = 1 / (1 + alpha * mu);

/* Calculate normal vector at impact site */

SUB(pos2, pos1, n_hat);

dum_dbl = MAG(n_hat);

if (ERROR_CHECK) {
    if (dum_dbl == 0)
        Error(FATAL, "Bounce():  Particles coincide!", "");
    if (!APPROX_EQ(r, dum_dbl)) {
        (void) sprintf(ErrorStr, "dist = %e r1 + r2 = %e", dum_dbl, r);
        Error(WARNING2, "Bounce():  Distance discrepancy.", ErrorStr);
    }
}

NORM(n_hat, dum_dbl);

/* Calculate relative linear velocity */

SUB(v2_i, v1_i, v);

/* Calculate relative spin velocity */
```

```
CROSS(w1_i, n_hat, s1);
CROSS(w2_i, n_hat, s2);

for (k = 0; k < NUM_PHYS_DIM; k++)
    s[k] = - (r1 * s1[k] + r2 * s2[k]);

/* Hence calculate total relative velocity */

ADD(v, s, u);

/* Get n and t components of total relative velocity */

dum_dbl = DOT(u, n_hat);

for (k = 0; k < NUM_PHYS_DIM; k++)
    u_n[k] = dum_dbl * n_hat[k];

SUB(u, u_n, u_t);

/* Get cross product of n_hat and u */

CROSS(n_hat, u, nxu);

/* Set coefficient of restitution components, checking for sliding cond. */

if ((dum_dbl = MAG(u_n)) < EvolPar.min_rad_vel)
    eps_n = 1;
else if ((eps_n = RunPar.rest_coef.radial) == BHL_FLAG)
    eps_n = bhl_formula(dum_dbl);

if ((eps_t = RunPar.rest_coef.transverse) == BHL_FLAG)
    eps_t = bhl_formula(MAG(u_t));

/* Construct new vectors p, q, and w to simplify things later on... */

for (k = 0; k < NUM_PHYS_DIM; k++) {
    p[k] = (1 + eps_n) * u_n[k] + beta * (1 - eps_t) * u_t[k];
    q[k] = mu * beta * (1 - eps_t) * nxu[k];
    w[k] = r1 * w1_i[k] + r2 * w2_i[k];
}

/* Print out info if desired */

if (ERROR_CHECK && VERY_VERBOSE)
    (void) printf(" Coef of rest:  n %.2f t %.2f\n", eps_n, eps_t);

/* Obtain final velocities and spins */

for (k = 0; k < NUM_PHYS_DIM; k++) {
    v1_f[k] = v1_i[k] + (m2 / m) * p[k];
    v2_f[k] = v2_i[k] - (m1 / m) * p[k];
    w1_f[k] = w1_i[k] + (r1 / i1) * q[k];
    w2_f[k] = w2_i[k] + (r2 / i2) * q[k];
}

/* Adjust total energy of system to reflect collisional loss */

if (ERROR_CHECK || RunPar.conserve_total_energy) {
    double dke;
```

```
        dke = 0.5 * mu * DOT(p, p) + 0.5 * alpha * DOT(q, q) -
            mu * DOT(v, p) + DOT(w, q);                      /* Analytic expression */

        DebugPar.collision_dke += dke;
        DebugPar.total_energy_adj -= dke;
    }

    /* Check results if desired */

    if (MONITOR)
        check_bounce(ptr1, ptr2, pos1, pos2, v1_i, v2_i, w1_i, w2_i,
                v1_f, v2_f, w1_f, w2_f);
}

#undef MAX_SEP_ERR

static double bhl_formula(v)
double v;
{
    /*
     * Returns coefficient of restitution corresponding to relative
     * velocity "v" using "Bridges-Hatzes-Lin"-type formula (specific
     * to ice balls).
     *
     */

    if (v == 0)
        return 1.0;

    /* Factor of 100 converts from mks to cgs */

    return MIN(0.34 * pow(v * RunPar.velocity_scale * 100, - 0.234), 1.0);
}

#define ACCURACY 5.0e-04                    /* Desired conservation accuracy before warning */

static void check_bounce(ptr1, ptr2, pos1, pos2, v1_i, v2_i, w1_i, w2_i,
                        v1_f, v2_f, w1_f, w2_f)
DATA_T *ptr1, *ptr2;
double *pos1, *pos2, *v1_i, *v2_i, *w1_i, *w2_i, *v1_f, *v2_f, *w1_f, *w2_f;
{
    /*
     * Checks for linear and angular momentum conservation following
     * collision and outputs some quantities if desired.
     *
     */

    double m1, m2, i1, i2, pi[NUM_PHYS_DIM], pi_mag, pf[NUM_PHYS_DIM],
        dp[NUM_PHYS_DIM], dp_mag, li[NUM_PHYS_DIM], li_mag,
        lf[NUM_PHYS_DIM], dl[NUM_PHYS_DIM], dl_mag, ke_i, ke_f,
        v1, v2, w1, w2;

    /* Get standard quantities */

    m1 = ptr1→mass;
    m2 = ptr2→mass;
    i1 = ptr1→inertia;
    i2 = ptr2→inertia;
```

```
/* Linear momentum check */

get_lin_mom(m1, v1_i, m2, v2_i, pi);
get_lin_mom(m1, v1_f, m2, v2_f, pf);

SUB(pf, pi, dp);

pi_mag = MAG(pi);

if (APPROX_GT(pi_mag, 0))
    NORM(dp, pi_mag);

dp_mag = MAG(dp);

if (dp_mag > ACCURACY)
    Error(WARNING2, "check_bounce(): Poor lin. mom. conservation.", "");

if (VERBOSE) {
    (void) printf(" Change in linear momentum / initial mag:\n");
    (void) printf(" dx = %+9.2e dy = %+9.2e dz = %+9.2e",
            dp[0], dp[1], dp[2]);
    (void) printf(" (mag %9.3e)\n", dp_mag);
}

/* Angular momentum check */

GetAngMom(m1, pos1, v1_i, i1, w1_i, m2, pos2, v2_i, i2, w2_i, li);
GetAngMom(m1, pos1, v1_f, i1, w1_f, m2, pos2, v2_f, i2, w2_f, lf);

SUB(lf, li, dl);

li_mag = MAG(li);

if (APPROX_GT(li_mag, 0))
    NORM(dl, li_mag);

dl_mag = MAG(dl);

if (dl_mag > ACCURACY)
    Error(WARNING2, "check_bounce(): Poor ang. mom. conservation.", "");

if (VERBOSE) {
    (void) printf(" Change in angular momentum / initial mag:\n");
    (void) printf(" dx = %+9.2e dy = %+9.2e dz = %+9.2e",
            dl[0], dl[1], dl[2]);
    (void) printf(" (mag %9.3e)\n", dl_mag);
}

/* Kinetic energy check */

v1 = MAG(v1_i);
v2 = MAG(v2_i);
w1 = MAG(w1_i);
w2 = MAG(w2_i);
ke_i = 0.5 * (m1 * SQ(v1) + m2 * SQ(v2) + i1 * SQ(w1) + i2 * SQ(w2));
v1 = MAG(v1_f);
v2 = MAG(v2_f);
w1 = MAG(w1_f);
w2 = MAG(w2_f);
ke_f = 0.5 * (m1 * SQ(v1) + m2 * SQ(v2) + i1 * SQ(w1) + i2 * SQ(w2));
```

```
    if (APPROX_GT(ke_f, ke_i))
        Error(WARNING2, "check_bounce(): K.E. gain after collision!", "");

    if (VERBOSE)
        (void) printf(" KE before %9.3e, after %9.3e, change %9.3e\n",
                ke_i, ke_f, ke_f - ke_i);
}

#undef ACCURACY

void GetAngMom(m1, r1, v1, i1, w1, m2, r2, v2, i2, w2, l)
double m1, *r1, *v1, i1, *w1, m2, *r2, *v2, i2, *w2, *l;
{
    /* Sets "l" to angular momentum of two-body system w.r.t. c-o-m */

    int k;
    double norm, rc_k, rc1[NUM_PHYS_DIM], rc2[NUM_PHYS_DIM], vc_k,
        vc1[NUM_PHYS_DIM], vc2[NUM_PHYS_DIM], rxv1[NUM_PHYS_DIM],
        rxv2[NUM_PHYS_DIM];

    norm = 1 / (m1 + m2);

    /* Calculate quantities w.r.t. c-o-m */

    for (k = 0; k < NUM_PHYS_DIM; k++) {
        rc_k = (m1 * r1[k] + m2 * r2[k]) * norm;
        rc1[k] = r1[k] - rc_k;
        rc2[k] = r2[k] - rc_k;
        vc_k = (m1 * v1[k] + m2 * v2[k]) * norm;
        vc1[k] = v1[k] - vc_k;
        vc2[k] = v2[k] - vc_k;
    }

    /* Get cross products */

    CROSS(rc1, vc1, rxv1);
    CROSS(rc2, vc2, rxv2);

    /* Calculate angular momentum */

    for (k = 0; k < NUM_PHYS_DIM; k++)
        l[k] = m1 * rxv1[k] + m2 * rxv2[k] + i1 * w1[k] + i2 * w2[k];
}

static void get_lin_mom(m1, v1, m2, v2, p)
double m1, *v1, m2, *v2, *p;
{
    /* Sets "p" to linear momentum of two-body system */

    int k;

    for (k = 0; k < NUM_PHYS_DIM; k++)
        p[k] = m1 * v1[k] + m2 * v2[k];
}

/* bounce.c */
```

B.1.7 `check.c`

The `check.c` file contains various error checking routines, currently all associated with the tree. There are four global functions and two local functions in the file. The `CheckTree()` routine performs a variety of simple tests, checking node sizes, child properties, etc. The routine is recursive, checking the tree from the top down. The local function `node_error()` is used in conjunction with `CheckTree()` to generate error messages (any errors are fatal). The `CheckMultipoles()` routine is also recursive, and is dedicated to calculating the multipole moments of the tree explicitly as a check. The time derivatives of the quadrupole are also checked. The routine assumes all particle and tree data are up to date, so it is currently only called after initial tree construction. The `CheckMultipolePrediction()` function compares the predicted multipole moments of the tree with the actual multipole moments and reports any large discrepancies. The routine itself is actually only a driver for the recursive function `check_multipole_prediction_r()` which does the bulk of the work. Finally, the `CheckForce()` routine calculates the current force on each particle using: (1) the direct method; (2) the tree method with the monopole approximation; (3) the tree with quadrupole; and (4) a direct method that allows for node boundary conditions (cf. §4.3). The forces are compared and differences are reported. This routine is not as useful as `check_force()` in `force.c`, which performs a node-by-node force comparison, but it can still be helpful.

```
/*
 * check.c - DCR 91-09-11
 * ======================
 * Miscellaneous self-check routines for box_tree code. Many of these routines
 * are based on or inspired by code developed by SLM.
 *
 * Global functions: CheckTree(), CheckMultipoles(), CheckMultipolePrediction(),
 *     CheckForce().
 *
 */

/* Include files */

#include "box_tree.h"

/* Additional definitions */

#define TOLERANCE 1.0e-06                    /* Tolerance level for multipole checks */

/* Local functions */

static void node_error(), check_mult_pred_r();

/* Local variables */

static LEAF_T offspring[MAX_NUM_PARTICLES];                             /* Mass storage */

/* End of preamble */

void CheckTree(node)
NODE_T *node;
{
    /*
     * Performs a number of tests to check tree integrity. This function
     * is recursive, so it must be provided with a tree node to start,
     * usually Root. The function node_error() is used for more efficient
     * handling of error messages.
     *
     */

    int i, k, particle, num_leaves, index;
    DATA_T *ptr;
    CHILD_T *child;
    double mass;

    /* Check tree index */

/* due to possible integer overflow, this test is not implemented
    if (Node(node->tree_index) != node) {
        (void) sprintf(ErrorStr, "tree index = %i", node->tree_index);
        node_error((NODE_T *) NULL, "Tree index/node mismatch.", ErrorStr);
    }
*/

    /* Check parent and node index */

    if (node == Root) {
        if (node→parent ≠ NULL) {
            (void) sprintf(ErrorStr, "parent = %p", (void *) node→parent);
            node_error((NODE_T *) NULL, "Root parent not NULL", ErrorStr);
        }
        if (node→node_index ≠ -1) {
            (void) sprintf(ErrorStr, "node index = %i", node→node_index);
            node_error((NODE_T *) NULL, "Root node index not -1.", ErrorStr);
        }
    }
    else if (node→parent→child[node→node_index].branch ≠ node) {
        (void) sprintf(ErrorStr, "tree index = %i", node→tree_index);
        node_error((NODE_T *) NULL, "Node/parent mismatch.", ErrorStr);
    }

    /* Check sizes */

    if (node == Root && node→size ≠ TREE_SIZE) {
        (void) sprintf(ErrorStr, "expected %g got %g", TREE_SIZE, node→size);
        node_error((NODE_T *) NULL, "Root size incorrect.", ErrorStr);
    }
    else if (node ≠ Root && !APPROX_EQ(node→size, node→parent→half_size)) {
        (void) sprintf(ErrorStr, "expected %g got %g",
            node→parent→half_size, node→size);
        node_error(node, "Node size incorrect.", ErrorStr);
    }

    if (APPROX_LT(node→max_size, 2 * node→max_ext)) {
        (void) sprintf(ErrorStr, "%g < %g", node→max_size, 2 * node→max_ext);
        node_error(node, "Node max size smaller than 2 * max ext.", ErrorStr);
    }

    if (node→size > node→max_size) {
        (void) sprintf(ErrorStr, "node size %g > max size %g", node→size,
            node→max_size);
        node_error(node, "Node size exceeds max size.", ErrorStr);
    }

    if (!APPROX_EQ(SQ(node→max_size), node→max_size_sq)) {
        (void) sprintf(ErrorStr, "expected %g got %g", node→max_size_sq,
        SQ(node→max_size));
    node_error(node, "Max size sq inconsistency.", ErrorStr);
}

/* Check centre */

if (node == Root) {
    for (k = 0; k < NUM_TREE_DIM; k++)
        if (!APPROX_EQ(node→centre[k], TREE_CENTRE[k]))
            node_error((NODE_T *) NULL, "Root centre incorrect.", "");
}
else
    for (k = 0; k < NUM_TREE_DIM; k++)
        if (!APPROX_EQ(node→centre[k], node→parent→centre[k] +
                ChildCoordOffset[k][node→node_index] * node→half_size))
            node_error(node, "Node centre incorrect.", "");

/* Check children */

for (i = 0; i < MAX_NUM_CHILDREN; i++) {
    child = &node→child[i];
    switch (node→child_type[i]) {
        case EMPTY:
            if (child→leaf ≠ -1) {
                (void) sprintf(ErrorStr, "child %i value %i", i,
                    child→leaf);
                node_error(node, "Empty cell incorrect code.", ErrorStr);
            }
            break;
        case LEAF:
            ptr = Data[particle = child→leaf];
            if (particle < 0 || particle ≥ NumParticles) {
                (void) sprintf(ErrorStr, "child %i value %i", i, particle);
                node_error(node, "Invalid leaf index.", ErrorStr);
            }
            if (ptr→node ≠ node) {
                (void) sprintf(ErrorStr, "child %i value %i", i, particle);
                node_error(node, "Leaf/node mismatch.", ErrorStr);
            }
            if (ptr→node_index ≠ i) {
                (void) sprintf(ErrorStr, "child %i value %i", i, particle);
                node_error(node, "Leaf/node_index mismatch.", ErrorStr);
            }
            if ((index = GetIndex(particle, ptr→pos0, node)) == -1) {
                (void) sprintf(ErrorStr, "child %i value %i", i, particle);
                node_error(node, "Leaf t0 pos'n outside node.", ErrorStr);
            }
            if (index ≠ i && !node→packed) {
                (void) sprintf(ErrorStr, "child %i value %i index %i)", i,
                    particle, index);
                node_error(node, "Leaf index mismatch.", ErrorStr);
            }
           break;
        case BRANCH:
            if (child→branch == NULL) {
                (void) sprintf(ErrorStr, "child %i", i);
                node_error(node, "Child branch is NULL.", ErrorStr);
            }
            break;
        default:
            (void) sprintf(ErrorStr, "child type %i", node→child_type[i]);
            node_error(node, "Unknown child type.", ErrorStr);
    }  /* switch */
}  /* for */

/* Check number of leaves and node mass against children */

for (mass = 0.0, num_leaves = i = 0; i < MAX_NUM_CHILDREN; i++) {
    child = &node→child[i];
    if (node→child_type[i] == LEAF) {
        ++num_leaves;
        mass += Data[child→leaf]→mass;
    }
    else if (node→child_type[i] == BRANCH) {
        num_leaves += child→branch→num_leaves;
        mass += child→branch→mass;
    }
}

if (num_leaves ≠ node→num_leaves) {
    (void) sprintf(ErrorStr, "expected %i got %i", num_leaves,
        node→num_leaves);
    node_error(node, "Node leaf count incorrect.", ErrorStr);
}

if (!APPROX_EQ(mass, node→mass)) {
    (void) sprintf(ErrorStr, "expected %g got %g", mass, node→mass);
    node_error(node, "Node mass incorrect.", ErrorStr);
}

/* Also check update times and time-steps */

if (TreePar.check_update_times) {
    if (node→mt0 < 0 || APPROX_GT(node→mt0, Clock.time)) {
        (void) sprintf(ErrorStr, "mt0 = %g", node→mt0);
        node_error(node, "Invalid monopole t0 time.", ErrorStr);
    }

    if (node→mts ≤ 0) {
        (void) sprintf(ErrorStr, "mts = %g", node→mts);
        node_error(node, "Invalid monopole time-step.", ErrorStr);
    }

    if (TreePar.pred_quad) {
        if (node→qt0 < 0 || APPROX_GT(node→qt0, Clock.time)) {
            (void) sprintf(ErrorStr, "qt0 = %g", node→qt0);
            node_error(node, "Invalid quadrupole t0 time.", ErrorStr);
        }

        if (node→qts ≤ 0) {
            (void) sprintf(ErrorStr, "qts = %g", node→qts);
            node_error(node, "Invalid quadrupole time-step.", ErrorStr);
        }
    }
}

/* Finally, display message if node is "extended" and/or "packed" */

if (node→extended)
    (void) printf("Extended node:  %s.\n", NodeInfo(node));
    if (node→packed)
        (void) printf("Packed node:  %s.\n", NodeInfo(node));

    /* Repeat procedure for any child branches */

    for (i = 0; i < MAX_NUM_CHILDREN; i++)
        if (node→child_type[i] == BRANCH)
            CheckTree(node→child[i].branch);
}

static void node_error(node, msg1, msg2)
NODE_T *node;
char *msg1, *msg2;
{
    /* Displays any error messages generated by CheckTree(). */

    char str1[MAX_STR_LEN], str2[MAX_STR_LEN];

    (void) sprintf(str1, "CheckTree():  %s", msg1);

    if (node)
        (void) sprintf(str2, "tree index %i:  %s", node→tree_index, msg2);
    else
        (void) strcpy(str2, msg2);

    Error(FATAL, str1, str2);
}

BOOLEAN CheckMultipoles(node)
NODE_T *node;
{
    /*
     * Checks centre of mass (but not derivatives), and quadrupole moments &
     * derivatives of branch "node" by summing up contributions from all
     * leaves of "node" and its descendants. This routine will also check
     * all branches that have "node" as their ancestor, in the same way.
     * All data is assumed to be up to date, so this routine should only be
     * called immediately after "node" and its children have been
     * (re)constructed, and Data[]->pos & Data[]->vel should be correct
     * for the current time. "TRUE" is returned if there is no output from
     * this routine (i.e. no errors).
     *
     */

    int i, k, num_leaves = 0;
    DATA_T *ptr;
    BOOLEAN error_flag = FALSE;

    /* Working variables */

    double xx[3], qq[5], qd[5], qd2[5], qd3[5], mass, error;
    double dx, dy, dz, du, dv, dw, dx2, dy2, dz2, dr2, dr, xdotv, vv2, vv,
        dfx, dfy, dfz, djx, djy, djz, xdotf, vdotf, xdotj, xsc, qsc,
        qdsc, qd2sc, qd3sc;

    /* Initialize to keep GCC happy (no problem if no quadrupole...) */

    du = dv = dw = xdotv = vv2 = vv = dfx = dfy = dfz = djx = djy = djz =
        xdotf = vdotf = xdotj = qsc = qdsc = qd2sc = qd3sc = 0;
```

```
/* Error check */

if (node == NULL)
    Error(FATAL, "CheckMultipoles():  Invalid argument.", "");

/* Get a list of all leaves on and below this node */

GetOffspring(node, &num_leaves, offspring);

if (num_leaves ≠ node→num_leaves) {
    (void) sprintf(ErrorStr, "%i != %i", node→num_leaves, num_leaves);
    Error(FATAL, "CheckMultipoles():  Inconsistent leaf count.", ErrorStr);
}

/* Initialize */

ZERO(xx);

xsc = 0;

if (TreePar.use_quad) {
    for (i = 0; i < NUM_QUAD_ELEM; i++)
        qq[i] = 0;
    qsc = 0;
    if (TreePar.pred_quad) {
        for (i = 0; i < NUM_QUAD_ELEM; i++)
            qd[i] = qd2[i] = qd3[i] = 0;
        qdsc = qd2sc = qd3sc = 0;
    }
}

/* Add in contributions from each particle */

for (i = 0; i < num_leaves; i++) {

    /* Get pointer to particle data */

    if ((ptr = Data[offspring[i]]) == NULL) {
        (void) sprintf(ErrorStr, "offspring = %i", offspring[i]);
        Error(FATAL, "CheckMultipoles():  Offspring not found.", ErrorStr);
    }

    mass = ptr→mass;

    /* Get quantities relative to computed centre of mass */

    dx = ptr→pos[0] - node→pos[0];
    dy = ptr→pos[1] - node→pos[1];
    dz = ptr→pos[2] - node→pos[2];

    dx2 = SQ(dx); dy2 = SQ(dy); dz2 = SQ(dz);
    dr2 = dx2 + dy2 + dz2; dr = sqrt(dr2);

    if (TreePar.pred_quad) {
        du = ptr→vel[0] - node→vel[0];
        dv = ptr→vel[1] - node→vel[1];
        dw = ptr→vel[2] - node→vel[2];

        xdotv = dx * du + dy * dv + dz * dw;
        vv2 = SQ(du) + SQ(dv) + SQ(dw); vv = sqrt(vv2);

        dfx = ptr→f[0] - node→f[0];
        dfy = ptr→f[1] - node→f[1];
        dfz = ptr→f[2] - node→f[2];

        djx = ptr→f_dot[0] - node→f_dot[0];
        djy = ptr→f_dot[1] - node→f_dot[1];
        djz = ptr→f_dot[2] - node→f_dot[2];

        xdotf = dx * dfx + dy * dfy + dz * dfz;
        vdotf = du * dfx + dv * dfy + dw * dfz;
        xdotj = dx * djx + dy * djy + dz * djz;
    }

    /* Accumulate magnitudes for error scaling */

    xsc += mass * dr;

    if (TreePar.use_quad) {
        qsc += mass * dr2;
        if (TreePar.pred_quad) {
            double ff = sqrt(SQ(dfx) + SQ(dfy) + SQ(dfz));

            qdsc += mass * dr * vv;
            qd2sc += mass * (dr * ff + vv2);
            qd3sc += mass * (ff * vv +
                dr * sqrt(SQ(djx) + SQ(djy) + SQ(djz)));
        }
    }

    /* Accumulate error components */

    xx[0] += mass * dx;
    xx[1] += mass * dy;
    xx[2] += mass * dz;

    if (TreePar.use_quad) {

        /*
         * NOTE: Quadrupole terms are computed using PREDICTED
         * centre of mass.
         *
         */

        qq[0] += mass * (2 * dx2 - dy2 - dz2);
        qq[1] += 3 * mass * dx * dy;
        qq[2] += 3 * mass * dx * dz;
        qq[3] += mass * (2 * dy2 - dx2 - dz2);
        qq[4] += 3 * mass * dy * dz;

        if (TreePar.pred_quad) {

            qd[0] += mass * (6 * dx * du - 2 * xdotv);
            qd[1] += 3 * mass * (du * dy + dx * dv);
            qd[2] += 3 * mass * (du * dz + dx * dw);
            qd[3] += mass * (6 * dy * dv - 2 * xdotv);
            qd[4] += 3 * mass * (dv * dz + dy * dw);
            /*
             * Note below that f & f_dot and qd2 & qd3 each contain
             * factors of 1/2 and 1/6, respectively.
             *
             */

            qd2[0] += mass * (6 * dx * dfx + 3 * SQ(du) - 2 * xdotf - vv2);
            qd2[1] += 3 * mass * (dx * dfy + dy * dfx + du * dv);
            qd2[2] += 3 * mass * (dx * dfz + dz * dfx + du * dw);
            qd2[3] += mass * (6 * dy * dfy + 3 * SQ(dv) - 2 * xdotf - vv2);
            qd2[4] += 3 * mass * (dz * dfy + dy * dfz + dw * dv);

            qd3[0] += mass * (6 * dx * djx + 6 * du * dfx -
                2 * (xdotj + vdotf));
            qd3[1] += 3 * mass * (dx * djy + dy * djx +
                du * dfy + dv * dfx);
            qd3[2] += 3 * mass * (dx * djz + dz * djx +
                du * dfz + dw * dfx);
            qd3[3] += mass * (6 * dy * djy + 6 * dv * dfy -
                2 * (xdotj + vdotf));
            qd3[4] += 3 * mass * (dz * djy + dy * djz +
                dw * dfy + dv * dfz);
        }
    }
}   /* for */

/* Check monopole terms */

for (k = 0; k < NUM_PHYS_DIM; k++) {
    error = (xsc == 0 ? 0 : xx[k] / xsc);
    if (TRACK || ABS(error) > TOLERANCE) {
        (void) printf("CHECK: mono %s error[%i] = %g\n",
            NodeInfo(node), k, error);
        error_flag = TRUE;
    }
}

/* Check quadrupole terms as applicable */

if (TreePar.use_quad) {

    for (i = 0; i < NUM_QUAD_ELEM; i++) {
        error = (qsc == 0 ? 0 : (qq[i] - node→q_mom[i]) / qsc);
        if (TRACK || ABS(error) > TOLERANCE) {
            (void) printf("CHECK: quad %s error [%i] = %g\n",
                NodeInfo(node), i, error);
            error_flag = TRUE;
        }
    }

    if (TreePar.pred_quad) {

        /* First derivative */

        for (i = 0; i < NUM_QUAD_ELEM; i++) {
            error = (qdsc == 0 ? 0 : (qd[i] - node→q_dot[i]) / qdsc);
            if (TRACK || ABS(error) > TOLERANCE) {
                (void) printf("CHECK: qdot %s error [%i] = %g\n",
                    NodeInfo(node), i, error);
                error_flag = TRUE;
                }
            }

            /* Second derivative */

            for (i = 0; i < NUM_QUAD_ELEM; i++) {
                error = (qd2sc == 0 ? 0 : (qd2[i] - node→q_2dot[i]) / qd2sc);
                if (TRACK || ABS(error) > TOLERANCE) {
                    (void) printf("CHECK: dq2 %s error [%i] = %g\n",
                        NodeInfo(node), i, error);
                    error_flag = TRUE;
                }
            }

            /* Third derivative */

            for (i = 0; i < NUM_QUAD_ELEM; i++) {
                error = (qd3sc == 0 ? 0 : (qd3[i] - node→q_3dot[i]) / qd3sc);
                if (TRACK || ABS(error) > TOLERANCE) {
                    (void) printf("CHECK: dq3 %s error [%i] = %g\n",
                        NodeInfo(node), i, error);
                    error_flag = TRUE;
                }
            }
        }
    }

    /* Now check branches attached to this branch */

    for (i = 0; i < MAX_NUM_CHILDREN; i++)
        if (node→child_type[i] == BRANCH)
            (void) CheckMultipoles(node→child[i].branch);

    return (error_flag == TRUE ? FALSE : TRUE);
}

void CheckMultipolePrediction()
{
    /*
     * Checks monopole and quadrupole moments of all tree nodes by
     * predicting all particle positions and node moments to the current
     * time and comparing. This routine acts as a driver for the
     * recursive function check_multipole_prediction_r().
     *
     */

    int i, num_nodes = 0;
    double me, me_max, qe, qe_max;

    /*
     * Predict all particle positions (and velocities) to LOW order,
     * to conform with maximum order saved for multipole predictions.
     *
     */

    for (i = 0; i < NumParticles; i++) {
        Data[i]→pos_status = Data[i]→vel_status = UN_PRED;
        PREDICT_POS_LO(Data[i]);
        PREDICT_VEL_LO(Data[i]);
    }
    /* Predict monopole and quadrupole for all nodes */

    PredictPosAndQMomAll(Root);

    /* Initialize */

    me = me_max = 0;

    if (TreePar.use_quad)
        qe = qe_max = 0;

    /* Now descend tree, checking each node in turn */

    check_mult_pred_r(Root, &num_nodes, &me, &me_max, &qe, &qe_max);

    /* Output results of check */

    (void) printf("CHECK -- multipole prediction (time %g):\n", TIME);

    (void) printf(" Monopole avg err %g, max err %g\n",
        sqrt(me) / num_nodes, sqrt(me_max));

    if (TreePar.use_quad)
        (void) printf(" Quadrupole avg err %g, max err %g\n",
            sqrt(qe) / num_nodes, sqrt(qe_max));
}

static void check_mult_pred_r(node, num_nodes, me, me_max, qe, qe_max)
NODE_T *node;
int *num_nodes;
double *me, *me_max, *qe, *qe_max;
{
    /* Recursive counterpart to CheckMultipolePrediction() */

    int i, num_leaves = 0;
    DATA_T *ptr;

    /* Working variables... */

    double xx[3], qq[5], xsc, qsc, mass, errx, errq;
    double dx, dy, dz, dx2, dy2, dz2, dr2, dr;

    /* Increment node counter */

    ++(*num_nodes);

    /* Get a list of all leaves on and below this node */

    GetOffspring(node, &num_leaves, offspring);

    if (num_leaves != node->num_leaves) {
        (void) sprintf(ErrorStr, "%i != %i", node->num_leaves, num_leaves);
        Error(FATAL, "check_mult_pred_r():  Inconsistent leaf count.", ErrorStr);
    }

    /* Initialize moments and scale factors */

    ZERO(xx);
```

```
xsc = qsc = 0;

if (TreePar.use_quad)
    qq[0] = qq[1] = qq[2] = qq[3] = qq[4] = 0;                /* (qq is not a 3-vector) */

/* Add in contributions of each particle in tree */

for (i = 0; i < num_leaves; i++) {

    /* Get pointer to particle data */

    if ((ptr = Data[offspring[i]]) == NULL) {
        (void) sprintf(ErrorStr, "offspring = %i", offspring[i]);
        Error(FATAL, "check_mult_pred_r():  Offspring not found.", ErrorStr);
    }

    mass = ptr→mass;

    /* Get quantities relative to node centre of mass */

    dx = ptr→pos[0] - node→pos[0];
    dy = ptr→pos[1] - node→pos[1];
    dz = ptr→pos[2] - node→pos[2];

    dx2 = SQ(dx); dy2 = SQ(dy); dz2 = SQ(dz);
    dr = sqrt(dr2 = dx2 + dy2 + dz2);

    /* Add scaled quantities to error factors and arrays */

    xsc += mass * dr;
    qsc += mass * dr2;

    xx[0] += mass * dx;
    xx[1] += mass * dy;
    xx[2] += mass * dz;

    /* Add contribution to quadrupole moment if applicable */

    if (TreePar.use_quad) {
        qq[0] += mass * (2.0 * dx2 - dy2 - dz2);
        qq[1] += 3 * mass * dx * dy;
        qq[2] += 3 * mass * dx * dz;
        qq[3] += mass * (2 * dy2 - dx2 - dz2);
        qq[4] += 3 * mass * dy * dz;
    }
}

/* Determine RMS error in moments for this node */

errx = (xsc == 0 ? 0 : DOT(xx, xx) / SQ(xsc));

if (TRACK || errx > TOLERANCE)
    (void) printf("CHECK: mono %s pred error = %g\n",
        NodeInfo(node), sqrt(errx));

*me += errx;
*me_max = MAX(*me_max, errx);

if (TreePar.use_quad) {
    errq = 0;
```

```
        if (qsc > 0) {
            for (i = 0; i < NUM_QUAD_ELEM; i++)
                errq += SQ(qq[i] - node→q_mom[i]);
            errq /= SQ(qsc);
        }

        if (TRACK || errq > TOLERANCE)
            (void) printf("CHECK: quad %s pred error = %g\n",
                NodeInfo(node), sqrt(errq));

        *qe += errq;
        *qe_max = MAX(*qe_max, errq);
    }

    /* Perform same calculations for all branches of this node */

    for (i = 0; i < MAX_NUM_CHILDREN; i++)
        if (node→child_type[i] == BRANCH)
            check_mult_pred_r(node→child[i].branch, num_nodes, me, me_max,
                qe, qe_max);
}

void CheckForce()
{
    /*
     * Checks accuracy of tree force (both mono and quad) against direct
     * force, and displays the results.
     *
     */

    int i;
    DATA_T *ptr;
    double old_force[NUM_PHYS_DIM], direct_force[NUM_PHYS_DIM],
        tree_force_m[NUM_PHYS_DIM], tree_force_q[NUM_PHYS_DIM],
        direct_tree_force[NUM_PHYS_DIM], df, tfm, tfq, dtf, dfe,
        me, qe, mme, mqe, err_sum_df, err_sum_m, err_sum_q;

    int debug = RunPar.debug_level, verbosity = RunPar.verbosity_level;
    BOOLEAN updates = TreePar.check_update_times, use_quad = TreePar.use_quad;

    /* Initialize */

    me = qe = mme = mqe = err_sum_df = err_sum_m = err_sum_q = 0;

    /* Switch off debugging flags and multipole update checks */

    RunPar.debug_level = RunPar.verbosity_level = 0;
    TreePar.check_update_times = FALSE;

    /* Message */

    (void) printf("Checking force on particles (time %g)...\n", TIME);

     /* Check each particle in turn */

    for (i = 0; i < NumParticles; i++) {

        ptr = Data[i];

        /*
```

```
 * Predict position and velocity of current particle to high order
 * (pos'n and velo of other particles reset in force routines).
 *
 */

ptr→pos_status = ptr→vel_status = UN_PRED;
PredictPosAndVelHi(ptr);

/* Save current force */

COPY(ptr→f, old_force);

/* Calculate direct force */

CalcDirectForce(i);

COPY(ptr→f, direct_force);

/* Calculate tree force using monopoles */

TreePar.use_quad = FALSE;

CalcTreeForce(i);

COPY(ptr→f, tree_force_m);

/* Ditto using quadrupoles if applicable */

if (use_quad) {
    TreePar.use_quad = TRUE;
    CalcTreeForce(i);
    COPY(ptr→f, tree_force_q);
}

/* Finally, calculate direct force but follow tree expansions */

TestTreeForce(i);

COPY(ptr→f, direct_tree_force);

/* Restore original force */

COPY(old_force, ptr→f);

/* Now compare and print */

df = MAG(direct_force);
tfm = MAG(tree_force_m);
tfq = (use_quad ? MAG(tree_force_q) : 0);
dtf = MAG(direct_tree_force);

dfe = (dtf == 0 ? 0 : ABS(df - dtf) / dtf);

me = (dtf == 0 ? 0 : ABS(tfm - dtf) / dtf);
mme = MAX(mme, me);

if (use_quad) {
    qe = (dtf == 0 ? 0 : ABS(tfq - dtf) / dtf);
    mqe = MAX(mqe, qe);
}
```

```
        if (verbosity) {
            (void) printf("CHECK: %4i (%4i) tree index %11i", i,
                ptr→orig_index, ptr→node→tree_index);
            (void) printf(" df/dtf %8.2e", dfe);
            (void) printf(" mono %8.2e", me);
            if (use_quad) {
                (void) printf(" quad %8.2e", qe);
                if (qe > me)
                    (void) printf("*");
            }
            (void) printf("\n");
        }

        /* Accumulate statistics */

        err_sum_df += dfe;
        err_sum_m += me;
        if (use_quad)
            err_sum_q += qe;

    }  /* for */

    /* Display RMS force error */

    (void) printf("CHECK: Avg force error df/dtf %8.2e",
        err_sum_df / NumParticles);
    (void) printf(" mono %8.2e max %8.2e", err_sum_m / NumParticles, mme);
    if (use_quad)
        (void) printf(" quad %8.2e max %8.2e", err_sum_q / NumParticles, mqe);
    (void) printf("\n");

    /* Restore flags */

    TreePar.check_update_times = updates;
    RunPar.verbosity_level = verbosity;
    RunPar.debug_level = debug;
}

/* check.c */
```

B.1.8 `draw.c`

This file contains the code for generating movie frames (Sun rasterfiles). There is one global function (`Draw()`) and many local functions. The `Draw()` routine is called by `MakeMovieFrame()` in `output.c`. The routine accepts an integer containing various drawing options, selected bit-wise using the preprocessor aliases defined in `box_tree.h` (e.g. `DRAW_TREE`, etc.). On the first call, `Draw()` sets up a colormap for use with the raster drawing operations. The local functions perform various tasks, such as rotating coordinates and drawing particular shapes. Note that if `SYSV` or `ALPHA` are defined, the entire `draw.c` file is ignored. For information on raster operations and the `pixrect` library, refer to the SunOS 4.1 user manual "Pixrect Reference".

```
/*
 * draw.c - DCR 91-05-08
 * ======================
 * Routines for generating raster files for movies from particle/tree data.
 * These routines will only work for Suns/Sparcs running SunOS 4.
 *
 * Global functions: Draw().
 *
 */

/* Skip if incompatible operating system or architecture */

#ifndef SYSV
#ifndef ALPHA

/* Include files */

#include "box_tree.h"
#include <rasterfile.h>
#include <pixrect/pixrect.h>
#include <pixrect/memvar.h>
#include <pixrect/pr_io.h>

/* Additional definitions */

#define NUM_COLORS 256                                          /* Use all available colors */

#define MAX_COLOR_INTENSITY 255

#define FRAME_SIZE MoviePar.frame_size                               /* For convenience */

/* Rotation angles for cube */

/* Uncomment for rotated cube (doesn't quite work)...
#define ALPHA 0.5 * PI
#define BETA  0.2 * PI
#define GAMMA 0.2 * PI
*/

#define ALPHA 0.0                                                /* Degenerate cube (2D) */
#define BETA 0.0
#define GAMMA 0.0

#define RAY0 100                    /* Light source vector for sphere (RAY0 is magnitude) */
#define RAY1 48
#define RAY2 -36
#define RAY3 80

#define SHADING 0.6                                           /* Shading factor for sphere */

#define NUM_VERTICES MAX_NUM_CHILDREN                                       /* Also 2^n */

/* Local variables */

static Pixrect *mpr;                                                   /* Memory pixrect */
static colormap_t colormap;                                                  /* Colormap */

/* Local functions */

static void
    make_colormap(),
    draw_tree(),
    draw_boxes(),
    plot_pos(),
    plot_vel(),
    plot_com(),
    draw_cell(),
    draw_box(),
    make_2d(),
    rotate(),
    scale(),
    draw_line(),
    draw_object(),
    draw_point(),
    draw_circle(),
    draw_square(),
    draw_diamond(),
    draw_disk(),
    draw_sphere();

/* End of preamble */

void Draw(options)
int options;
{
    /* Main drawing routine */

    static BOOLEAN first_call = TRUE;

    char *movie_filename;
    FILE *rasfile;

    /* Prepare output file */

    if ((movie_filename = MakeFilename(MoviePar.basename,
            MoviePar.frame_number++, ".ras")) == NULL) {
        Error(WARNING2, "Draw():  Movie frame skipped.", "");
        return;
    }

    (void) printf("Writing movie frame %i (time %g)...\n",
            MoviePar.frame_number, TIME);

    if (BackupFiles)
        (void) BackupFile(movie_filename, BACKUP_MARKER);

    if ((rasfile = fopen(movie_filename, "w")) == NULL) {
        Error(IO, "Draw()", movie_filename);
        return;
    }

    /* Construct colormap if first call to Draw() */

    if (first_call) {
        make_colormap();
        first_call = FALSE;
    }

    /* Make space for pixrect */

    if ((mpr = mem_create(FRAME_SIZE, FRAME_SIZE, 8)) == NULL) {
        (void) sprintf(ErrorStr, "frame size = %i", FRAME_SIZE);
        Error(WARNING2, "Draw():  Unable to allocate pixrect memory.", ErrorStr);
        (void) fclose(rasfile);
        return;
    }

    /* Plot data in ascending order of importance */

    if (options & DRAW_TREE)
        draw_tree(Root);

    if (options & DRAW_BOXES)
        draw_boxes();

    if (options & PLOT_COM)
        plot_com(Root, options);

    if (options & PLOT_POS)
        plot_pos();

    if (options & PLOT_VEL)
        plot_vel();

    /* Output raster and close file */

    if (pr_dump(mpr, rasfile, &colormap, 2, 0) ≠ 0) {
        (void) sprintf(ErrorStr, "Draw():  pr_dump() error, code %i", PIX_ERR);
        Error(IO, ErrorStr, movie_filename);
    }

    if (pr_close(mpr) ≠ 0) {
        (void) sprintf(ErrorStr, "Draw():  pr_close() error, code %i", PIX_ERR);
        Error(IO, ErrorStr, movie_filename);
    }

    if (fclose(rasfile) == EOF)
        Error(IO, "Draw()", movie_filename);
}

static void make_colormap()
{
    /* Constructs colormap with simple colors plus gray scale */

    int i;
    COLOR_T red[NUM_COLORS], green[NUM_COLORS], blue[NUM_COLORS];
    double gray_scale;

    colormap.type = RMT_EQUAL_RGB;
    colormap.length = NUM_COLORS;

    for (i = 0; i < 3; i++)
        colormap.map[i] = (COLOR_T *) malloc(NUM_COLORS * sizeof(COLOR_T));

    red[BLACK] = 0; green[BLACK] = 0; blue[BLACK] = 0;
    red[RED] = 255; green[RED] = 0; blue[RED] = 0;
    red[PINK] = 255; green[PINK] = 0; blue[PINK] = 159;
    red[YELLOW] = 255; green[YELLOW] = 255; blue[YELLOW] = 0;
    red[GREEN] = 0; green[GREEN] = 255; blue[GREEN] = 0;
    red[ORANGE] = 255; green[ORANGE] = 191; blue[ORANGE] = 0;
    red[PURPLE] = 255; green[PURPLE] = 0; blue[PURPLE] = 255;
    red[CYAN] = 0; green[CYAN] = 255; blue[CYAN] = 255;
    red[BLUE] = 0; green[BLUE] = 0; blue[BLUE] = 255;

    gray_scale = (double) MAX_COLOR_INTENSITY / (LAST_GRAY - FIRST_GRAY);
    for (i = FIRST_GRAY; i ≤ LAST_GRAY; i++)
        red[i] = green[i] = blue[i] = gray_scale ∗ (i - FIRST_GRAY);

    for (i = 0; i < colormap.length; i++) {
        colormap.map[0][i] = red[i];
        colormap.map[1][i] = green[i];
        colormap.map[2][i] = blue[i];
    }
}

static void draw_tree(node)
NODE_T ∗node;
{
    /∗ Routine to draw connected cell boxes (recursive) ∗/

    int i, j, k;
    double r, x[NUM_TREE_DIM][NUM_VERTICES];

    r = 0.5 ∗ node→half_size;

    for (i = 0; i < MAX_NUM_CHILDREN; i++) {
        switch (node→child_type[i]) {
            case EMPTY:
            case LEAF:
                for (j = 0; j < NUM_VERTICES; j++) {
                    for (k = 0; k < NUM_TREE_DIM; k++)
                        x[k][j] = node→centre[k] + ChildCoordOffset[k][i] ∗ r +
                            ChildCoordOffset[k][j] ∗ r;
                    make_2d(&x[0][j], NUM_VERTICES);
                }
                draw_cell(x, WHITE);
                break;
            case BRANCH:
                draw_tree(node→child[i].branch);
        }
    }
}

static void draw_boxes()
{
    /∗ Routine to draw ghost boxes (2D tree only) ∗/

    int i, k;
    double centre[NUM_BOX_DIM];

    for (i = 0; i < NumBoxes; i++) {
        for (k = 0; k < NUM_BOX_DIM; k++)
            centre[k] = SYS_CENTRE[k] + BOX_POS[i][k];
        draw_box(centre, HALF_BOX_SIZE, YELLOW);
    }
}

static void plot_pos()
{
    /∗ Routine to draw points at particle positions ∗/

    int i, j, k, index[MAX_NUM_PARTICLES];
    DATA_T ∗ptr;
    double z[MAX_NUM_PARTICLES], x[NUM_PHYS_DIM][MAX_NUM_BOXES],
          pos[NUM_PHYS_DIM];

     /* Get positions, preparing "z" for sorting if required */

     for (i = 0; i < NumParticles; i++) {
          index[i] = i;
          if (MoviePar.hide_blocked_objects) {
               ptr = Data[i];
               if (RunPar.use_tree && NUM_TREE_DIM == 3) {
                    for (j = 0; j < NumBoxes; j++) {
                         COPY(ptr→pos, pos);
                         ADD_BOX_OFFSET(pos, j);
                         if (RunPar.bc_opt == PERIODIC)
                              REDUCE(pos);
                         for (k = 0; k < NUM_PHYS_DIM; k++)
                              x[k][j] = pos[k];
                         make_2d(&x[0][j], MAX_NUM_BOXES);
                    }
                    z[i] = x[2][0];
               }
               else
                    z[i] = ptr→pos[2];
          }
     }

     /* Sort by "z" (i.e. projected) component */

     if (MoviePar.hide_blocked_objects)
          Sort2(NumParticles, z, index);

     /* Draw particles */

     for (i = 0; i < NumParticles; i++) {
          ptr = Data[index[i]];
          for (j = 0; j < NumBoxes; j++) {
               COPY(ptr→pos, pos);
               ADD_BOX_OFFSET(pos, j);
               if (RunPar.bc_opt == PERIODIC)
                    REDUCE(pos);
               for (k = 0; k < NUM_PHYS_DIM; k++)
                    x[k][j] = pos[k];
               make_2d(&x[0][j], MAX_NUM_BOXES);
          }

          for (j = 0; j < NumBoxes; j++)
               draw_object(x[0][j], x[1][j], x[2][j], ptr→radius,
                         MoviePar.particle_shape, ptr→color);
     }
}

#define DELTA_T 0.001                                    /* Arbitrary velocity scaling */

static void plot_vel()
{
     /* Routine to draw velocity vectors at particle positions */

     int i, j, k;
     double pos[NUM_PHYS_DIM], vel[NUM_PHYS_DIM], x[NUM_PHYS_DIM][2];

     for (i = 0; i < NumParticles; i++)
```

```
        for (j = 0; j < NumBoxes; j++) {
            COPY(Data[i]→pos, pos);
            ADD_BOX_OFFSET(pos, j);
            COPY(Data[i]→vel, vel);
            ADD_BOX_SHEAR(vel, j);
            if (RunPar.bc_opt == PERIODIC)
                REDUCE(pos);
            for (k = 0; k < NUM_PHYS_DIM; k++) {
                x[k][0] = pos[k];
                x[k][1] = x[k][0] + vel[k] * DELTA_T;
            }
            make_2d(&x[0][0], 2);
            make_2d(&x[0][1], 2);
            draw_line(0, 1, x[0], x[1], PIX_SRC, GREEN);
        }
}

#undef DELTA_T

static void plot_com(node, options)
NODE_T *node;
int options;
{
    /*
     * Routine to draw points at centre-of-mass positions, optionally
     * connecting lines between points (recursive).
     *
     */

    int i, k;
    double x[NUM_PHYS_DIM];

    if (TreePar.pred_mono)
        PREDICT_COM_POS(node);

    COPY(node→pos, x);

    make_2d(x, 1);

    draw_object(x[0], x[1], 0.0, 0.01 * VIEW_SIZE / MoviePar.radius_mag,
        DIAMOND, (COLOR_T) (node == Root ? PINK : BLUE));

    if (options & COM_LINES) {
        int j, particle;
        CHILD_T *child;
        COLOR_T c[MAX_NUM_CHILDREN];
        double v[NUM_PHYS_DIM][MAX_NUM_CHILDREN + 1];

        for (j = i = 0; i < MAX_NUM_CHILDREN; i++) {
            child = &node→child[i];
            switch (node→child_type[i]) {
            case EMPTY:
                break;
            case BRANCH:
                if (TreePar.pred_mono)
                    PREDICT_COM_POS(child→branch);
                for (k = 0; k < NUM_PHYS_DIM; k++)
                    v[k][j] = (child→branch)→pos[k];
                c[j] = BLUE;
                ++j;
                break;
            case LEAF:                                  /* (note particles already predicted) */
                particle = child→leaf;
                for (k = 0; k < NUM_PHYS_DIM; k++)
                    v[k][j] = Data[particle]→pos[k];
                c[j] = RED;
                ++j;
            }
        }
        if (j > 0) {
            v[0][j] = x[0];
            v[1][j] = x[1];
            for (i = 0; i < j; i++) {
                make_2d(&v[0][i], MAX_NUM_CHILDREN + 1);
                draw_line(i, j, v[0], v[1], PIX_SRC, c[i]);
            }
        }
    }

    /* Repeat for child branches */

    for (i = 0; i < MAX_NUM_CHILDREN; i++)
        if (node→child_type[i] == BRANCH)
            plot_com(node→child[i].branch, options);
}

static void draw_cell(x, color)
double x[NUM_TREE_DIM][NUM_VERTICES];
COLOR_T color;
{
    /* Draws a single cell */

    int op = PIX_SRC;
    double *xp = x[0], *yp = x[1];

    if (NUM_TREE_DIM == 2) {
        draw_line(0, 1, xp, yp, op, color);
        draw_line(1, 3, xp, yp, op, color);
        draw_line(3, 2, xp, yp, op, color);
        draw_line(2, 0, xp, yp, op, color);
    }
    else {
        draw_line(0, 1, xp, yp, op, color);                          /* Bottom plane */
        draw_line(1, 3, xp, yp, op, color);
        draw_line(3, 2, xp, yp, op, color);
        draw_line(2, 0, xp, yp, op, color);
        draw_line(4, 5, xp, yp, op, color);                             /* Top plane */
        draw_line(5, 7, xp, yp, op, color);
        draw_line(7, 6, xp, yp, op, color);
        draw_line(6, 4, xp, yp, op, color);
        draw_line(0, 4, xp, yp, op, color);                      /* Vertical "struts" */
        draw_line(1, 5, xp, yp, op, color);
        draw_line(2, 6, xp, yp, op, color);
        draw_line(3, 7, xp, yp, op, color);
    }
}

static void draw_box(centre, half_size, color)
double *centre, half_size;
COLOR_T color;
```

```
{
    /* Draws a box (2D only) */

    int i, k;
    double x[NUM_BOX_DIM][NUM_VERTICES];

    if (NUM_TREE_DIM ≠ 2)                          /* (will do better than this someday...) */
        return;

    for (i = 0; i < NUM_VERTICES; i++) {
        for (k = 0; k < NUM_BOX_DIM; k++)
            x[k][i] = centre[k] + ChildCoordOffset[k][i] * half_size;
        make_2d(&x[0][i], NUM_VERTICES);
    }

    draw_cell(x, color);
}

static void make_2d(x, inc)
double *x;
int inc;
{
    /*
     * Converts 2- or 3- dimensional coordinates to appropriate
     * two-dimensional drawing coordinates.
     *
     */

    if (NUM_TREE_DIM == 3)
        rotate(x, x + inc, x + 2 * inc);                           /* Parallel projection */

    scale(x, x + inc);
}

static void rotate(x, y, z)
double *x, *y, *z;
{
    /* Rotates 3D coordinates by angles ALPHA, BETA, and GAMMA */

    double xx, yy, zz;

    xx = *x;
    yy = *y;
    zz = *z;

    *x = cos(ALPHA) * cos(BETA) * xx + (sin(ALPHA) * cos(GAMMA) +
        cos(ALPHA) * sin(BETA) * sin(GAMMA)) * yy + (sin(ALPHA) * sin(GAMMA) -
        cos(ALPHA) * sin(BETA) * cos(GAMMA)) * zz;

    *y = - sin(ALPHA) * cos(BETA) * xx + (cos(ALPHA) * cos(GAMMA) -
        sin(ALPHA) * sin(BETA) * sin(GAMMA)) * yy + (cos(ALPHA) * sin(GAMMA) +
        sin(ALPHA) * sin(BETA) * cos(GAMMA)) * zz;

    *z = sin(BETA) * xx - cos(BETA) * sin(GAMMA) * yy +
        cos(BETA) * cos(GAMMA) * zz;
}

static void scale(x, y)
double *x, *y;
{
    /* Scales and shifts coordinates to pixel positions on raster */

    *x = (*x - VIEW_CENTRE[0] + HALF_VIEW_SIZE) * FRAME_SIZE / VIEW_SIZE;
    *y = (HALF_VIEW_SIZE - *y + VIEW_CENTRE[1]) * FRAME_SIZE / VIEW_SIZE;
}

static void draw_line(v1, v2, x, y, op, color)
int v1, v2, op;
double x[], y[];
COLOR_T color;
{
    /* Draws line between two vertices */

    pr_vector(mpr, (int) x[v1], (int) y[v1], (int) x[v2], (int) y[v2], op,
        color);
}

static void draw_object(x, y, z, r, object, color)
double x, y, z, r;
SHAPE_T object;
COLOR_T color;
{
    /* Draws object of specified size and color at specified position */

    double denom, size;

    if ((denom = MoviePar.distance - z * MoviePar.z_mag) == 0 ||
            (size = (r / VIEW_SIZE) * FRAME_SIZE * MoviePar.radius_mag *
            MoviePar.distance / denom) < 0 || size > FRAME_SIZE) {
        Error(WARNING2, "draw_object():  Too large -- not drawn.", "");
        return;
    }

    switch (object) {
        case POINT:
            draw_point(x, y, color);
            break;
        case CIRCLE:
            draw_circle(x, y, size, color);
            break;
        case SQUARE:
            draw_square(x, y, size, color);
            break;
        case DIAMOND:
            draw_diamond(x, y, size, color);
            break;
        case DISK:
            draw_disk(x, y, size, color);
            break;
        case SPHERE:
            draw_sphere(x, y, size, color);
            break;
        default:
            (void) sprintf(ErrorStr, "object type = %i", object);
            Error(FATAL, "draw_object():  Unknown object type.", ErrorStr);
    }
}

static void draw_point(x, y, color)
double x, y;
```

```
COLOR_T color;
{
    /* Plots a single pixel */

    (void) pr_put(mpr, (int) x, (int) y, color);
}

static void draw_circle(x, y, r, color)
double x, y, r;
COLOR_T color;
{
    /* Draws a circle */

    int i, num_pix, ix, iy;
    double theta;

    num_pix = TWO_PI * r;

    for (i = 0; i < num_pix; i++) {
        theta = i / r;
        ix = x + r * cos(theta);
        iy = y + r * sin(theta);
        (void) pr_put(mpr, ix, iy, color);
    }
}

static void draw_square(x, y, size, color)
double x, y, size;
COLOR_T color;
{
    /* Draws a square */

    int i, j;

    for (i = - size; i ≤ size; i++)
        for (j = -size; j ≤ size; j++)
            (void) pr_put(mpr, (int) x + i, (int) y + j, color);
}

static void draw_diamond(x, y, size, color)
double x, y, size;
COLOR_T color;
{
    /* Draws a diamond (square rotated 45 degrees) */

    int i, j;

    for (i = - size; i ≤ size; i++)
        for (j = ABS(i) - size; j ≤ size - ABS(i); j++)
            (void) pr_put(mpr, (int) x + i, (int) y + j, color);
}

static void draw_disk(x, y, r, color)
double x, y, r;
COLOR_T color;
{
    /* Draws a disk (filled-in circle) */

    int xx, yy, ix, iy;
    xx = r;

    for (ix = -xx; ix ≤ xx; ix++) {
        yy = sqrt((double) (SQ(r) - SQ(ix)));
        for (iy = -yy; iy ≤ yy; iy++)
            (void) pr_put(mpr, (int) x + ix, (int) y + iy, color);
    }
}

static void draw_sphere(x, y, r, color)
double x, y, r;
COLOR_T color;
{
    /*
     * Draws a shaded sphere (based on algorithm "spheres.c" 1.4 88/02/05
     * Copyright 1986 Sun Microsystems).
     *
     */

    int xx, yy, ix, iy;
    double limit;

    limit = r * RAY0 * SHADING;
    xx = r;

    for (ix = -xx; ix ≤ xx; ix++) {
        yy = sqrt(SQ(r) - SQ(ix));
        for (iy = -yy; iy ≤ yy; iy++) {
            if (Ran() * limit ≤ RAY1 * ix + RAY2 * iy +
                    RAY3 * sqrt(SQ(r) - SQ(ix) - SQ(iy)))
                (void) pr_put(mpr, (int) x + ix, (int) y + iy, color);
            else
                (void) pr_put(mpr, (int) x + ix, (int) y + iy, BLACK);
        }
    }
}

#endif
#endif

/* draw.c */
```

B.1.9 extern.c

This file simply contains the global variable declarations that are referenced with the `extern` keyword in `box_tree.h`. Note that some of the global variables are initialized here.

```
/*
 * extern.c - DCR 91-04-30
 * =======================
 * External variable declarations for box_tree.
 *
 */

/* Include files */

#include "box_tree.h"
```

```
/* Logfile: pointer to log file (NULL if logging disabled) */

FILE *Logfile;

/* SaveFilename: Name of file for restart data */

char SaveFilename[MAX_FILENAME_LEN];

/* BackupFiles: TRUE to move existing files rather than overwrite them */

BOOLEAN BackupFiles;

/* NumParticles: Number of particles */

int NumParticles;

/* NumBoxes: Number of boxes to use (0, 1, or 9) */

int NumBoxes;

/* RunPar: Other main parameters to use for each run */

RUN_PAR_T RunPar;

/* EvolPar: Evolving parameters (e.g. velocity dispersion, etc.) */

EVOL_PAR_T EvolPar;

/* TreePar: Parameters for tree */

TREE_PAR_T TreePar;

/* MoviePar: Parameters for movies */

MOVIE_PAR_T MoviePar;

/* DebugPar: Debugging/self-check parameters */

DEBUG_PAR_T DebugPar;

/* Clock: Various program timers */

CLOCK_T Clock;

/* Counter: Array of performance-monitoring counters */

int Counter[NUM_COUNTERS];

/* Data: Array of pointers to particle data structures */

DATA_T *Data[MAX_NUM_PARTICLES];

/* Tsl: Time-step list for Integrate() */

TSL_T Tsl;

/* Root: Pointer to root cell (node structure) of tree */

NODE_T *Root;
```

```
/* Initialize tree index and coordinate offset arrays */

int ChildIndexOffset[NUM_TREE_DIM] = CHILD_INDEX_OFFSET_ARRAY;
int ChildCoordOffset[NUM_TREE_DIM][MAX_NUM_CHILDREN] =
CHILD_COORD_OFFSET_ARRAY;

/* Workspace: Dummy character array for string manipulation */

char Workspace[WORKSPACE_SIZE];

/* ErrorStr: Storage for error messages */

char ErrorStr[WORKSPACE_SIZE];

/* External variables for mathematical macros/definitions */

double OneThird = (double) 1 / 3;
double TwoThirds = (double) 2 / 3;
double FourThirds = (double) 4 / 3;
double OneSixth = (double) 1 / 6;
double OneNinth = (double) 1 / 9;
double OneTwelfth = (double) 1 / 12;

/* End of preamble */

/* extern.c */
```

B.1.10 `force.c`

All of the `box_tree` force calculation routines are contained in this file. The functions `CalcTreeForce()` and `CalcDirectForce()` are self-explanatory, returning the force per unit mass on a given particle using the corresponding calculation method. The `AddGhostForce()` routine simply calculates the force contribution of ghost particles using the direct method (the function is called by `InitLoOrderPoly()` in `integrate.c`). The `TestTreeForce()` routine uses the direct method but takes into account the fact that ghost nodes may be wrapped around the box system if necessary to minimize asymmetry (cf. §3.4.2). With the exception of `AddGhostForce()`, these routines all call the local function `initialize()` to zero the force on the given particle, initialize the structure that will contain data concerning the closest particle, and reset the prediction flags (used to prevent re-predicting particle positions and velocities unnecessarily). The `add_tree_force()` routine is used with `CalcTreeForce()` exclusively. The routine is recursive, and descends the tree in the manner described in §2.2 to calculate the force contributions of nodes and particles. Node contributions are added in using `add_node_multipole_force()`; particle contributions are added using `add_direct_force()`. The latter routine is used by `CalcDirectForce()` and `AddGhostForce()` as well. It is also used by the other two global force routines for any particles that are excluded from the tree. In addition to calculating the force contribution of a particle, `add_direct_force()` checks to see if the particle is a candidate for being the closest neighbour, using `CheckForCp1()` or `CheckForCp3()` as appropriate (see `misc.c`).

There are other local functions in `force.c` that are used only for tree force checking. The routine `add_direct_tree_force()` is the equivalent of `add_tree_force()` and is for use with `TestTreeForce()`. The function `add_node_direct_force()` sums up the force contribution of all the particles in a node and is the direct equivalent of the function `add_node_multipole_force()`. The routine `quick_direct_force()` is a stripped-down version of `add_direct_force()` for use with the test routines. Note that this version does not include a neighbour check. The function `check_force()` compares the direct force and multipole force contribution of a node, and calls the routine `check_multipole_force()` if the discrepancy is large. This latter function performs an exhaustive check on the force contribution of the node, and includes a calculation of the octupole component as well.

Note the use of the local macro `CALC_R2_DATA()` for calculating the relative position and square distance between two given position vectors. The results are stored in `rel_pos[]` and `r2`, respectively, which are variables local to the file. Such in-line coding provides a noticeable improvement in CPU efficiency.

```
/*
 * force.c - DCR 91-05-17
 * ======================
 * Routines for calculating forces (accelerations) on particles.
 *
 * Global functions: CalcTreeForce(), CalcDirectForce(), AddGhostForce(),
 *     TestTreeForce().
 */

/* Include files */

#include "box_tree.h"

/* Additional definitions (in-line code for faster execution) */

#define CALC_R2_DATA(pos1, pos2) {\
    rel_pos[0] = pos1[0] - pos2[0];\
    rel_pos[1] = pos1[1] - pos2[1];\
    rel_pos[2] = pos1[2] - pos2[2];\
    r2 = SQ(rel_pos[0]) + SQ(rel_pos[1]) + SQ(rel_pos[2]);\
}

/* Local variables */

static double rel_pos[NUM_PHYS_DIM], r2;                /* (see CALC_R2_DATA() above) */

/* Local function declarations */

static void
    initialize(),
    add_tree_force(),
    add_direct_force(),
    add_node_multipole_force(),
    add_direct_tree_force(),
    add_node_direct_force(),
    quick_direct_force(),
    check_force(),
    check_multipole_force();

/* End of preamble */

void CalcTreeForce(particle)
int particle;
{
    /*
     * Obtains force on "particle" using current tree. Contributions
     * from ghost particles are obtained from ghost trees if desired.
     * This routine also obtains closest particle data for "particle".
     * This data and the new force are stored in the particle's data
     * structure.
     *
     */

    int i, j;

    /* Initialize force and closest-particle structure */

    initialize(particle);

    /* Loop over central and ghost boxes */

    for (i = 0; i < NumBoxes; i++)
        add_tree_force(particle, i, Root);

    /* Use direct force with any excluded particles */

    for (i = 0; i < TreePar.num_excluded; i++)
        for (j = 0; j < NumBoxes; j++)
            if (TreePar.exclude_list[i] ≠ particle || j ≠ CENTRE)
                add_direct_force(particle, TreePar.exclude_list[i], j);
}

void CalcDirectForce(particle)
int particle;
{
    /*
     * Obtains force on "particle" by direct summation over all other
     * particles (and ghosts if desired). Also obtains data concerning
     * closest particle, if any.
     *
     */

    int i, j;

    initialize(particle);

    for (i = 0; i < NumParticles; i++)
        for (j = 0; j < NumBoxes; j++)
            if (i ≠ particle || j ≠ CENTRE)
                add_direct_force(particle, i, j);
}

void AddGhostForce(particle)
int particle;
{
    /*
     * Adds contribution to force on "particle" due to ghost particles
     * only, by direct summation. Note that it is assumed that a call
     * to initialize() is not required.
     *
     */

    int i, j;

    for (i = 0; i < NumParticles; i++)
        for (j = 0; j < NumBoxes; j++)
            if (j ≠ CENTRE)
                add_direct_force(particle, i, j);
}
```

```
void TestTreeForce(particle)
int particle;
{
    /*
     * Obtains force on "particle" by direct summation but on a node-by-
     * node basis, using the opening angle rule of the tree. This prevents
     * particles in a ghost node from wrapping around in the y direction
     * before the node itself is wrapped. If there are no ghosts, this
     * routine should give the same result as AddTreeForce(). Note that
     * closest particle info is NOT generated.
     *
     */

    int i, j;

    initialize(particle);

    for (i = 0; i < NumBoxes; i++)
        add_direct_tree_force(particle, i, Root);

    for (i = 0; i < TreePar.num_excluded; i++)
        for (j = 0; j < NumBoxes; j++)
            if (TreePar.exclude_list[i] ≠ particle || j ≠ CENTRE)
                add_direct_force(particle, TreePar.exclude_list[i], j);
}

static void initialize(particle)
int particle;
{
    /* Initializes "particle" data in preparation for force calculations */

    static double last_time = 0;

    int i;
    DATA_T *ptr = Data[particle];

    /* Zero force */

    ZERO(ptr→f);

    /* Initialize closest-particle structure */

    InitCp(particle);

    /* Reset prediction flags of all other particles if new time */

    if (Clock.time == last_time)
        return;

    for (i = 0; i < NumParticles; i++)
        if (i ≠ particle)
            Data[i]→pos_status = Data[i]→vel_status = UN_PRED;

    last_time = Clock.time;
}

static void add_tree_force(particle, box, node)
int particle, box;
NODE_T *node;
```

```
{
    /*
     * Adds contribution to force on "particle" due to "node" in "box"
     * depending on whether angle subtended by node from particle is
     * small enough. Otherwise contributions from node's children are
     * considered recursively. Checks for closest particle are performed
     * only on leaves that are summed over directly. Called from
     * CalcTreeForce().
     *
     */

    DATA_T *ptr = Data[particle];
    double node_pos[NUM_PHYS_DIM];

    /* Update monopole if necessary, otherwise predict c-o-m position */

    if (TreePar.pred_mono) {
        if (TreePar.check_update_times && Clock.time - node→mt0 > node→mts) {
            UpdateMonopole(node);
            ++Counter[TOTAL_MONO_UPDATES];
        }
        else
            PREDICT_COM_POS(node);
    }

    /* Apply ghost correction if applicable */

    COPY(node→pos, node_pos);

    if (box ≠ CENTRE) {
        ADD_BOX_OFFSET(node_pos, box);
        WRAP(node_pos);
    }

    /* Get distance info for branch centre of mass */

    CALC_R2_DATA(ptr→pos, node_pos);

    /*
     * If angle subtended by node size at centre of mass is small enough,
     * use multipole expansion to approximate force. Otherwise, recursively
     * consider node children, using direct force on any leaves.
     *
     * Note: the maximum node size is used when calculating the opening
     * angle (c.f. get_max_size() in update_tree.c). Also, node "self-
     * gravity" problems are checked for only if the particle-node distance
     * squared is less than EvolPar.self_grav_r2 (because checking is quite
     * expensive). Also note that any changes made to the expansion
     * criterion here must also be made in the corresponding check routine
     * (viz. add_direct_tree_force()).
     *
     */

    if (node→max_size_sq < TreePar.theta_sq * r2 &&
            (r2 > EvolPar.self_grav_r2 || NotOffspring(particle, node))) {

        /* Message if self-gravity self_grav_r2 criterion insufficient... */

        if (MONITOR && box == CENTRE && !NotOffspring(particle, node)) {
            (void) sprintf(ErrorStr, "particle %i (%i) dist %e %s",
                particle, ptr→orig_index, sqrt(r2), NodeInfo(node));
            Error(WARNING2, "add_tree_force():  Self-grav problem detected.",
                ErrorStr);
        }

        /* Check multipole expansion if desired, otherwise just accept it */

        if (DebugPar.check_force) {
            double old_force[NUM_PHYS_DIM];

            COPY(ptr→f, old_force);
            add_node_multipole_force(particle, box, node);
            check_force(particle, box, node, old_force);
        }
        else
            add_node_multipole_force(particle, box, node);
    }
    else {
        int i;
        CHILD_T *child;

        /* Consider node children */

        for (i = 0; i < MAX_NUM_CHILDREN; i++) {
            child = &node→child[i];
            if (node→child_type[i] == BRANCH)
                add_tree_force(particle, box, child→branch);
            else if (node→child_type[i] == LEAF &&
                    (child→leaf ≠ particle || box ≠ CENTRE)) {
                add_direct_force(particle, child→leaf, box);
            }
        }
    }
}

static void add_direct_force(particle0, particle, box)
int particle0, particle, box;
{
    /*
     * Adds contribution to force on "particle0" due to "particle" in "box"
     * using Newtonian law for point masses. Checks for closest particle
     * are also performed.
     *
     */

    DATA_T *ptr0 = Data[particle0], *ptr = Data[particle];
    double cp_pos[NUM_PHYS_DIM], r2_inv, f;

    /* Predict position of "particle" to first order in force derivative */

    PREDICT_POS_LO(ptr);

    /*
     * Get distance info between passed position and particle, allowing
     * for ghost particle case.
     *
     */

    COPY(ptr→pos, cp_pos);
    if (box ≠ CENTRE) {
        ADD_BOX_OFFSET(cp_pos, box);
        WRAP(cp_pos);
    }

    CALC_R2_DATA(ptr0→pos, cp_pos);

    /* Check whether this is closest particle distance so far */

    if (r2 < EvolPar.cp_zone_sq) {
        if (RunPar.self_grav)
            CheckForCp1(particle0, particle, box, cp_pos, r2);
        else
            CheckForCp3(particle0, particle, box, cp_pos, r2, rel_pos);
    }

    /* Return now if self-gravity switched off */

    if (!RunPar.self_grav)
        return;

    /* Otherwise add contribution to force, including softening if desired */

    if (RunPar.use_softening)
        r2 += MAX(ptr0→radius_sq, ptr→radius_sq);

    r2_inv = 1 / r2;

    f = - ptr→mass * r2_inv * sqrt(r2_inv);

    ptr0→f[0] += f * rel_pos[0];
    ptr0→f[1] += f * rel_pos[1];
    ptr0→f[2] += f * rel_pos[2];
}

static void add_node_multipole_force(particle, box, node)
int particle, box;
NODE_T *node;
{
    /*
     * Adds contribution of "node" to force on "particle" using multipole
     * expansion about centre of mass. This routine is for use with
     * add_tree_force().
     *
     */

    DATA_T *ptr = Data[particle];
    double node_pos[NUM_PHYS_DIM], r2_inv, r3_inv, f_mag, qrx, qry, qrz,
        rqr, r5_inv;

    /* Return if self-gravity switched off */

    if (!RunPar.self_grav)
        return;

    /* Add ghost correction if applicable */

    COPY(node→pos, node_pos);

    if (box ≠ CENTRE) {
        ADD_BOX_OFFSET(node_pos, box);
        WRAP(node_pos);
    }

    /* Get distance info */

    CALC_R2_DATA(ptr→pos, node_pos);

    /* Monopole term */

    if (RunPar.use_softening)
        r2 += ptr→radius_sq;

    r2_inv = 1 / r2;
    r3_inv = r2_inv * sqrt(r2_inv);
    f_mag = node→mass * r3_inv;

    ptr→f[0] -= f_mag * rel_pos[0];
    ptr→f[1] -= f_mag * rel_pos[1];
    ptr→f[2] -= f_mag * rel_pos[2];

    /* Return now if not using quadrupole */

    if (TreePar.use_quad == FALSE)
        return;

    /* Update quadrupole if necessary */

    if (TreePar.pred_quad) {
        if (TreePar.check_update_times && Clock.time - node→qt0 > node→qts) {
            UpdateQuadrupole(node);
            ++Counter[TOTAL_QUAD_UPDATES];
        }
        else
            PREDICT_Q_MOM(node);
    }

    /* Calculate Q dot r */

    qrx = node→q_mom[0] * rel_pos[0] + node→q_mom[1] * rel_pos[1] +
        node→q_mom[2] * rel_pos[2];
    qry = node→q_mom[1] * rel_pos[0] + node→q_mom[3] * rel_pos[1] +
        node→q_mom[4] * rel_pos[2];
    qrz = node→q_mom[2] * rel_pos[0] + node→q_mom[4] * rel_pos[1] -
        (node→q_mom[0] + node→q_mom[3]) * rel_pos[2];

    /* Calculate r dot Q dot r and multiply by 5/2r^2 */

    rqr = 2.5 * r2_inv *
        (rel_pos[0] * qrx + rel_pos[1] * qry + rel_pos[2] * qrz);

    r5_inv = r2_inv * r3_inv;

    /* Now add quadrupole contribution */

    ptr→f[0] += (qrx - rqr * rel_pos[0]) * r5_inv;
    ptr→f[1] += (qry - rqr * rel_pos[1]) * r5_inv;
    ptr→f[2] += (qrz - rqr * rel_pos[2]) * r5_inv;
}
```

```
static void add_direct_tree_force(particle, box, node)
int particle, box;
NODE_T *node;
{
    /*
     * Same as AddTreeForce() except add_node_direct_force() and
     * quick_direct_force() are used in place of add_multipole_force()
     * and add_direct_force(), respectively. This routine is for use
     * with TestTreeForce().
     *
     */

    DATA_T *ptr0 = Data[particle];
    double node_pos[NUM_PHYS_DIM], node_offset = 0;

    /* Predict c-o-m position if desired (update NOT performed) */

    if (TreePar.pred_mono)
        PREDICT_COM_POS(node);

    /* Obtain distance info, recording wrap-around if ghost node */

    COPY(node→pos, node_pos);

    if (box ≠ CENTRE) {
        double offset;

        ADD_BOX_OFFSET(node_pos, box);

        while (ABS(node_pos[1]) > HALF_SYS_SIZE) {
            offset = SGN(node_pos[1]) * SYS_SIZE;
            node_pos[1] -= offset;
            node_offset += offset;
        }
    }

    CALC_R2_DATA(ptr0→pos, node_pos);

    /* Perform opening angle check */

    if (node→max_size_sq < TreePar.theta_sq * r2 &&
            (r2 > EvolPar.self_grav_r2 || NotOffspring(particle, node)))
        add_node_direct_force(particle, box, node, node_offset);
    else {
        int i;
        CHILD_T *child;

        for (i = 0; i < MAX_NUM_CHILDREN; i++) {
            child = &node→child[i];
            if (node→child_type[i] == BRANCH)
                add_direct_tree_force(particle, box, child→branch);
            else if (node→child_type[i] == LEAF &&
                    (child→leaf ≠ particle || box ≠ CENTRE)) {
                DATA_T *ptr = Data[child→leaf];
                double leaf_pos[NUM_PHYS_DIM];

                PREDICT_POS_LO(ptr);
                COPY(ptr→pos, leaf_pos);
                if (box ≠ CENTRE) {
                    ADD_BOX_OFFSET(leaf_pos, box);
```

```
                        WRAP(leaf_pos);
                    }
                    quick_direct_force(ptr0, ptr, leaf_pos);
                }
            }
        }
}

static void add_node_direct_force(particle, box, node, offset)
int particle, box;
NODE_T *node;
double offset;
{
    /*
     * Performs direct summation of forces over leaves in "node",
     * offsetting positions as required for ghost box wrap-around.
     * For use with add_direct_tree_force() and check_force().
     *
     */

    int i, num_leaves = 0;
    DATA_T *ptr0 = Data[particle], *ptr;
    LEAF_T offspring[MAX_NUM_PARTICLES];
    double leaf_pos[NUM_PHYS_DIM];

    GetOffspring(node, &num_leaves, offspring);

    for (i = 0; i < num_leaves; i++) {
        if (VERBOSE && particle == offspring[i]) {
            Error(WARNING2, "add_node_direct_force():  self-grav problem.", "");
            continue;
        }
        ptr = Data[offspring[i]];
        PREDICT_POS_LO(ptr);
        COPY(ptr→pos, leaf_pos);
        ADD_BOX_OFFSET(leaf_pos, box);
        leaf_pos[1] -= offset;
        quick_direct_force(ptr0, ptr, leaf_pos);
    }
}

static void quick_direct_force(ptr0, ptr, pos)
DATA_T *ptr0, *ptr;
double *pos;
{
    /* Calculates Newtonian force between "ptr0" and "ptr" (at "pos") */

    double r2_inv, f;

    CALC_R2_DATA(ptr0→pos, pos);

    if (RunPar.use_softening)
        r2 += MAX(ptr0→radius_sq, ptr→radius_sq);

    r2_inv = 1 / r2;

    f = - ptr→mass * r2_inv * sqrt(r2_inv);

    ptr0→f[0] += f * rel_pos[0];
    ptr0→f[1] += f * rel_pos[1];
    ptr0→f[2] += f * rel_pos[2];
}

#define LARGE_ERROR 0.25                               /* Message if error exceeds this */
#define HUGE_ERROR 1.0                                   /* Crash if error exceeds this */
#define ERROR_STATS_INTERVAL 250000             /* Stats output interval in time-steps */

static void check_force(particle, box, node, old_force)
int particle, box;
NODE_T *node;
double *old_force;
{
    /*
     * Compares "old_force" obtained from "node" with direct force,
     * accumulating error statistics in DebugPar and printing out
     * messages if the errors are large. Called from add_tree_force().
     *
     */

    DATA_T *ptr = Data[particle];
    DEBUG_PAR_T *ptrd = &DebugPar;
    double node_pos[NUM_PHYS_DIM], node_offset = 0, tf[NUM_PHYS_DIM],
        new_force[NUM_PHYS_DIM], df[NUM_PHYS_DIM], dfm, tfm, err;

    /* Increment counter */

    ++(ptrd→num_force_checks);

    /* Get distance info, allowing for ghost case */

    COPY(node→pos, node_pos);

    if (box ≠ CENTRE) {
        double offset;

        ADD_BOX_OFFSET(node_pos, box);

        while (ABS(node_pos[1]) > HALF_SYS_SIZE) {
            offset = SGN(node_pos[1]) * SYS_SIZE;
            node_pos[1] -= offset;
            node_offset += offset;
        }
    }

    /* ptr->f contains "old_force" plus force due to "node" */

    COPY(ptr→f, new_force);

    /* Hence obtain force due to "node" in isolation */

    SUB(new_force, old_force, tf);

    /* Obtain direct force from particles in node, allowing for wrap-around */

    ZERO(ptr→f);

    add_node_direct_force(particle, box, node, node_offset);

    COPY(ptr→f, df);
    /* Restore new force */

    COPY(new_force, ptr->f);

    /* Accumulate statistics and calculate errors */

    dfm = MAG(df);
    tfm = MAG(tf);

    ptrd->avg_force += dfm;
    ptrd->max_force = MAX(ptrd->max_force, dfm);

    err = ABS(dfm - tfm) / dfm;

    /* Message if large error, and check multipoles explicitly */

    if (err > LARGE_ERROR) {
        ++Counter[FORCE_ERRORS];
        (void) printf("MONITOR: %i (%i) t = %g:  abs err %.0f%%",
            particle, ptr->orig_index, TIME, 100 * err);
        if (ptrd->max_force > 0)
            (void) printf(" (%%avg %.1e %%max %.1e)\n",
                100 * ptrd->num_force_checks * tfm / ptrd->avg_force,
                100 * tfm / ptrd->max_force);
        else
            (void) printf("\n");
        (void) printf(" box %i particle pos %e %e %e\n", box,
            ptr->pos[0], ptr->pos[1], ptr->pos[2]);
        (void) printf(" node info:  %s\n", NodeInfo(node));
        (void) printf(" node geom pos %e %e %e\n",
            node_pos[0], node_pos[1], node_pos[2]);
        check_multipole_force(particle, box, node, df, tf);
        if (err > HUGE_ERROR)
            Error(FATAL, "check_force():  Error too large.", "");
    }

    /* Display error statistics at regular intervals */

    ptrd->total_err += err;
    ptrd->max_err = MAX(ptrd->max_err, err);

    if (ptrd->num_force_checks % ERROR_STATS_INTERVAL == 0) {
        (void) printf("MONITOR -- Force error, t %g:  ", TIME);
        (void) printf("avg abs err = %.2g%%, max err = %.2g%%\n",
            100 * ptrd->total_err / ptrd->num_force_checks,
            100 * ptrd->max_err);
    }
}

#undef ERROR_STATS_INTERVAL
#undef HUGE_ERROR
#undef LARGE_ERROR

static void check_multipole_force(particle, box, node, df, tf)
int particle, box;
NODE_T *node;
double *df, *tf;
{
    /*
     * Calculates monopole, quadrupole, and octupole contributions
```

```
 * of "node" in "box" to force on "particle" explicitly and
 * compares with supplied direct ("df") and tree ("tf") forces.
 * Called from check_force().
 *
 */

DATA_T *ptr0 = Data[particle], *ptr;

int i, k, num_leaves = 0;
LEAF_T offspring[MAX_NUM_PARTICLES];
double node_pos[NUM_PHYS_DIM], node_offset = 0, leaf_pos[NUM_PHYS_DIM],
    true_pos[NUM_PHYS_DIM], total_mass, node_posm, true_posm, pose,
    tfm[NUM_PHYS_DIM], tfq[NUM_PHYS_DIM], tfo[NUM_PHYS_DIM];

/* Working variables... */

double m, q11, q22, q21, q31, q32, s11, s22, s33, s12, s13, s21, s23,
    s31, s32, s123, dx, dy, dz, dx2, dy2, dz2, r, r_inv, r2_inv,
    r3_inv, r5_inv, r7_inv, r9_inv, qrx, qry, qrz, rqr, sd1, sd2,
    sd3, phioct, dfm, tfx, tfmm, tfqm, tfom, tfe, tfme, tfqe, tfoe;

/* Get node leaves and adjust positions for any wrap-around */

COPY(node->pos, node_pos);

if (box != CENTRE) {
    double offset;

    ADD_BOX_OFFSET(node_pos, box);

    while (ABS(node_pos[1]) > HALF_SYS_SIZE) {
        offset = SGN(node_pos[1]) * SYS_SIZE;
        node_pos[1] -= offset;
        node_offset += offset;
    }
}

GetOffspring(node, &num_leaves, offspring);

/* Calculate "true" centre of mass */

ZERO(true_pos);
total_mass = 0;

for (i = 0; i < num_leaves; i++) {
    ptr = Data[offspring[i]];
    PREDICT_POS_LO(ptr);
    COPY(ptr->pos, leaf_pos);
    ADD_BOX_OFFSET(leaf_pos, box);
    leaf_pos[1] -= node_offset;
    m = ptr->mass;
    total_mass += m;
    for (k = 0; k < NUM_PHYS_DIM; k++)
        true_pos[k] += m * leaf_pos[k];
}

NORM(true_pos, total_mass);

(void) printf("Predicted node c-o-m position:  %12.5e %12.5e %12.5e\n",
    node_pos[0], node_pos[1], node_pos[2]);
```

```
(void) printf("Actual node c-o-m position:  %12.5e %12.5e %12.5e\n",
    true_pos[0], true_pos[1], true_pos[2]);
node_posm = sqrt(SQ(node_pos[0]) + SQ(node_pos[1]) + SQ(node_pos[2]));
true_posm = sqrt(SQ(true_pos[0]) + SQ(true_pos[1]) + SQ(true_pos[2]));
pose = 100 * ABS(node_posm - true_posm) / true_posm;
(void) printf("Absolute error:  %11.5e%%\n", pose);

/* Calculate multipole moments */

q11 = q21 = q31 = q22 = q32 = 0;
s11 = s22 = s33 = s12 = s13 = s21 = s23 = s31 = s32 = s123 = 0;

for (i = 0; i < num_leaves; i++) {
    ptr = Data[offspring[i]];
    COPY(ptr→pos, leaf_pos);
    ADD_BOX_OFFSET(leaf_pos, box);
    leaf_pos[1] -= node_offset;

    if (num_leaves < 10 && pose < 10)
        (void) printf("Leaf %i (%i) pos %12.5e %12.5e %12.5e\n",
            offspring[i], ptr→orig_index, leaf_pos[0],
            leaf_pos[1], leaf_pos[2]);

    /* Monopole moment is total mass */

    m = ptr→mass;

    /* Quadrupole moment */

    CALC_R2_DATA(leaf_pos, true_pos);

    dx = rel_pos[0];
    dy = rel_pos[1];
    dz = rel_pos[2];

    dx2 = SQ(dx);
    dy2 = SQ(dy);
    dz2 = SQ(dz);

    q11 += m * (3 * dx2 - r2);
    q22 += m * (3 * dy2 - r2);
    q21 += 3 * m * dx * dy;
    q31 += 3 * m * dx * dz;
    q32 += 3 * m * dy * dz;

    /* Octupole moment */

    s11 += m * (dx2 - 1.5 * (dy2 + dz2)) * dx;
    s22 += m * (dy2 - 1.5 * (dx2 + dz2)) * dy;
    s33 += m * (dz2 - 1.5 * (dx2 + dy2)) * dz;
    s12 += 6 * m * (dx2 - 0.25 * (dy2 + dz2)) * dy;
    s13 += 6 * m * (dx2 - 0.25 * (dy2 + dz2)) * dz;
    s21 += 6 * m * (dy2 - 0.25 * (dx2 + dz2)) * dx;
    s23 += 6 * m * (dy2 - 0.25 * (dx2 + dz2)) * dz;
    s31 += 6 * m * (dz2 - 0.25 * (dx2 + dy2)) * dx;
    s32 += 6 * m * (dz2 - 0.25 * (dx2 + dy2)) * dy;
    s123 += 15 * m * dx * dy * dz;
}

/* Calculate multipole expansions of force */
```

```
CALC_R2_DATA(ptr0→pos, true_pos);

dx = rel_pos[0];
dy = rel_pos[1];
dz = rel_pos[2];

dx2 = SQ(dx);
dy2 = SQ(dy);
dz2 = SQ(dz);

r = sqrt(r2);
r_inv = 1 / r;
r2_inv = SQ(r_inv);

/* Monopole contribution */

r3_inv = r_inv * r2_inv;

tfm[0] = - total_mass * dx * r3_inv;
tfm[1] = - total_mass * dy * r3_inv;
tfm[2] = - total_mass * dz * r3_inv;

/* Quadrupole contribution */

qrx = q11 * dx + q21 * dy + q31 * dz;
qry = q21 * dx + q22 * dy + q32 * dz;
qrz = q31 * dx + q32 * dy - (q11 + q22) * dz;

rqr = qrx * dx + qry * dy + qrz * dz;

r5_inv = r2_inv * r3_inv;
r7_inv = r2_inv * r5_inv;

tfq[0] = tfm[0] + qrx * r5_inv - 2.5 * rqr * dx * r7_inv;
tfq[1] = tfm[1] + qry * r5_inv - 2.5 * rqr * dy * r7_inv;
tfq[2] = tfm[2] + qrz * r5_inv - 2.5 * rqr * dz * r7_inv;

/* Octupole contribution */

sd1 = s11 * dx + s12 * dy + s13 * dz;
sd2 = s21 * dx + s22 * dy + s23 * dz;
sd3 = s31 * dx + s32 * dy + s33 * dz;

r9_inv = r2_inv * r7_inv;

phioct = - 7 * r9_inv * (dx2 * sd1 + dy2 * sd2 + dz2 * sd3 +
              dx * dy * dz * s123);

tfo[0] = tfq[0] + dx * phioct + (2 * dx * sd1 + dx2 * s11 + dy2 * s21 +
         dz2 * s31 + dy * dz * s123) * r7_inv;

tfo[1] = tfq[1] + dy * phioct + (2 * dy * sd2 + dx2 * s12 + dy2 * s22 +
         dz2 * s32 + dx * dz * s123) * r7_inv;

tfo[2] = tfq[2] + dz * phioct + (2 * dz * sd3 + dx2 * s13 + dy2 * s23 +
         dz2 * s33 + dx * dy * s123) * r7_inv;

/* Evaluate performance... */
```

```
    dfm = sqrt(SQ(df[0]) + SQ(df[1]) + SQ(df[2]));
    tfx = sqrt(SQ(tf[0]) + SQ(tf[1]) + SQ(tf[2]));
    tfmm = sqrt(SQ(tfm[0]) + SQ(tfm[1]) + SQ(tfm[2]));
    tfqm = sqrt(SQ(tfq[0]) + SQ(tfq[1]) + SQ(tfq[2]));
    tfom = sqrt(SQ(tfo[0]) + SQ(tfo[1]) + SQ(tfo[2]));

    tfe = 100 * ABS(tfx - dfm) / dfm;
    tfme = 100 * ABS(tfmm - dfm) / dfm;
    tfqe = 100 * ABS(tfqm - dfm) / dfm;
    tfoe = 100 * ABS(tfom - dfm) / dfm;

    (void) printf("method f_x f_y f_z f\n");
    (void) printf("direct %12.5e %12.5e %12.5e %11.5e\n",
            df[0], df[1], df[2], dfm);
    (void) printf("tree %12.5e %12.5e %12.5e %11.5e\n",
            tf[0], tf[1], tf[2], tfx);
    (void) printf("m only %12.5e %12.5e %12.5e %11.5e\n",
            tfm[0], tfm[1], tfm[2], tfmm);
    (void) printf("q only %12.5e %12.5e %12.5e %11.5e\n",
            tfq[0], tfq[1], tfq[2], tfqm);
    (void) printf("o only %12.5e %12.5e %12.5e %11.5e\n",
            tfo[0], tfo[1], tfo[2], tfom);
    (void) printf("errors t-d %f%% m-d %f%% q-d %f%% o-d %f%% ",
            tfe, tfme, tfqe, tfoe);
    if (tfqe > tfme)
        (void) printf("***");
    if (tfoe > tfqe)
        (void) printf("###");
    if (tfoe > tfme)
        (void) printf("!!!");
    (void) printf("\n");
}

/* force.c */
```

B.1.11 `init_cond.c`

The various routines for generating or loading initial conditions are in this file. There is only one global function, `SetInitCond()`, which calls various local functions depending on the initial conditions option. Memory for particle data structures is allocated in this file (note that `lint` may complain about `malloc()` commands that allocate memory for complex structures; these messages can safely be ignored). After obtaining the basic particle data, `SetInitCond()` performs various initializations, such as zeroing collision counters and assigning particle colours. A boundary condition check is also performed for supplied initial conditions. Finally, a call is made to `CalcEvolPar()` (cf. `output.c`) in order to initialize various data-dependent parameters, such as the self-gravity check zone (cf. §A.5.1).

The various local functions will only be described briefly. The first of these functions, `set_aligned_com()`, generates the "aligned c-o-m" initial conditions. This is accomplished by choosing particle positions in uniform random fashion, subtracting the centre-of-mass position of the resulting configuration, and repositioning any particles that end up outside the box as a result of the centre-of-mass shift. This process in repeated until all particles are inside the box. The `set_uniform_ran()` routine places sets of particles in grids laid out across the box, choosing positions in each grid from a uniform random distribution. Any particles left over are distributed randomly throughout the entire box. The centre-of-mass position is *not* adjusted to coincide with the origin. However, as with `set_aligned_com()`, the centre-of-mass velocity is set to zero by offsetting each particle velocity (the local function `sub_com_vel()` is used for this purpose). Both routines make use of `place_particle()`, which chooses a random position in a region of the central box in which to place a given particle. The function also sets the mass, radius, moment of inertia, and drag factor of the particle (choosing from a mass distribution if desired). The velocity is also set, using any initial dispersions (see §A.4.2). Currently the initial spin is set to zero. To eliminate initial overlaps or (if desired) potential binaries, `place_particle()` calls the `BOOLEAN` function `rejected_particle()`, which returns `TRUE` if the current particle position should be rejected.

The routines `wt()` and `pack_box()` generate WT-type and close-packed initial conditions, respectively. The `wt()` function makes use of the `BOOLEAN` function `check_pos()` to prevent particle overlaps, including overlaps with ghosts (something which is *not* done in `rejected_particle()`). Finally, the function `read_init_cond()` is used to read a text file containing preset initial conditions.

```
/*
 * init_cond.c - DCR 91-06-27
 * ===========================
 * Routines for initializing particle data.
 *
 * Global functions: SetInitCond().
 *
 */

/* Include files */

#include "box_tree.h"

/* Additional definitions */

#define SX RunPar.init_x_vel_disp
#define SY RunPar.init_y_vel_disp
#define SZ RunPar.init_z_vel_disp

/* Local functions */

static void
    set_aligned_com(),
    set_uniform_ran(),
    place_particle(),
    wt(),
    pack_box(),
    read_init_cond(),
    sub_com_vel();

static BOOLEAN rejected_particle(), check_pos();

/* Local variables */

static BOOLEAN particle_placed[MAX_NUM_PARTICLES];             /* place_particle() flags */
static int num_rejects = 0;                                     /* For rejected_particle() */
static double est_mean_mass = 0.0;                              /* Estimated mean mass */
static double max_rr = 0.0;                                     /* Maximum Roche radius */

/* End of preamble */

#define IC_OPT ptr→ic_opt
```

```
void SetInitCond()
{
    /*
     * Allocates memory for particle data, assigns initial positions,
     * velocities, and spins (according to initial conditions option)
     * and initializes several other quantities.
     *
     */

    int i, j;
    COLOR_T color;
    DATA_T *ptrd;
    RUN_PAR_T *ptr = &RunPar;

    if (IC_OPT ≠ SUPPLIED) {

        /* Assign storage space */

        for (i = 0; i < NumParticles; i++)
            Data[i] = (DATA_T *) malloc(sizeof(DATA_T));

        /* Assign default colors */

        for (i = 0; i < NumParticles; i++)
            Data[i]→color = MoviePar.dflt_color;

        /* Initialize flags if required */

        if (IC_OPT == ALIGNED_COM || IC_OPT == UNIFORM_RAN) {
            for (i = 0; i < NumParticles; i++)
                particle_placed[i] = FALSE;
            max_rr = RocheRadius(RunPar.init_max_mass);
        }

        if (IC_OPT == ALIGNED_COM || IC_OPT == UNIFORM_RAN || IC_OPT == WT) {
            est_mean_mass = EstMeanMass();
            (void) printf("Estimated mean mass = %e\n", est_mean_mass);
        }
    }

    /* Place particles in box according to desired scheme */

    switch (IC_OPT) {
        case ALIGNED_COM:
            set_aligned_com();
            break;
        case UNIFORM_RAN:
            set_uniform_ran();
            break;
        case CLOSE_PACKED:
            pack_box();
            break;
        case WT:
            wt();
            break;
        case SUPPLIED:
            read_init_cond();
            break;
        default:
            (void) sprintf(ErrorStr, "init cond opt = %i", IC_OPT);
        Error(FATAL, "SetInitCond():  Unknown/invalid option.", ErrorStr);
}

/* Perform other initializations for each particle */

ptr→total_mass = 0;

for (i = 0; i < NumParticles; i++) {

    ptrd = Data[i];

    ptr→total_mass += ptrd→mass;

    /* Apply shear to particle y-velocity (within box) if req'd */

    if (IC_OPT ≠ SUPPLIED || ptr→add_shear)
        ADD_SHEAR(ptrd);

    /* Set original-particle-number array entry */

    if (IC_OPT ≠ SUPPLIED)
        ptrd→orig_index = i;

    /* Disable predictions until integration has begun */

    ptrd→pos_status = ptrd→vel_status = NO_PRED;

    /* Initialize collision counters */

    ptrd→last_collider = -1;
    ptrd→num_collisions = 0;

    /* Set default tree occupancy flag */

    ptrd→in_tree = ptr→use_tree;

    /* Initialize particle node data */

    ptrd→node = NULL; ptrd→node_index = -1;

    /* Assign monitor flags and tracking colors */

    ptrd→monitor = FALSE;

    for (j = 0; j < ptr→num_to_track; j++)
        if (ptrd→orig_index == ptr→track_list[j]) {
            if ((color = ptr→track_colors[j]) ≠ MoviePar.dflt_color) {
                ptrd→monitor = TRUE;
                if (color == BLACK)
                    color = MoviePar.dflt_color;
            }
            ptrd→color = color;
            break;
        }
}  /* for */

/* Exclude particles from tree if requested */

if (ptr→use_tree)
    for (i = 0; i < TreePar.num_excluded; i++)
```

```
            Data[TreePar.exclude_list[i]]→in_tree = FALSE;

    /* Check boundary conditions for supplied positions */

    if (IC_OPT == SUPPLIED && RunPar.bc_opt ≠ UNBOUNDED) {
        for (i = 0; i < NumParticles; i++)
            if (OUTSIDE_CENTRE(Data[i]→pos)) {
                (void) sprintf(ErrorStr, "particle %i (%i) x %g y %g", i,
                    Data[i]→orig_index, Data[i]→pos[0], Data[i]→pos[1]);
                Error(WARNING2,
                    "SetInitCond():  Particle outside centre -- applying BCs",
                    ErrorStr);
                (void) ApplyBndryCond(i);
            }
    }

    /* Calculate/set initial evolving parameters */

    CalcEvolPar();

    /* Check mass estimate */

    if (IC_OPT == ALIGNED_COM || IC_OPT == UNIFORM_RAN || IC_OPT == WT) {
        double x;

        if ((x = ABS(EvolPar.mean_mass - est_mean_mass) / est_mean_mass)
                > 0.01) {
            (void) sprintf(ErrorStr, "mean mass dev = %f%%", 100 * x);
            Error(WARNING2, "SetInitCond():  Poor statistics?", ErrorStr);
        }
    }

    /* Zero CPU timer */

    EvolPar.total_cpu = 0.0;
}

#undef IC_OPT

static void set_aligned_com()
{
    /*
     * Arranges particles randomly in centre box but adjusts positions
     * and velocities so that c-o-m position = SYS_CENTRE and c-o-m
     * velocity = 0. An iterative procedure is used to ensure all
     * particles are confined to centre box after adjustements.
     *
     */

    int i, j, k, num_active_particles, active_list[MAX_NUM_PARTICLES],
        num_outside;
    double total_mass, com_pos[NUM_PHYS_DIM], x;

    (void) printf("Aligning centre of mass and box centre...\n");

    if (!ROTATING_FRAME)
        Error(WARNING2, "set_aligned_com():  Assuming SHM in z.", "");

    /* Initialize */
```

```
total_mass = 0;

ZERO(com_pos);

num_active_particles = NumParticles;

for (i = 0; i < NumParticles; i++)
    active_list[i] = i;

/* Keep looping until all particles fit inside box */

while (num_active_particles) {

    /* Assign positions and velocities and add to centre of mass */

    for (i = 0; i < num_active_particles; i++) {
        j = active_list[i];
        place_particle(j, SYS_CENTRE, BOX_SIZE, BOX_SIZE);
        total_mass += Data[j]→mass;
        for (k = 0; k < NUM_PHYS_DIM; k++)
            com_pos[k] += Data[j]→mass * Data[j]→pos[k];
    }

    /* Check if c-o-m correction leaves any particles outside box */

    for (num_outside = i = 0; i < NumParticles; i++) {
        for (k = 0; k < NUM_BOX_DIM; k++) {
            x = Data[i]→pos[k] - com_pos[k] / total_mass;
            if (ABS(x) > HALF_BOX_SIZE) {
                (void) printf("Particle %i outside box.\n", i);
                active_list[num_outside++] = i;
                break;
            }
        }
    }

    (void) printf("No.  particles outside box = %i\n", num_outside);

    /* Remove contribution to c-o-m of outside particles */

    num_active_particles = num_outside;

    for (i = 0; i < num_active_particles; i++) {
        j = active_list[i];
        total_mass -= Data[j]→mass;
        for (k = 0; k < NUM_PHYS_DIM; k++)
            com_pos[k] -= Data[j]→mass * Data[j]→pos[k];
        particle_placed[j] = FALSE;
    }
}  /* while */

/* Finally, align c-o-m by subtracting components from positions */

NORM(com_pos, total_mass);

for (i = 0; i < NumParticles; i++)
    SUB(Data[i]→pos, com_pos, Data[i]→pos);

/* Output some statistics */
```

```
    (void) printf("Done!  %i rejected pair(s), dx = %f, dy = %f\n",
        num_rejects, com_pos[0], com_pos[1]);

    /* Finally, align centre-of-mass velocity as well */

    sub_com_vel();
}

static void set_uniform_ran()
{
    /* Arranges particles in 2-D uniform random fashion in (centre) box */

    int i, ix, iy, particle, nx = RunPar.num_x_div, ny = RunPar.num_y_div,
        subbox_num_particles = NumParticles / (nx * ny);
    double subbox_size_x = BOX_SIZE / nx, subbox_size_y = BOX_SIZE / ny,
        subbox_centre[NUM_BOX_DIM], offset_x, offset_y;

    if (!ROTATING_FRAME)
        Error(WARNING2, "set_uniform_ran():  Assuming SHM in z.", "");

    particle = 0;

    /* Loop over subboxes */

    for (ix = 0; ix < nx; ix++)
        for (iy = 0; iy < ny; iy++) {
            offset_x = (ix - 0.5 * (nx - 1)) * subbox_size_x;
            offset_y = (iy - 0.5 * (ny - 1)) * subbox_size_y;
            subbox_centre[0] = SYS_CENTRE[0] + offset_x;
            subbox_centre[1] = SYS_CENTRE[1] + offset_y;
            for (i = 0; i < subbox_num_particles; i++)
                place_particle(particle++, subbox_centre, subbox_size_x,
                    subbox_size_y);
        }

    /* Place any leftover particles randomly */

    for (i = particle; i < NumParticles; i++)
        place_particle(i, SYS_CENTRE, BOX_SIZE, BOX_SIZE);

    /*
     * Subtract c-o-m velocity from all particles if rotating frame.
     * This ensures tzam is conserved (actually only need x component
     * of c-o-m velocity to be zero).
     *
     */

    if (ROTATING_FRAME)                 /* (redundant: currently must be rotating frame) */
        sub_com_vel();
}

static void place_particle(particle, centre, size_x, size_y)
int particle;
double *centre, size_x, size_y;
{
    /*
     * Places "particle" randomly in subbox with position "centre" and
     * dimension "size_x" and "size_y". Initial velocities are weighted by
     * by the dispersions "SX", "SY", and "SZ".
     *
```

```
 */

DATA_T *ptr = Data[particle];
double sqrt2 = sqrt(2.0), vel_norm, pos[NUM_PHYS_DIM], vel[NUM_PHYS_DIM],
    phase;

/* Error checks */

if (ERROR_CHECK) {
    if (centre[0] + size_x > SYS_CENTRE[0] + BOX_SIZE ||
        centre[0] - size_x < SYS_CENTRE[0] - BOX_SIZE ||
        centre[1] + size_y > SYS_CENTRE[1] + BOX_SIZE ||
        centre[1] - size_y < SYS_CENTRE[1] - BOX_SIZE)
            Error(FATAL, "place_particle():  Subbox too big!", "");
    if (particle_placed[particle]) {
        (void) sprintf(ErrorStr, "particle %i", particle);
        Error(FATAL, "place_particle():  Particle already placed.",
            ErrorStr);
    }
}

/* Choose mass then set radius and moment of inertia */

if (particle == 0 && RunPar.seed_mass)
    ptr→mass = RunPar.seed_mass;
else
    ptr→mass = InitMassFunc(Ran());

ptr→radius = Radius(ptr→mass);
ptr→radius_sq = SQ(ptr→radius);
ptr→inertia = MomentOfInertia(ptr→mass, ptr→radius);

/* Velocity dispersion normalization for non-uniform masses */

vel_norm = sqrt(est_mean_mass / ptr→mass);

/* Calculate drag factor */

ptr→drag_fac = DragFactor(ptr→mass);

/* Choose position and velocity, rejecting close pairs */

do {
    if (particle == 0 && RunPar.seed_mass) {
        ZERO(pos);
        ZERO(vel);
        break;
    }

    pos[0] = centre[0] + size_x * (Ran() - 0.5);
    pos[1] = centre[1] + size_y * (Ran() - 0.5);

    vel[0] = vel_norm * SX * Gasdev();
    vel[1] = vel_norm * SY * Gasdev();

    /* z position and velocity assigned assuming unit SHM frequency */

    phase = TWO_PI * Ran();

    if (RunPar.init_scale_height == 0) {
            double z_max = SZ * Gasdev();

            pos[2] = sqrt2 * vel_norm * z_max * cos(phase);
            vel[2] = - sqrt2 * vel_norm * z_max * sin(phase);
        }
        else {
            double z_max = max_rr * RunPar.init_scale_height;

            pos[2] = vel_norm * z_max * cos(phase);
            vel[2] = - vel_norm * z_max * sin(phase);
        }

    } while(rejected_particle(particle, ptr→radius, pos, vel));

    /* Save particle position and velocity */

    COPY(pos, ptr→pos);
    COPY(vel, ptr→vel);

    /* Set spin to zero initially */

    ZERO(ptr→spin);

    /* Set flag */

    particle_placed[particle] = TRUE;
}

#define MAX_NUM_REJECTS NumParticles                /* Allow this many rejects maximum */

static BOOLEAN rejected_particle(particle, radius, pos, vel)
int particle;
double radius, *pos, *vel;
{
    /*
     * Returns TRUE if "particle" is too close to another.
     * Note that ghost particles are currently not considered.
     *
     */

    int i, k;
    double r2, sum_radii_sq;
    BOOLEAN reject;

    /* If using softening and initial binaries are ok, just return */

    if (RunPar.use_softening && !RunPar.rej_init_bin)
        return FALSE;

    /* Loop over placed particles */

    for (i = 0; i < NumParticles; i++) {

        if (i == particle || !particle_placed[i])
            continue;

        /* Reset flag */

        reject = FALSE;
        /*
         * Reject if overlapping and not using softening, otherwise check
         * for binary and reject it if desired.
         *
         */

        sum_radii_sq = SQ(radius + Data[i]→radius);

        for (r2 = 0.0, k = 0; k < NUM_PHYS_DIM; k++)
            r2 += SQ(pos[k] - Data[i]→pos[k]);

        if (!RunPar.use_softening && r2 < sum_radii_sq)
            reject = TRUE;
        else if (RunPar.rej_init_bin) {

            /* Only consider particles within 10 maximum Roche radii */

            if (r2 < 100 * SQ(max_rr)) {
                double v2, semi_inv;

                for (v2 = 0.0, k = 0; k < NUM_PHYS_DIM; k++)
                    v2 += SQ(vel[k] - Data[i]→vel[k]);

                semi_inv = 2 / sqrt(r2) - v2 / (Data[particle]→mass +
                    Data[i]→mass);

                /* Reject binary */

                if (semi_inv > 0 && 1 / semi_inv < max_rr)
                    reject = TRUE;
            }
        }

        /* Increment counter and abort if too many rejects */

        if (reject) {
            if (++num_rejects > MAX_NUM_REJECTS)
                Error(FATAL, "rejected_particle():  Too many rejects.", "");
            return TRUE;
        }
    }  /* for */

    return FALSE;
}

#undef MAX_NUM_REJECTS

#define MAX_NUM_TRIES 100000          /* Maximum number of placement attempts allowed */

static void wt()
{
    /*
     * Assigns initial conditions conforming with Wisdom J., Tremaine S.,
     * 1988, AJ, 95, 925. For use in rotating frame only.
     *
     */

    int odd, n, i0, i, k, counter;
    DATA_T *ptr1, *ptr2;
    double r, h, f, m_ratio, mri, tau;
```

```
BOOLEAN need_pos;

/* Need N odd if seed mass, even otherwise */

odd = (n = NumParticles) % 2;
n -= (i0 = (RunPar.seed_mass ? 1 : 0));
odd -= i0;

if (i0 && odd)
    Error(FATAL, "wt():  Unable to create seed mass -- use odd N.", "");

if (odd)
    Error(FATAL, "wt():  Odd N not supported for WT init.  cond.", "");

/* Heuristic check */

r = Radius(RunPar.init_max_mass);
h = RunPar.init_disk_thickness * r;

if (0.1 * h * SQ(BOX_SIZE) / CUBE(r) < NumParticles)
    Error(WARNING2, "wt():  Insufficient space for particles?", "");

/* Construct seed mass if desired */

if (i0) {
    ptr1 = Data[0];
    ptr1→mass = RunPar.seed_mass;
    ptr1→radius = Radius(ptr1→mass);
    ptr1→radius_sq = SQ(ptr1→radius);
    ptr1→inertia = MomentOfInertia(ptr1→mass, ptr1→radius);
    ptr1→drag_fac = DragFactor(ptr1→mass);
    ZERO(ptr1→pos);
    ZERO(ptr1→vel);
    ZERO(ptr1→spin);
    (void) printf("[seed mass placed at origin]\n");
}

/* Place particles pair-wise */

for (i = i0; i < n; i += 2) {
    ptr1 = Data[i];
    ptr2 = Data[i + 1];

    /*
     * Following gives smooth mass distribution, going from biggest
     * to smallest to aid positioning.
     *
     */

    f = 1 - (double) (i - i0) / (n - 1);
    ptr1→mass = InitMassFunc(f);
    f = 1 - (double) (i + 1 - i0) / (n - 1);
    ptr2→mass = InitMassFunc(f);
    if (ptr1→mass < ptr2→mass)
        Error(FATAL, "wt():  Mass increasing.", "");
    ptr1→radius = Radius(ptr1→mass);
    ptr1→radius_sq = SQ(ptr1→radius);
    ptr2→radius = Radius(ptr2→mass);
    ptr2→radius_sq = SQ(ptr2→radius);
```

```
        /* Attempt to place particles in box */

        counter = 0;
        need_pos = TRUE;
        while (need_pos) {
            if (counter++ > MAX_NUM_TRIES)
                Error(FATAL, "wt():  Too many positioning attempts.", "");
            m_ratio = ptr2→mass / ptr1→mass;
            ptr1→pos[0] = m_ratio * BOX_SIZE * (Ran() - 0.5);
            ptr1→pos[1] = m_ratio * BOX_SIZE * (Ran() - 0.5);
            ptr1→pos[2] = m_ratio * h * (Ran() - 0.5);
            need_pos = check_pos(ptr1→pos[0], ptr1→pos[1], ptr1→pos[2],
                ptr1→radius, i);
            if (need_pos)
                continue;
            mri = - 1 / m_ratio;
            for (k = 0; k < NUM_PHYS_DIM; k++)
                ptr2→pos[k] = mri * ptr1→pos[k];
            need_pos = check_pos(ptr2→pos[0], ptr2→pos[1], ptr2→pos[2],
                ptr2→radius, i + 1);
        }

        /* Assign velocities */

        for (k = 0; k < NUM_PHYS_DIM; k++) {
            ptr1→vel[k] = (2 * Ran() - 1) * ptr1→radius;
            ptr2→vel[k] = (2 * Ran() - 1) * ptr2→radius;
        }

        /* Compute remaining quantities */

        ZERO(ptr1→spin);
        ZERO(ptr2→spin);
        ptr1→inertia = MomentOfInertia(ptr1→mass, ptr1→radius);
        ptr2→inertia = MomentOfInertia(ptr2→mass, ptr2→radius);
        ptr1→drag_fac = DragFactor(ptr1→mass);
        ptr2→drag_fac = DragFactor(ptr2→mass);
    }

    /* Ensure zero c-o-m velocity */

    ptr1 = Data[i0];                                  /* Largest mass (excluding seed) */
    ZERO(ptr1→vel);
    for (i = i0 + 1; i < n; i++)
        for (k = 0; k < NUM_PHYS_DIM; k++)
            ptr1→vel[k] -= Data[i]→mass * Data[i]→vel[k];
    NORM(ptr1→vel, ptr1→mass);
    (void) printf("[Particle %i vel set to %e %e %e]\n", i0,
        ptr1→vel[0], ptr1→vel[1], ptr1→vel[2]);

    /* Calculate true dynamical optical depth */

    for (tau = 0.0, i = 0; i < NumParticles; i++)
        tau += PI * Data[i]→radius_sq;

    tau /= SQ(BOX_SIZE);

    (void) printf("[Actual dynamical optical depth = %e]\n", tau);
}
```

```
#undef MAX_NUM_TRIES

static BOOLEAN check_pos(x, y, z, r, n)
double x, y, z, r;
int n;
{
    /*
     * Returns TRUE if a sphere of radius "r" at "x", "y", "z" is
     * unable to fit in central box due to overlap with any of the
     * first "n" particles (and their ghosts) in Data[] (recursive).
     *
     */

    int i, ix, iy;
    DATA_T *ptr;

    /* Check for overlap */

    for (i = 0; i ≤ n; i++) {
        if (i == n && ABS(x) < HALF_BOX_SIZE && ABS(y) < HALF_BOX_SIZE)
            continue;
        ptr = Data[i];
        if (SQ(ptr→pos[0] - x) + SQ(ptr→pos[1] - y) +
                SQ(ptr→pos[2] - z) < SQ(r + ptr→radius))
            return TRUE;
    }

    /* Loop over ghost boxes */

    if (GHOSTS && ABS(x) < HALF_BOX_SIZE && ABS(y) < HALF_BOX_SIZE)
        for (ix = -1; ix ≤ 1; ix++)
            for (iy = -1; iy ≤ 1; iy++) {
                if (ix == 0 && iy == 0)
                    continue;
                if (check_pos(x + ix * BOX_SIZE, y + iy * BOX_SIZE, z, r, n))
                    return TRUE;
            }

    return FALSE;
}

static void pack_box()
{
    /*
     * Packs box so that all particles are touching (assumes
     * equal-sized particles, i.e. min_mass = max_mass).
     *
     */

    int num_layers, num_on_side, ix, iy, iz;
    DATA_T *ptr;
    double half_size, radius, dxy, dz, xypos0, zpos0,
        mass = RunPar.init_min_mass;

    /* Calculate number of particles on a side in (centre) box */

    num_layers = RunPar.num_layers;
    num_on_side = sqrt((double) NumParticles / num_layers);

    /* Calculate half particle separation length */
```

```
half_size = HALF_BOX_SIZE / num_on_side;

/* Assign new density if appropriate (expanded radius = half_size) */

if (RunPar.expand_radii) {

    radius = half_size;

    if (radius < RocheRadius(mass)) {
        (void) sprintf(ErrorStr, "radius = %g", radius);
        Error(WARNING2, "pack_box(): Roche > particle radius.", ErrorStr);
    }

    (void) printf("[close packing: radii = %g]\n", radius);

    /* Assign new particle density */

    RunPar.density = Density(mass, radius);

    (void) printf("[close packing: new density = %g g/cm^3]\n",
        RunPar.density / RunPar.density_conv / DENSITY_CGS_TO_MKS);
}
else
    radius = Radius(mass);

/* Reset initial velocity dispersions if applicable */

if (RunPar.small_disp) {
    SZ = SY = SX = radius;
    (void) printf("[close packing: new vel disp = %g]\n", SZ);
}

/* Output optical depth */

(void) printf("[close packing: dynamical optical depth = %g]\n",
    NumParticles * PI * SQ(radius) / SQ(BOX_SIZE));

/* Get starting position in x & y */

xypos0 = (1 - num_on_side) * half_size;

/* Ensure centre-of-mass remains at origin */

if (RunPar.stagger_in_z)
    xypos0 -= (num_layers / 2) * (half_size / num_layers);

/* Squish in z-direction if appropriate */

dz = (RunPar.expand_radii && RunPar.stagger_in_z ?
    (2 - sqrt(3.0)) * half_size : 0);

/* Get starting position in z */

zpos0 = (1 - num_layers) * half_size + (num_layers / 2) * dz;

/* Assign data */

for (iz = 0; iz < num_layers; iz++) {
    dxy = (RunPar.stagger_in_z ? iz % 2 : 0);
```

```
        for (iy = 0; iy < num_on_side; iy++)
            for (ix = 0; ix < num_on_side; ix++) {
                ptr = Data[ix + (iy + iz * num_on_side) * num_on_side];
                ptr->mass = mass;
                ptr->radius = radius;
                ptr->radius_sq = SQ(radius);
                ptr->inertia = MomentOfInertia(mass, radius);
                ptr->drag_fac = DragFactor(mass);
                ptr->pos[0] = xypos0 + (2 * ix + dxy) * half_size;
                ptr->vel[0] = SX * Gasdev();
                ptr->pos[1] = xypos0 + (2 * iy + dxy) * half_size;
                ptr->vel[1] = SY * Gasdev();
                ptr->pos[2] = zpos0 + iz * (2 * half_size - dz);
                ptr->vel[2] = SZ * Gasdev();                        /* (ignore any z SHM) */
                ZERO(ptr->spin);
            }
    }

    /* Subtract centre-of-mass velocity from system */

    sub_com_vel();
}

#define NUM_FIELDS 15                       /* Number of data items per particle in data file */

static void read_init_cond()
{
    /* Reads initial conditions from RunPar.init_cond_filename */

    int status, i, oi, c;
    FILE *fp;
    DATA_T *ptr;
    double m, r, x, y, z, vx, vy, vz, sx, sy, sz, dum_dbl, avg_density = 0;

    (void) printf("Reading initial conditions from \"%s\"...\n",
            RunPar.init_cond_filename);

    NumParticles = 0;

    /* Try to open file */

    if ((fp = fopen(RunPar.init_cond_filename, "r")) == NULL)
        Error(FATAL_IO, "read_init_cond()", RunPar.init_cond_filename);

    /* Discard header lines */

    (void) printf("[skipping %i line(s) of header]\n", RunPar.num_header_lines);

    while (RunPar.num_header_lines-- > 0)
        if (fgets(Workspace, WORKSPACE_SIZE, fp) == NULL)
            Error(FATAL_IO, "read_init_cond()", RunPar.init_cond_filename);

    /* Reset mass data */

    RunPar.init_min_mass = HUGE_VAL;
    RunPar.init_max_mass = 0;
    RunPar.mass_exponent = 0;                   /* Actual mass function NOT determined... */

    /* Read data */
```

```
while ((status = fscanf(fp, "%i%i%lf%lf%lf%lf%lf%lf%lf%lf%lf%lf%lf%lf%i",
        &i, &oi, &m, &r, &x, &y, &z, &vx, &vy, &vz, &dum_dbl, &sx, &sy, &sz,
        &c)) ≠ EOF) {
    if (status ≠ NUM_FIELDS)
        Error(FATAL_IO, "read_init_cond()", RunPar.init_cond_filename);
    if (i ≠ NumParticles) {
        (void) sprintf(ErrorStr, "index %i reset to %i", i, NumParticles);
        Error(WARNING2, "read_init_cond():  Index mismatch.", ErrorStr);
        i = NumParticles;
    }
    if (i == MAX_NUM_PARTICLES) {
        Error(WARNING1,
            "read_init_cond():  Too many particles -- skipping...", "");
        break;
    }
    ptr = Data[i] = (DATA_T *) malloc(sizeof(DATA_T));
    if (oi < 0 || oi > MAX_NUM_PARTICLES) {
        (void) sprintf(ErrorStr, "par %i org idx %i reset to %i", i, oi, i);
        Error(WARNING2, "read_init_cond():  Orig index out of range.",
            ErrorStr);
        oi = i;
    }
    if (oi ≠ i) {
        (void) sprintf(ErrorStr, "par %i orig index %i", i, oi);
        Error(WARNING2, "read_init_cond():  Orig index not unique?",
            ErrorStr);
    }
    ptr→orig_index = oi;
    if (m ≤ 0) {
        (void) sprintf(ErrorStr, "particle %i (%i) mass %g", i, oi, m);
        Error(FATAL, "read_init_cond():  Invalid mass.", ErrorStr);
    }
    RunPar.init_min_mass = MIN(RunPar.init_min_mass, m);
    RunPar.init_max_mass = MAX(RunPar.init_max_mass, m);
    ptr→mass = m;
    if (r ≤ 0) {
        (void) sprintf(ErrorStr, "particle %i (%i) radius %g", i, oi, r);
        Error(FATAL, "read_init_cond():  Invalid radius.", ErrorStr);
    }
    ptr→radius = r;
    ptr→radius_sq = SQ(r);
    ptr→inertia = MomentOfInertia(m, r);
    ptr→drag_fac = DragFactor(ptr→mass);
    avg_density += Density(m, r);
    ptr→pos[0] = x;
    ptr→pos[1] = y;
    ptr→pos[2] = z;
    ptr→vel[0] = vx + SX * Gasdev();
    ptr→vel[1] = vy + SY * Gasdev();
    ptr→vel[2] = vz + SZ * Gasdev();                              /* (no z SHM) */
    ptr→spin[0] = sx;
    ptr→spin[1] = sy;
    ptr→spin[2] = sz;
    ptr→color = (c == BLACK ? MoviePar.dflt_color : c);
    ++NumParticles;
}  /* while */

(void) fclose(fp);

(void) printf("Done!...%i particle(s) read in.\n", NumParticles);
```

```
    if (NumParticles == 0)
        Error(FATAL, "read_init_cond(): No data found!", "");

    /* Determine average density */

    RunPar.density = avg_density / NumParticles;

    (void) printf("[average particle density = %g g/cm^3]\n",
        RunPar.density / RunPar.density_conv / DENSITY_CGS_TO_MKS);

    /* Check ability to conserve tzam in rotating frame */

    if (ROTATING_FRAME) {
        double com_vel_x = 0;

        for (i = 0; i < NumParticles; i++)
            com_vel_x += Data[i]->mass * Data[i]->vel[0];

        if (!APPROX_EQ(com_vel_x, 0))
            Error(WARNING1,
                "read_init_cond(): com vel != 0 ==> no tzam conserv.", "");
    }
}

#undef NUM_FIELDS

static void sub_com_vel()
{
    /* Removes centre-of-mass velocity from particle velocities */

    int i, k;
    double com_vel[NUM_PHYS_DIM], total_mass = 0;

    ZERO(com_vel);

    for (i = 0; i < NumParticles; i++) {
        for (k = 0; k < NUM_PHYS_DIM; k++)
            com_vel[k] += Data[i]->mass * Data[i]->vel[k];
        total_mass += Data[i]->mass;
    }

    if (total_mass == 0)
        Error(FATAL, "sub_com_vel(): Zero total mass.", "");

    NORM(com_vel, total_mass);

    for (i = 0; i < NumParticles; i++)
        SUB(Data[i]->vel, com_vel, Data[i]->vel);
}

/* init_cond.c */
```

B.1.12 `integrate.c`

This file is at the heart of the `box_tree` code. It is also the largest source file, with just under 2 000 lines of code (including comments). The polynomial initialization routines are found here, along with the integrator itself and routines for collision detection and particle merging. Various time-step functions and the output timer routines are also in this file.

The functions `InitLoOrderPoly()` and `InitHiOrderPoly()` perform low-order (force and first derivative) and high-order (second and third derivative) polynomial initializations on a given particle. When reinitializing a set of particles, the low-order routine must be called first for each particle before the high-order routine is called. This is because the high-order routine requires the force and first derivative of *all* particles to be correct in order to calculate the higher derivatives (cf. Aarseth 1985). The routines assume the positions and velocities of all particles are up to date, so unless these quantities have just been assigned, the function `PredictPosAndVelHiAll()` (cf. `misc.c`) should be called first. As explained in §3.6, polynomial reinitialization takes place after each collision and boundary crossing event. Note that ghost particles are used only in the calculation of the force, not of the derivatives. The contributions from any external potentials must be included, although higher derivatives could perhaps be ignored in some cases by choosing a smaller initial time-step instead. The high order routine is responsible for assigning a new time-step and resetting the previous update times of the particle. It also converts the force derivatives from Taylor series form to divided differences and initializes the start-of-step position and velocity variables.

The `InitTsl()` function is used to initialize the time-step list. The TSL facilitates the search for the next particle to update by keeping track of a subset of the particles that are due for updating in the near future. The TSL update interval is adjusted automatically to stabilize the list membership on $\sim N^{1/2}$. The local functions `make_tsl()`, `sort_tsl()`, `change_pos_on_tsl()`, `remove_from_tsl()`, and `add_to_tsl()` are used to maintain the list. Several of these functions are called by the integrator as well as the collision and merger routines. For example, if two particles are merged it may be necessary to remove the TSL entry of the second particle using `remove_from_tsl()`. Similarly, the new particle may need to be replaced on the list using `change_pos_on_tsl()` if the particle still has a short step after initialization. See the code for further examples.

The function `Integrate()` controls the entire run once the simulation has started. The routine is called only once, by `box_tree()` in `box_tree.c`. The routine consists of an outer infinite `while` loop that is broken only when the run is complete. The loop may also be exited if a fatal error is detected or if a halt is requested. On each pass of the loop, the next particle for updating is determined. This particle is used for all subsequent calculations until the end of the loop. Once a particle is selected, the integration proceeds as outlined in §3.1 for a predictor-evaluator-corrector algorithm, calling global functions to calculate forces, check boundary conditions, etc., as required. In addition to the TSL functions mentioned above, `Integrate()` calls the local functions `process_any_other_events()` to check the output timers, `collision()` to check for a collision, and `stop_run()` in the event of CPU expiry or a `STOP` file halt.

The `set_time_step()` function for assigning particle time-steps is a bit complicated, mostly because of the variety of TSF options. For example, if the "RV only" option is being used and a close neighbour was not found for the current particle, a warning is printed and the `set_max_step()` routine is called to assign a maximum step (which may or may not be defined in the parameter file). If applicable, a check for a "missed collision" (cf. §3.5.5) is made in the time-step routine as well. If the full expression using the forces and derivatives is to be used for calculating the time-step [cf. equation (3.7)], the routine calls either `sts_before()` or `sts_after()`, depending on whether the Taylor series derivatives or the divided differences, respectively, are available. The former is faster, but generally can be used only when reinitializing.

The `process_any_other_events()` function checks the clock against any defined output timers and processes the corresponding events if they are due to occur. An event is due to occur if the current simulation time has exceeded the event time, or if the simulation is due to terminate and the simulation time matches the event time. A special `BOOLEAN` variable local to `integrate.c` (`last_loop`) is used to flag the end of the simulation. A series of nested loops that call the local `BOOLEAN` functions `event()`, `do_event()`, and `do_event_loop()` are required to ensure that all events occur in the correct order. Much use is made of the rough comparison macros defined in `macros.h` to minimize problems introduced by the finite precision of floating point numbers.

The `BOOLEAN` function `collision()` performs all of the tasks outlined in §3.5 to determine whether the current particle and its closest neighbour (if any) have collided. If so, the routine calls `Bounce()` to apply the collision equations. Any adjustments to the conservation variables are also made as appropriate (cf. §4.4). If merging is enabled, the function `merge()` will be called if applicable (cf. §3.5.7). Particle reinitialization, tree updates, and TSL checks are performed in both `collision()` and `merge()`. The collision function returns `TRUE` if a collision occurred, and `FALSE` otherwise.

```
/*
 * integrate.c - DCR 91-06-12
 * ==========================
 * Main N-body integration routines (includes collision detection & merging).
 *
 * Note: Particle "forces" are really accelerations, i.e. forces per unit mass.
 *
 * Global functions: InitLoOrderPoly(), InitHiOrderPoly(), InitTsl(),
 *     Integrate().
 *
 */

/* Include files */

#include "box_tree.h"

/* Additional definitions */

#define BULGE_MASS 1.0                                    /* For GALAXY_FRAME */
#define BULGE_SCALE 0.3

#define STS_BEFORE sts_before                              /* For set_time_step() */
#define STS_AFTER sts_after

/* Local variables */

static BOOLEAN last_loop = FALSE;                  /* TRUE if run is about to terminate */

/* Local functions */

static void
    set_time_step(),
    set_max_step(),
    make_tsl(),
    sort_tsl(),
    change_pos_on_tsl(),
    remove_from_tsl(),
    add_to_tsl(),
    process_any_other_events(),
    merge(),
    stop_run();

static double
```

```
    sts_before(),
    sts_after();

static BOOLEAN
    event(),
    do_event(),
    do_event_loop(),
    collision();

/* End of preamble */

void InitLoOrderPoly(particle)
int particle;
{
    /*
     * Calculates total force and first derivative on "particle" explicitly.
     * Also finds closest particle. If there is no interparticle gravity,
     * only the force due to any external potential is calculated.
     *
     * Note: the high order terms MUST be initialized after calling this
     * routine, but only after low order terms for ALL particles are up to
     * date. Also, particle positions and velocities (i.e. ptr->pos and
     * ptr->vel) must be up to date before calling (e.g. use
     * PredictPosAndVelHiAll()).
     *
     */

    int i;
    DATA_T *ptr = Data[particle];
    double rel_pos[NUM_PHYS_DIM], rel_vel[NUM_PHYS_DIM], r2, rv, w0, w1, w2;

    /* Initialize */

    ZERO(ptr→f);
    ZERO(ptr→f_dot);
    ZERO(ptr→d2);
    ZERO(ptr→d3);

    InitCp(particle);

    /* Loop over other particles */

    for (i = 0; i < NumParticles; i++) {

        if (i == particle)
            continue;

        /* Relative positions and velocities */

        SUB(Data[i]→pos, ptr→pos, rel_pos);
        SUB(Data[i]→vel, ptr→vel, rel_vel);
        r2 = DOT(rel_pos, rel_pos);
        rv = DOT(rel_pos, rel_vel);

        /* Check whether this is a "close" particle */

        if (r2 < EvolPar.cp_zone_sq) {
            if (RunPar.self_grav)
                CheckForCp1(particle, i, CENTRE, Data[i]→pos, r2);
            else if (rv < 0)
            CheckForCp2(particle, i, Data[i]→pos, r2);
      }

      /* Skip next part if no interparticle gravity */

      if (!RunPar.self_grav)
            continue;

      /* Add softening term if desired */

      if (RunPar.use_softening)
            r2 += MAX(ptr→radius_sq, Data[i]→radius_sq);

      /* Useful quantities */

      w0 = 1 / r2;
      w1 = Data[i]→mass * w0 * sqrt(w0);
      w2 = 3 * rv * w0;

      /* Total force (note rel_pos is "backwards") */

      ptr→f[0] += w1 * rel_pos[0];
      ptr→f[1] += w1 * rel_pos[1];
      ptr→f[2] += w1 * rel_pos[2];

      /* First derivative */

      ptr→f_dot[0] += (rel_vel[0] - w2 * rel_pos[0]) * w1;
      ptr→f_dot[1] += (rel_vel[1] - w2 * rel_pos[1]) * w1;
      ptr→f_dot[2] += (rel_vel[2] - w2 * rel_pos[2]) * w1;
}

/* Add contribution of ghost particles (f_dot ignored) */

if (GHOSTS)
      AddGhostForce(particle);

/* Add force and derivative due to external potential as applicable */

if (ROTATING_FRAME) {                                   /* Coriolis and linear term */
      ptr→f[0] += 2 * ptr→vel[1] + 3 * ptr→pos[0];
      ptr→f[1] -= 2 * ptr→vel[0];
      ptr→f[2] -= RunPar.g_factor_sq * ptr→pos[2];

      ptr→f_dot[0] += 2 * ptr→f[1] + 3 * ptr→vel[0];
      ptr→f_dot[1] -= 2 * ptr→f[0];
      ptr→f_dot[2] -= RunPar.g_factor_sq * ptr→vel[2];
}
else if (GALAXY_FRAME) {                          /* Acceleration due to large galaxy */

      /* Origin is bulge of small galaxy; particle 0 is large galaxy */

      int k;
      DATA_T *ptri, *ptr0;
      double mi, m0, ri2, r02, inv_ri2, inv_r02, inv_ri3, inv_r03, ai, a0,
            fi[NUM_PHYS_DIM], f0[NUM_PHYS_DIM];

      ptri = ptr;
      ptr0 = Data[0];
```

```
            mi = BULGE_MASS;
            m0 = Data[0]→mass;

            ri2 = DOT(ptri→pos, ptri→pos);                     /* From bulge to particle */
            r02 = DOT(ptr0→pos, ptr0→pos);                    /* From bulge to big galaxy */

            if (RunPar.use_softening) {
                ri2 += SQ(BULGE_SCALE);
                r02 += SQ(BULGE_SCALE);
            }

            inv_ri2 = 1 / ri2;
            inv_r02 = 1 / r02;

            inv_ri3 = inv_ri2 * sqrt(inv_ri2);
            inv_r03 = inv_r02 * sqrt(inv_r02);

            for (k = 0; k < NUM_PHYS_DIM; k++) {
                fi[k] = - mi * ptri→pos[k] * inv_ri3;
                f0[k] = - m0 * ptr0→pos[k] * inv_r03;
                ptr→f[k] += fi[k] + f0[k];
            }

            ai = DOT(ptri→pos, ptri→vel) * inv_ri2;
            a0 = DOT(ptr0→pos, ptr0→vel) * inv_r02;

            for (k = 0; k < NUM_PHYS_DIM; k++)
                ptr→f_dot[k] += (
                    - mi * ptri→vel[k] * inv_ri3 - 3 * ai * fi[k]
                    - m0 * ptr0→vel[k] * inv_r03 - 3 * a0 * f0[k]
                );
        }  /* GALAXY */

        /* Include gas drag if desired */

        if (RunPar.include_drag) {
            DRAG_COEF_T *ptrd = &RunPar.drag_coef;

            ptr→f[0] -= ptr→drag_fac * ptrd→x * ptr→vel[0];
            ptr→f[1] -= ptr→drag_fac * ptrd→y * ptr→vel[1];
            ptr→f[2] -= ptr→drag_fac * ptrd→z * ptr→vel[2];

            ptr→f_dot[0] -= ptr→drag_fac * ptrd→x * ptr→f[0];
            ptr→f_dot[1] -= ptr→drag_fac * ptrd→y * ptr→f[1];
            ptr→f_dot[2] -= ptr→drag_fac * ptrd→z * ptr→f[2];

            if (ROTATING_FRAME) {
                ptr→f[1] -= ptr→drag_fac * (1.5 * ptr→pos[0] * ptrd→y +
                    ptrd→hdot);
                ptr→f_dot[1] -= 1.5 * ptr→drag_fac * ptrd→y * ptr→vel[0];
            }
        }
    }

    void InitHiOrderPoly(particle)
    int particle;
    {
        /*
         * Initializes second and third force derivative for "particle"
         * and updates data arrays (including time-step).
```

```
 *
 * Note: This routine should only be called when low order terms
 * of ALL other particles are up to date (see InitLoOrderPoly()).
 *
 */

DATA_T *ptr = Data[particle];

int i, k;
double rel_pos[NUM_PHYS_DIM], rel_vel[NUM_PHYS_DIM],
     rel_f[NUM_PHYS_DIM], rel_f_dot[NUM_PHYS_DIM];

/* Working variables... */

double r2, rv, v2, rf, vf, rf_dot, f1dotk, f2dotk, f3dotk, dt, w0, w1,
     w2, w3, w4, w5, w6, w7, w8, w9;

/* Obtain second and third force derivatives */

if (RunPar.self_grav)
     for (i = 0; i < NumParticles; i++) {

          if (i == particle)
               continue;

          /* Relative positions, velocities, and forces */

          SUB(Data[i]→pos, ptr→pos, rel_pos);
          SUB(Data[i]→vel, ptr→vel, rel_vel);
          r2 = DOT(rel_pos, rel_pos);
          rv = DOT(rel_pos, rel_vel);
          v2 = DOT(rel_vel, rel_vel);
          SUB(Data[i]→f, ptr→f, rel_f);
          SUB(Data[i]→f_dot, ptr→f_dot, rel_f_dot);
          rf = DOT(rel_pos, rel_f);
          vf = DOT(rel_vel, rel_f);
          rf_dot = DOT(rel_pos, rel_f_dot);

          if (RunPar.use_softening)
               r2 += MAX(ptr→radius_sq, Data[i]→radius_sq);

          w0 = 1 / r2;
          w1 = Data[i]→mass * w0 * sqrt(w0);
          w2 = rv * w0;
          w3 = w2 * w2;
          w4 = 3 * w2;
          w5 = 6 * w2;
          w6 = 9 * w2;
          w7 = 3 * ((v2 + rf) * w0 + w3);
          w8 = 3 * w7;
          w9 = (9 * vf + 3 * rf_dot) * w0 + w4 * (w7 - 4 * w3);

          /* Second and third force derivatives */

          for (k = 0; k < NUM_PHYS_DIM; k++) {
               f1dotk = rel_vel[k] - w4 * rel_pos[k];
               f2dotk = (rel_f[k] - w5 * f1dotk - w7 * rel_pos[k]) * w1;
               f3dotk = (rel_f_dot[k] - w8 * f1dotk - w9 * rel_pos[k]) * w1 -
                    w6 * f2dotk;
               ptr→d2[k] += f2dotk;
```

```
                ptr->d3[k] += f3dotk;
            }
        }   /* for */

/* External potentials */

if (ROTATING_FRAME) {
        ptr->d2[0] += 2 * ptr->f_dot[1] + 3 * ptr->f[0];
        ptr->d2[1] -= 2 * ptr->f_dot[0];
        ptr->d2[2] -= RunPar.g_factor_sq * ptr->f[2];

        ptr->d3[0] += 2 * ptr->d2[1] + 3 * ptr->f_dot[0];
        ptr->d3[1] -= 2 * ptr->d2[0];
        ptr->d3[2] -= RunPar.g_factor_sq * ptr->f_dot[2];
}
else if (GALAXY_FRAME) {
        DATA_T *ptri, *ptr0;
        double fi[NUM_PHYS_DIM], f0[NUM_PHYS_DIM], fi_dot[NUM_PHYS_DIM],
                f0_dot[NUM_PHYS_DIM], fi_2dot[NUM_PHYS_DIM], f0_2dot[NUM_PHYS_DIM];
        double mi, m0, ri2, r02, inv_ri2, inv_r02, inv_ri3, inv_r03, ai, a0,
                bi, b0, ci, c0;

        ptri = ptr;
        ptr0 = Data[0];

        mi = BULGE_MASS;
        m0 = ptr0->mass;

        ri2 = DOT(ptri->pos, ptri->pos);
        r02 = DOT(ptr0->pos, ptr0->pos);

        if (RunPar.use_softening) {
                ri2 += SQ(BULGE_SCALE);
                r02 += SQ(BULGE_SCALE);
        }

        inv_ri2 = 1 / ri2;
        inv_r02 = 1 / r02;

        inv_ri3 = inv_ri2 * sqrt(inv_ri2);
        inv_r03 = inv_r02 * sqrt(inv_r02);

        for (k = 0; k < NUM_PHYS_DIM; k++) {
                fi[k] = - mi * ptri->pos[k] * inv_ri3;
                f0[k] = - m0 * ptr0->pos[k] * inv_r03;
        }

        ai = DOT(ptri->pos, ptri->vel) * inv_ri2;
        a0 = DOT(ptr0->pos, ptr0->vel) * inv_r02;

        for (k = 0; k < NUM_PHYS_DIM; k++) {
                fi_dot[k] = - mi * ptri->vel[k] * inv_ri3 - 3 * ai * fi[k];
                f0_dot[k] = - m0 * ptr0->vel[k] * inv_r03 - 3 * a0 * f0[k];
        }

        bi = SQ(ai) + (DOT(ptri->vel, ptri->vel) + DOT(ptri->pos, ptri->f)) *
                inv_ri2;
        b0 = SQ(a0) + (DOT(ptr0->vel, ptr0->vel) + DOT(ptr0->pos, ptr0->f)) *
                inv_r02;
        for (k = 0; k < NUM_PHYS_DIM; k++) {
            fi_2dot[k] = - mi * ptri→f[k] * inv_ri3 - 6 * ai * fi_dot[k] -
                3 * bi * fi[k];
            f0_2dot[k] = - m0 * ptr0→f[k] * inv_r03 - 6 * a0 * f0_dot[k] -
                3 * b0 * f0[k];
            ptr→d2[k] += fi_2dot[k] + f0_2dot[k];
        }

        ci = c0 = 0;

        for (k = 0; k < NUM_PHYS_DIM; k++) {
            ci += (3 * ptri→vel[k] * ptri→f[k] +
                ptri→pos[k] * ptri→f_dot[k]) * inv_ri2;
            c0 += (3 * ptr0→vel[k] * ptr0→f[k] +
                ptr0→pos[k] * ptr0→f_dot[k]) * inv_r02;
        }

        ci += ai * (3 * bi - 4 * SQ(ai));
        c0 += a0 * (3 * b0 - 4 * SQ(a0));

        for (k = 0; k < NUM_PHYS_DIM; k++)
            ptr→d3[k] += (
                - mi * ptri→f[k] * inv_ri3 - 9 * ai * fi_2dot[k] -
                9 * bi * fi_dot[k] - 3 * ci * fi[k]
                - m0 * ptr0→f[k] * inv_r03 - 9 * a0 * f0_2dot[k] -
                9 * b0 * f0_dot[k] - 3 * c0 * f0[k]
            );
    }  /* GALAXY */

    /* Add gas drag terms if desired */

    if (RunPar.include_drag) {
        DRAG_COEF_T *ptrd = &RunPar.drag_coef;

        ptr→d2[0] -= ptr→drag_fac * ptrd→x * ptr→f_dot[0];
        ptr→d2[1] -= ptr→drag_fac * ptrd→y * ptr→f_dot[1];
        ptr→d2[2] -= ptr→drag_fac * ptrd→z * ptr→f_dot[2];

        ptr→d3[0] -= ptr→drag_fac * ptrd→x * ptr→d2[0];
        ptr→d3[1] -= ptr→drag_fac * ptrd→y * ptr→d2[1];
        ptr→d3[2] -= ptr→drag_fac * ptrd→z * ptr→d2[2];

        if (ROTATING_FRAME) {
            ptr→d2[1] -= 1.5 * ptr→drag_fac * ptrd→y * ptr→f[0];
            ptr→d3[1] -= 1.5 * ptr→drag_fac * ptrd→y * ptr→f_dot[0];
        }
    }

    /* Initialize time-step, using max step if defined, unity otherwise */

    ptr→time_step = (RunPar.max_time_step ? RunPar.max_time_step : 1);

    /*
     * Set time-step, noting that the force derivatives have not yet been
     * converted to divided differences, so that a faster time-step
     * algorithm (viz. sts_before()) can be used if appropriate.
     *
     */

    set_time_step(particle, STS_BEFORE);
    /* Set last update time to current time */

    ptr→t0 = Clock.time;

    /* Set remaining update times using constant steps before Clock.time */

    dt = ptr→time_step;

    ptr→t1 = Clock.time - dt;
    ptr→t2 = Clock.time - 2 * dt;
    ptr→t3 = Clock.time - 3 * dt;

    /*
     * Convert from Taylor series derivatives to divided differences and
     * initialize primary coordinates and velocities.
     *
     */

    for (k = 0; k < NUM_PHYS_DIM; k++) {
        ptr→d1[k] = (OneSixth * ptr→d3[k] * dt -
            0.5 * ptr→d2[k]) * dt + ptr→f_dot[k];
        ptr→d2[k] = 0.5 * (ptr→d2[k] - ptr→d3[k] * dt);
        ptr→d3[k] *= OneSixth;

        /* Use reduced quantities for fast prediction */

        ptr→f[k] *= 0.5;
        ptr→f_dot[k] *= OneSixth;
    }

    COPY(ptr→pos, ptr→pos0);
    COPY(ptr→vel, ptr→vel0);
}

void InitTsl()
{
    /* Initializes time-step list */

    int i;
    double dt = 0;

    /* Make initial time-step list update interval 1/4 the average step */

    for (i = 0; i < NumParticles; dt += Data[i]→time_step, i++)
        /* (empty) */;

    Tsl.update_interval = 0.25 * dt / NumParticles;

    /* Try to stabilize on root N */

    Tsl.stab = sqrt((double) NumParticles);
    Tsl.stab1 = Tsl.stab2 = 1 / 1.1;                    /* Will be 1 when first used */

    make_tsl();
}

void Integrate()
{
    /*
```

```
 * Core of box_tree: selects next particle on time-step list, sets
 * new time accordingly, and checks for other events (data output,
 * movie frame, etc.) that should be processed first. Integration
 * proceeds by predicting position and velocity of selected particle
 * to high order, obtaining force on particle, setting new differences,
 * adding fourth order correction, checking for collisions and boundary
 * conditions, setting new time-step, and checking for stop conditions.
 * This process is repeated until termination time is reached.
 *
 * This routine is modeled on the fourth-order "Predictor-Evaluator-
 * Corrector" algorithm described by Sverre J. Aarseth (1985) in
 * Brackill J. U., Cohen B. I., eds, Multiple Time Scales, Academic
 * Press, New York, p. 377.
 *
 */

int k, particle;
DATA_T *ptr;
double new_time;
BOOLEAN exit_loop = FALSE;

/* Working variables... */

double dt, t1pr, t2pr, t12pr, dt06, dt19, dt12, dt34, dt32, dt20,
    f0[NUM_PHYS_DIM], f2dotk, f4dotk, dt1, dt2, dt3, t3pr, s2, s3, s4, s5,
    s6, s7, a1, a2, a3, a4, ak4, ak7, ak10;

/* Main loop */

while(TRUE) {

    /*
     * Find next body to be treated and set new time, constructing
     * new time-step list if necessary.
     *
     */

    if (Tsl.index == Tsl.num_on_list)
        make_tsl();

    new_time = Tsl.times[Tsl.index];
    particle = Tsl.list[Tsl.index];

    /* Error checks */

    if (ERROR_CHECK && new_time < Clock.time) {
        (void) sprintf(ErrorStr, "new time %e < clock time %e", new_time,
            Clock.time);
        Error(FATAL, "Integrate():  Backwards step.", ErrorStr);
    }

    if (ERROR_CHECK && !APPROX_EQ(new_time, Data[particle]->t0 +
            Data[particle]->time_step)) {
        (void) sprintf(ErrorStr, "particle %i (%i)", particle,
            Data[particle]->orig_index);
        Error(FATAL, "Integrate():  Corrupted time-step list.", ErrorStr);
    }

    if (ERROR_CHECK && new_time > Clock.tsl_time) {
        (void) sprintf(ErrorStr, "new time %e > tsl time %e", new_time,
            Clock.tsl_time);
        Error(FATAL, "Integrate():  Time-step list out of date.", ErrorStr);
    }

    /* Check for termination */

    if (new_time > RunPar.termination_time) {
        new_time = RunPar.termination_time;
        exit_loop = TRUE;
    }

    /* Update clock */

    Clock.time = new_time;

    /* Check whether anything else should be done first */

    process_any_other_events();

    /* Calculate new ghost box positions if applicable */

    if (ROTATING_FRAME && GHOSTS)
        UpdateBoxPos();

    /* Exit loop if integration completed */

    if (exit_loop) {
        if (VERBOSE)
            (void) printf("*** At termination...exiting Integrate()...\n");
        last_loop = TRUE;
        process_any_other_events();
        SaveRestartData();
        break;
    }

    /* Calculate position and velocity of particle to high order */

    ptr = Data[particle];

    dt = Clock.time - ptr→t0;                              /* (same as ptr->time_step) */

    if (dt ≤ 0) {
        (void) sprintf(ErrorStr, "particle %i (%i), dt = %e", particle,
            ptr→orig_index, dt);
        Error(FATAL, "Integrate():  Invalid time-step.", ErrorStr);
    }

    t1pr = ptr→t0 - ptr→t1;
    t2pr = ptr→t0 - ptr→t2;
    t12pr = t1pr + t2pr;
    dt06 = 0.6 * dt;
    dt19 = OneNinth * dt;
    dt12 = OneTwelfth * dt;
    dt34 = 0.75 * dt;
    dt32 = 1.5 * dt;
    dt20 = dt + dt;

    for (k = 0; k < NUM_PHYS_DIM; k++) {
        f2dotk = ptr→d3[k] * t12pr + ptr→d2[k];
        ptr→pos[k] = ((((ptr→d3[k] * dt06 + f2dotk) * dt12 +
                ptr→f_dot[k]) * dt + ptr→f[k]) * dt +
                ptr→vel0[k]) * dt + ptr→pos0[k];
        ptr→vel[k] = (((ptr→d3[k] * dt34 + f2dotk) * dt19 +
                ptr→f_dot[k]) * dt32 + ptr→f[k]) * dt20 + ptr→vel0[k];
        f0[k] = 2 * ptr→f[k];
    }

    ptr→pos_status = ptr→vel_status = HI_PRED;

    /* Obtain current force and closest particle */

    if (RunPar.use_tree)
        CalcTreeForce(particle);
    else
        CalcDirectForce(particle);

    /* Output data for current particle if desired */

    if (ptr→monitor && TRACK) {
        (void) printf("TRACK %i (%i):  t %.5e r %.1e %.1e %.1e", particle,
            ptr→orig_index, TIME, ptr→pos[0], ptr→pos[1], ptr→pos[2]);
        (void) printf(" pf %.1e %.1e %.1e\n", ptr→f[0], ptr→f[1],
            ptr→f[2]);                          /* ("p" stands for "perturbing force"...) */
        if (ptr→cp.index == CP_UNDEF)
            (void) printf("no closest particle.\n");
        else
            (void) printf("cp %i (%i) box %i r2 %e step %e\n",
                ptr→cp.index, Data[ptr→cp.index]→orig_index,
                ptr→cp.box, ptr→cp.rel_pos_sq, ptr→time_step);
    }

    /* Add force due to external potential as applicable */

    if (ROTATING_FRAME) {                             /* Coriolis force and tidal term */
        ptr→f[0] += 2 * ptr→vel[1] + 3 * ptr→pos[0];
        ptr→f[1] -= 2 * ptr→vel[0];
        ptr→f[2] -= RunPar.g_factor_sq * ptr→pos[2];
    }
    else if (GALAXY_FRAME) {                          /* Acceleration due to large galaxy */
        DATA_T *ptri, *ptr0;
        double mi, m0, ri2, r02, inv_ri2, inv_r02, f_magi, f_mag0;

        ptri = ptr;
        ptr0 = Data[0];

        mi = BULGE_MASS;
        m0 = ptr0→mass;

        PredictPosAndVelHi(ptr0);

        ri2 = DOT(ptri→pos, ptri→pos);
        r02 = DOT(ptr0→pos, ptr0→pos);

        if (RunPar.use_softening) {
            ri2 += SQ(BULGE_SCALE);
            r02 += SQ(BULGE_SCALE);
        }

        inv_ri2 = 1 / ri2;
        inv_r02 = 1 / r02;
        f_magi = - mi * inv_ri2 * sqrt(inv_ri2);
        f_mag0 = - m0 * inv_r02 * sqrt(inv_r02);

        ptr→f[0] += f_magi * ptri→pos[0] + f_mag0 * ptr0→pos[0];
        ptr→f[1] += f_magi * ptri→pos[1] + f_mag0 * ptr0→pos[1];
        ptr→f[2] += f_magi * ptri→pos[2] + f_mag0 * ptr0→pos[2];
}

/* Include gas drag if desired */

if (RunPar.include_drag) {
        DRAG_COEF_T *ptrd = &RunPar.drag_coef;

        ptr→f[0] -= ptr→drag_fac * ptrd→x * ptr→vel[0];
        ptr→f[1] -= ptr→drag_fac * ptrd→y * ptr→vel[1];
        ptr→f[2] -= ptr→drag_fac * ptrd→z * ptr→vel[2];

        if (ROTATING_FRAME)
                ptr→f[1] -= ptr→drag_fac * (1.5 * ptr→pos[0] * ptrd→y +
                        ptrd→hdot);
}

/* Set time intervals for new divided diffs and update times */

dt1 = Clock.time - ptr→t1;
dt2 = Clock.time - ptr→t2;
dt3 = Clock.time - ptr→t3;
t3pr = ptr→t0 - ptr→t3;
s2 = t1pr * t2pr;
s3 = s2 * t3pr;
s4 = s2 + t3pr * t12pr;
s5 = t12pr + t3pr;
s6 = (((TwoThirds * dt + s5) * dt06 + s4) * dt12 + OneSixth * s3) * dt;
s7 = ((0.2 * dt + 0.25 * s5) * dt + OneThird * s4) * dt + 0.5 * s3;
ptr→t3 = ptr→t2;
ptr→t2 = ptr→t1;
ptr→t1 = ptr→t0;
ptr→t0 = Clock.time;
a1 = 1 / dt;
a2 = 1 / dt1;
a3 = 1 / dt2;
a4 = SQ(dt) / dt3;

/* Form new differences and include 4th-order semi-iteration */

for (k = 0; k < NUM_PHYS_DIM; k++) {
        ak4 = (ptr→f[k] - f0[k]) * a1;
        ak7 = (ak4 - ptr→d1[k]) * a2;
        ak10 = (ak7 - ptr→d2[k]) * a3;
        f4dotk = (ak10 - ptr→d3[k]) * a4;
        ptr→d1[k] = ak4;
        ptr→d2[k] = ak7;
        ptr→d3[k] = ak10;
        ptr→pos[k] += f4dotk * s6;
        ptr→vel[k] += f4dotk * s7;
        ptr→f[k] *= 0.5;
        ptr→f_dot[k] = OneSixth * ((ak10 * dt1 + ak7) * dt + ak4);
}
```

```
/*
 * Check for collision or boundary crossing. If there is a
 * collision with closest neighbour, particle updates are
 * handled in collision() (or merge() if appropriate).
 *
 */

if (RunPar.use_softening || !collision(&particle)) {

    /* No collision, so check for boundary crossing */

    int i, bc_event;
    BOOLEAN update_root = FALSE;

    /*
     * Apply b.c.'s if particle now outside central box
     * (or outside tree in case of unbounded system).
     * Otherwise save start-of-step positions and set new
     * time-step for particle.
     *
     */

    bc_event = ApplyBndryCond(particle);

    switch (bc_event) {
        case BC_NONE:
            COPY(ptr→pos, ptr→pos0);
            COPY(ptr→vel, ptr→vel0);
            set_time_step(particle, STS_AFTER);
            break;
        case BC_TREE:
            break;                  /* (init'ns performed in ApplyBndryCond()) */
        case BC_BOX:
            for (i = 0; i < NumParticles; i++)
                if (i ≠ particle)
                    PredictPosAndVelHi(Data[i]);
            InitLoOrderPoly(particle);
            InitHiOrderPoly(particle);
            update_root = TRUE;                           /* (c.f. MoveInTree()) */
            break;
        default:
            (void) sprintf(ErrorStr, "event = %i", bc_event);
            Error(FATAL, "Integrate():  Unknown BC event.", ErrorStr);
    }

    if (bc_event ≠ BC_TREE) {

        /* Move particle in tree if applicable */

        if (ptr→in_tree) {
            if (TreePar.use_move)                                  /* Faster... */
                MoveInTree(particle, update_root);
            else {                                          /* More accurate... */
                RemoveFromTree(particle);
                PlaceInTree(particle, UPDATE, Root);
            }
        }

        /*
         * Put current body back on time-step list if necessary.
```

```
                 * Otherwise increment list index.
                 *
                 */

                if (Clock.time + ptr→time_step < Clock.tsl_time)
                    change_pos_on_tsl(particle);
                else
                    ++Tsl.index;
            }

        }  /* if */

        /* Increment step counter */

        ++Counter[TIME_STEPS];

        /* Check for stop condition */

        if (RunPar.stop_check && Counter[TIME_STEPS] % RunPar.stop_check == 0 &&
                fopen("STOP", "r") ≠ NULL) {
            (void) unlink("STOP");                          /* Remove STOP from directory */
            stop_run("User STOP request detected.");
        }

        /* Also check for CPU time limit expiration */

        if (RunPar.cpu_check && Counter[TIME_STEPS] % RunPar.cpu_check == 0 &&
                TotalCpu() ≥ RunPar.run_time)
            stop_run("CPU time limit exceeded.");

        /* Perform safety dump if required */

        if (RunPar.safety_dump && Counter[TIME_STEPS] % RunPar.safety_dump == 0)
            SaveRestartData();

        /* Finally, stamp log file if required */

        if (Logfile && RunPar.time_stamp &&
                Counter[TIME_STEPS] % RunPar.time_stamp == 0)
            TimeStamp();

    }  /* while */
}

#define MAX_INC_FAC 1.2                    /* Maximum allowed fractional time-step increase */

static void set_time_step(particle, sts_func)
int particle;
double (*sts_func)();
{
    /*
     * Calculates new time-step for "particle" using formula specified
     * in RunPar.tsf_opt. The function "sts_func" should point to one
     * of sts_before() and sts_after(), depending on whether particle
     * force data contains Taylor series derivatives or divided
     * differences, respectively. The function is used for the RV_AND_F
     * and F_ONLY time-step formula options, and is ignored for the
     * RV_ONLY option. Closest particle data is used with RV_ONLY and
     * RV_AND_F, and is ignored for the F_ONLY option.
     *
```

```
 */

int k, opt = RunPar.tsf_opt;
DATA_T *ptr = Data[particle];
CP_T *cp = &(ptr→cp);
double new_step, tiny_step;

if (RunPar.use_softening)                    /* (no collisions, opt must be F_ONLY) */
    new_step = sts_func(ptr);
else {
    double rel_vel[NUM_PHYS_DIM], rel_vel_sq, rdotv, sum_radii_sq;

    rel_vel_sq = rdotv = sum_radii_sq = 0;

   /* Message if no close neighbour found */

    if (opt == RV_ONLY && cp→index == CP_UNDEF) {
        if (ERROR_CHECK) {
            (void) sprintf(ErrorStr, "particle %i (%i) time %g", particle,
                ptr→orig_index, TIME);
            Error(WARNING2, "set_time_step():  No close neighbour.",
                ErrorStr);
        }
        set_max_step(ptr);
        return;
    }

    /* Calculate relative velocity of neighbouring particle */

    if (cp→index ≠ CP_UNDEF)
        for (k = 0; k < NUM_PHYS_DIM; k++)
            rel_vel_sq += SQ(rel_vel[k] = ptr→vel[k] - cp→vel[k]);

    /* Message if relative velocity is zero */

    if (opt == RV_ONLY && rel_vel_sq == 0) {
        if (ERROR_CHECK) {
            (void) sprintf(ErrorStr, "%i (%i) & %i (%i) time %g", particle,
                ptr→orig_index, cp→index, Data[cp→index]→orig_index,
                TIME);
            Error(WARNING2, "set_time_step():  CP has zero rel.  vel.",
                ErrorStr);
        }
        set_max_step(ptr);
        return;
    }

    /* Calculate r dot v */

    if (cp→index ≠ CP_UNDEF)
        for (k = 0; k < NUM_PHYS_DIM; k++)
            rdotv += (ptr→pos[k] - cp→pos[k]) * rel_vel[k];

    sum_radii_sq = SQ(ptr→radius + cp→radius);

    /* Set new time-step, first checking for missed collision */

    if (rdotv < 0 && cp→rel_pos_sq < sum_radii_sq) {
        if (ERROR_CHECK) {
            (void) sprintf(ErrorStr, "%i (%i) & %i (%i) time %e r2 %e",
                particle, ptr→orig_index, cp→index,
                Data[cp→index]→orig_index, TIME, cp→rel_pos_sq);
            Error(WARNING2, "set_time_step(): Missed collision?", ErrorStr);
        }
        new_step = 0.01 * (Data[cp→index]→t0 + Data[cp→index]→time_step
            - Clock.time);                                    /* Magic formula... */
    }
    else {
        if (opt == RV_ONLY || (opt == RV_AND_F && rdotv < 0)) {
            double dum_dbl;

            dum_dbl = cp→rel_pos_sq - RunPar.cp_fac_sq * sum_radii_sq;

            if (ERROR_CHECK && dum_dbl ≤ 0) {
                (void) sprintf(ErrorStr, "%i (%i) & %i (%i) t %e r2 %e",
                    particle, ptr→orig_index, cp→index,
                    Data[cp→index]→orig_index, TIME, cp→rel_pos_sq);
                Error(WARNING2, "set_time_step(): Overlapping particles.",
                    ErrorStr);
                dum_dbl = sum_radii_sq;
            }
            new_step = RunPar.time_step_coef * sqrt(dum_dbl / rel_vel_sq);
        }
        else
            new_step = sts_func(ptr);
    }
} /* if no colliders */

/* Limit time-step increase to factor of MAX_INC_FAC */

new_step = MIN(new_step, MAX_INC_FAC * ptr→time_step);

/* New step must be no smaller than minimum step (if defined) */

if (new_step < RunPar.min_time_step) {
    new_step = RunPar.min_time_step;
    ++Counter[MIN_TIME_STEPS];
}

/* Determine smallest possible step consistent with round-off error */

tiny_step = 1.0e-15 * MAX(1, Clock.time);

/* Message if step smaller than smallest allowable step */

if (new_step < tiny_step) {
    if (ERROR_CHECK) {
        (void) sprintf(ErrorStr, "%i (%i) & %i (%i), dt = %e < %e",
            particle, ptr→orig_index, cp→index,
            Data[cp→index]→orig_index, new_step, tiny_step);
        Error(WARNING2, "set_time_step(): Very short step.", ErrorStr);
    }
    new_step = tiny_step;
}

/* Check whether maximum time-step exceeded */

if (RunPar.max_time_step && new_step > RunPar.max_time_step) {
    new_step = RunPar.max_time_step;
    ++Counter[MAX_TIME_STEPS];
    }

    /* Finally, assign new step */

    ptr→time_step = new_step;
}

static void set_max_step(ptr)
DATA_T *ptr;
{
    /* Assigns maximum time-step to "ptr" (or else a large increase) */

    if (RunPar.self_grav) {
        ptr→time_step *= MAX_INC_FAC;
        if (RunPar.max_time_step && ptr→time_step > RunPar.max_time_step) {
            ptr→time_step = RunPar.max_time_step;
            ++Counter[MAX_TIME_STEPS];
        }
    }
    else if (RunPar.max_time_step) {
        ptr→time_step = RunPar.max_time_step;
        ++Counter[MAX_TIME_STEPS];
    }
    else
        ptr→time_step *= 2;                    /* Large increase if no maximum step */
}

#undef MAX_INC_FAC

static double sts_before(ptr)
DATA_T *ptr;
{
    /* Returns time-step based on Taylor series force derivatives of "ptr" */

    double f, fd, f2d, f3d, fd2, f2d2, dt2;

    f = ABS(ptr→f[0]) + ABS(ptr→f[1]) + ABS(ptr→f[2]);
    fd = ABS(ptr→f_dot[0]) + ABS(ptr→f_dot[1]) + ABS(ptr→f_dot[2]);
    f2d = ABS(ptr→d2[0]) + ABS(ptr→d2[1]) + ABS(ptr→d2[2]);
    f3d = ABS(ptr→d3[0]) + ABS(ptr→d3[1]) + ABS(ptr→d3[2]);
    fd2 = SQ(ptr→f_dot[0]) + SQ(ptr→f_dot[1]) + SQ(ptr→f_dot[2]);
    f2d2 = SQ(ptr→d2[0]) + SQ(ptr→d2[1]) + SQ(ptr→d2[2]);

    if (f2d2 == 0 && (fd == 0 || f3d == 0)) {
        (void) sprintf(ErrorStr, "fd = %e, f3d = %e, f2d2 = %e", fd, f3d, f2d2);
        Error(FATAL, "sts_before():  Zero divisor.", ErrorStr);
    }

    dt2 = RunPar.time_step_coef * (f * f2d + fd2) / (fd * f3d + f2d2);

    return sqrt(dt2);
}

static double sts_after(ptr)
DATA_T *ptr;
{
    /* Returns time-step based on divided differences of "ptr" */

    int k;
    double dt, dt1, s, f_dot[NUM_PHYS_DIM], f_2dot[NUM_PHYS_DIM], f, fd,
          f2d, f3d, fd2, f2d2, dt2;

     dt = ptr→t0 - ptr→t1;
     dt1 = ptr→t0 - ptr→t2;
     s = dt + dt1;

     for (k = 0; k < NUM_PHYS_DIM; k++) {
          f_dot[k] = ptr→d2[k] * dt + ptr→d1[k];
          f_2dot[k] = ptr→d3[k] * s + ptr→d2[k];
     }

     f = 2 * (ABS(ptr→f[0]) + ABS(ptr→f[1]) + ABS(ptr→f[2]));
     fd = ABS(f_dot[0]) + ABS(f_dot[1]) + ABS(f_dot[2]);
     f2d = 2 * (ABS(f_2dot[0]) + ABS(f_2dot[1]) + ABS(f_2dot[2]));
     f3d = 6 * (ABS(ptr→d3[0]) + ABS(ptr→d3[1]) + ABS(ptr→d3[2]));
     fd2 = SQ(f_dot[0]) + SQ(f_dot[1]) + SQ(f_dot[2]);
     f2d2 = 4 * (SQ(f_2dot[0]) + SQ(f_2dot[1]) + SQ(f_2dot[2]));


     if (f2d2 == 0 && (fd == 0 || f3d == 0)) {
          (void) sprintf(ErrorStr, "fd = %e, f3d = %e, f2d2 = %e", fd, f3d, f2d2);
          Error(FATAL, "sts_after():  Zero divisor.", ErrorStr);
     }

     dt2 = RunPar.time_step_coef * (f * f2d + fd2) / (fd * f3d + f2d2);

     return sqrt(dt2);
}

#define MAX_NUM_TSL_LOOPS 100              /* Warning if make_tsl() loops more than this */

static void make_tsl()
{
     /* Constructs time-step list */

     int i, list_index, counter;
     double time;

     Tsl.num_on_list = counter = 0;

     /* Loop until at least one particle is on list */

     while (Tsl.num_on_list == 0) {

          if (counter++ == MAX_NUM_TSL_LOOPS)
               Error(WARNING1, "make_tsl():  Possible infinite loop.", "");

          list_index = 0;
          Clock.tsl_time += Tsl.update_interval;

          /* Add particles whose update times are less than Clock.tsl_time */

          for (i = 0; i < NumParticles; i++)
               if ((time = Data[i]→t0 + Data[i]→time_step) < Clock.tsl_time) {
                    if (list_index == MAX_NUM_ON_TSL) {
                         list_index = -1;
                         break;
                    }
                    Tsl.list[list_index] = i;
                    Tsl.times[list_index++] = time;
```

```
            }

        /* Try again if list filled up, using smaller interval */

        if (list_index == -1) {                                          /* (list filled up) */
            Clock.tsl_time -= Tsl.update_interval;
            Tsl.update_interval *= 0.65;
            continue;
        }

        /* Otherwise save number selected (terminates loop) */

        Tsl.num_on_list = list_index;

        /* Stabilize list membership */

        if (Tsl.short_step) {
            Tsl.stab1 = MAX(0.9 * Tsl.stab1, 0.25);
            Tsl.stab2 = MAX(0.9 * Tsl.stab2, 0.5);
        }
        else {
            Tsl.stab1 = MIN(1.1 * Tsl.stab1, 2.0);
            Tsl.stab2 = MIN(1.1 * Tsl.stab2, 4.0);
        }
        if (list_index > Tsl.stab2 * Tsl.stab)
            Tsl.update_interval *= 0.75;
        else if (list_index < Tsl.stab1 * Tsl.stab)
            Tsl.update_interval *= 1.25;

    } /* while */

    /* Sort list */

    sort_tsl();
}

#undef MAX_NUM_TSL_LOOPS

static void sort_tsl()
{
    /* Sorts time-step list in chronological order */

    Sort2(Tsl.num_on_list, Tsl.times, Tsl.list);

    /* Initialize (point to first particle on list) */

    Tsl.index = 0;
    Tsl.short_step = FALSE;
}

static void change_pos_on_tsl(particle)
int particle;
{
    /*
     * Moves tsl data for given particle to appropriate position in
     * sorted list. Only current Integrate() particle (pointed to by
     * Tsl.index) and later particles may be so moved.
     */

    int i, new_index;
    double time = Data[particle]→t0 + Data[particle]→time_step;
    BOOLEAN particle_found = FALSE;

    /* Remove particle data from lists */

    for (i = Tsl.index; i < Tsl.num_on_list; i++)
        if (Tsl.list[i] == particle) {
            particle_found = TRUE;
            remove_from_tsl(i);
            break;
        }

    if (ERROR_CHECK && !particle_found) {
        (void) sprintf(ErrorStr, "particle %i (%i)", particle,
            Data[i]→orig_index);
        Error(FATAL, "change_pos_on_tsl():  Not on active list.", ErrorStr);
    }

    /* Find new location for particle in time-ordered sequential list */

    Locate(Tsl.times, Tsl.num_on_list, time, &new_index);

    if (ERROR_CHECK && new_index < Tsl.index)
        Error(FATAL, "change_pos_on_tsl():  Corrupted time-step list.", "");

    /* Place particle in new position */

    add_to_tsl(new_index, particle, time);

    /* Set flag so that list update interval will be reduced on next pass */

    Tsl.short_step = TRUE;
}

static void remove_from_tsl(list_index)
int list_index;
{
    /* Removes particle at "list_index" from tsl */

    int i;

    if (ERROR_CHECK) {
        if (list_index < Tsl.index)
            Error(FATAL, "remove_from_tsl():  Entry out of date.", "");
        if (Tsl.num_on_list == 0)
            Error(FATAL, "remove_from_tsl():  No particles on list!", "");
    }

    --Tsl.num_on_list;

    for (i = list_index; i < Tsl.num_on_list; i++) {
        Tsl.list[i] = Tsl.list[i + 1];
        Tsl.times[i] = Tsl.times[i + 1];
    }
}

static void add_to_tsl(list_index, particle, time)
int list_index, particle;
double time;
{
    /* Adds "particle" (update time "time") to tsl at "list_index" */

    int i;

    if (ERROR_CHECK && (time < Clock.time || list_index < Tsl.index))
        Error(FATAL, "add_to_tsl():  Attempt to add outdated entry.", "");

    if (Tsl.num_on_list == MAX_NUM_ON_TSL)
        Error(FATAL, "add_to_tsl():  List overflow.", "");

    for (i = Tsl.num_on_list++; i > list_index; i--) {
        Tsl.list[i] = Tsl.list[i - 1];
        Tsl.times[i] = Tsl.times[i - 1];
    }

    Tsl.list[i] = particle;                                        /* (i == list_index) */
    Tsl.times[i] = time;
}

static void process_any_other_events()
{
    /*
     * Calls output, movie, and/or check routines if their clocks show times
     * before "Clock.time" (or if "last_loop" is TRUE and the clock times
     * equal "Clock.time"). Note that approximate comparison macros are used
     * to minimize rounding anomalies. Only Integrate() should call this
     * function.
     *
     */

    int i;
    double clock_time;

    /* Save current time */

    clock_time = Clock.time;

    /* Loop while any clock is still behind */

    while (do_event())
        for (i = 0; i < NUM_TIMERS; i++)
            while (do_event_loop(i)) {
                Clock.time = Clock.timer[i];
                switch (i) {
                    case OUTPUT:
                        LongOutput();
                        break;
                    case STATS:
                        OutputStats();
                        break;
                    case DAT:
                        OutputDat();
                        break;
                    case EVOL:
                        CalcEvolPar();
                        break;
                    case MOVIE:
                        MakeMovieFrame();
                        break;
                    case CHECK:
```

```
                    if (DebugPar.check_tree) {
                        if (VERBOSE)
                            (void) printf("Tree check, time %g...\n", TIME);
                        CheckTree(Root);
                        if (VERBOSE)
                            (void) printf("No errors detected.\n");
                    }
                    if (DebugPar.check_multipoles)
                        CheckMultipolePrediction();
                    if (DebugPar.check_force)
                        CheckForce();
                    break;
                default:
                    (void) sprintf(ErrorStr, "event = %i", i);
                    Error(FATAL,
                        "process_any_other_events(): Unknown event.",
                        ErrorStr);
            }
            Clock.timer[i] += RunPar.interval[i];
            Clock.time = clock_time;
        }
}

static BOOLEAN event(timer)
int timer;
{
    /* Returns TRUE if event associated with "timer" should happen soon */

    return (RunPar.interval[timer] &&
        (APPROX_LT(Clock.timer[timer], Clock.time) ||
        (last_loop && APPROX_EQ(Clock.timer[timer], Clock.time))));
}


static BOOLEAN do_event()
{
    /* Returns TRUE if there is an unprocessed event */

    int i;

    for (i = 0; i < NUM_TIMERS; i++)
        if (event(i))
            return TRUE;

    return FALSE;
}

static BOOLEAN do_event_loop(timer)
int timer;
{
    /* Returns TRUE if it is time to do event associated with "timer" */

    int i;

    /* Should it happen soon? */

    if (!event(timer))
        return FALSE;

    /* Do any other events take precedence? */
    for (i = 0; i < NUM_TIMERS; i++)
        if (i ≠ timer && event(i) && (i < timer ?
                Clock.timer[i] ≤ Clock.timer[timer] :
                Clock.timer[i] < Clock.timer[timer]))
            return FALSE;

    /* Then do it */

    return TRUE;
}

static BOOLEAN collision(particle1)
int *particle1;
{
    /*
     * Returns TRUE and performs necessary calculations (mergers, time-step
     * updates, etc.) if real particle "*particle1" is due to collide with
     * "cp->index" (box "cp->box"). Note that "*particle1" is assumed to
     * be the current particle in Integrate(), so that prediction is only
     * required for the second particle. The second particle may be a
     * ghost.
     *
     * Note: The ADDRESS of the first particle is passed in case there is a
     * merger and the actual index must be changed.
     *
     */

    int i, k, particle2, box2;
    DATA_T *ptr1 = Data[*particle1], *ptr2;
    CP_T *cp = &(ptr1→cp);

    /* Working variables... */

    double sum_radii_sq, *pos1, *pos2, init_vel1[NUM_PHYS_DIM],
        init_vel2[NUM_PHYS_DIM], init_spin1[NUM_PHYS_DIM],
        init_spin2[NUM_PHYS_DIM], old_gpe = 0, old_pos1[NUM_PHYS_DIM],
        old_pos2[NUM_PHYS_DIM], old_vel1[NUM_PHYS_DIM], old_vel2[NUM_PHYS_DIM],
        old_dist, new_dist, adj_dist, fraction, rel_pos[NUM_PHYS_DIM],
        rel_vel[NUM_PHYS_DIM], r2, v2, rdotv, new_gpe = 0, r, v, net_mass, semi,
        time2;

    /* No collision if no close particle found */

    if (cp→index == CP_UNDEF)
        return FALSE;

    /* Get closest particle info and store as particle2 */

    particle2 = cp→index;
    box2 = cp→box;
    ptr2 = Data[particle2];

    sum_radii_sq = SQ(ptr1→radius + ptr2→radius);

    /* Calculate new separation following update of first particle */

    for (cp→rel_pos_sq = 0.0, k = 0; k < NUM_PHYS_DIM; k++)
        cp→rel_pos_sq += SQ(rel_pos[k] = cp→pos[k] - ptr1→pos[k]);
```

```
/* No collision if particles do not overlap */

if (cp→rel_pos_sq ≥ sum_radii_sq)
    return FALSE;

/* Make sure particles are still approaching each other */

for (rdotv = 0.0, k = 0; k < NUM_PHYS_DIM; k++)
    rdotv += rel_pos[k] * (cp→vel[k] - ptr1→vel[k]);

if (rdotv ≥ 0)
    return FALSE;

/* Predict particle2 to high order and repeat checks */

PredictPosAndVelHi(ptr2);

COPY(ptr2→pos, cp→pos);

if (box2 ≠ CENTRE) {
    ADD_BOX_OFFSET(cp→pos, box2);
    WRAP(cp→pos);
}

COPY(ptr2→vel, cp→vel);
ADD_BOX_SHEAR(cp→vel, box2);

for (cp→rel_pos_sq = 0.0, k = 0; k < NUM_PHYS_DIM; k++)
    cp→rel_pos_sq += SQ(rel_pos[k] = cp→pos[k] - ptr1→pos[k]);

if (cp→rel_pos_sq ≥ sum_radii_sq) {
    (void) sprintf(ErrorStr, "%i (%i) & %i (%i), r2 = %e", *particle1,
        ptr1→orig_index, particle2, ptr2→orig_index, cp→rel_pos_sq);
    Error(WARNING2, "collision(): Near miss?", ErrorStr);
    return FALSE;
}

for (rdotv = 0.0, k = 0; k < NUM_PHYS_DIM; k++)
    rdotv += rel_pos[k] * (cp→vel[k] - ptr1→vel[k]);

if (rdotv ≥ 0) {
    (void) sprintf(ErrorStr, "%i (%i) & %i (%i), rdotv = %e", *particle1,
        ptr1→orig_index, particle2, ptr2→orig_index, rdotv);
    Error(WARNING2, "collision(): Near miss?", ErrorStr);
    return FALSE;
}

/*
 * Predict pos & vel of all other particles to high order NOW for 2 reasons:
 *     1) Need accurate positions for energy check/velocity adjustment.
 *     2) Need to reinitialize colliding particles at the end anyway.
 *
 */

for (i = 0; i < NumParticles; i++)
    if (i ≠ *particle1 && i ≠ particle2)
        PredictPosAndVelHi(Data[i]);

/*
 * Adjust positions of colliders (outward along their relative position
 * vector) so that the particles just touch, saving current GPE and
 * collider positions & velocities first so that tzam/total energy
 * corrections may be made.
 *
 */

COPY(ptr1→pos, old_pos1);
COPY(ptr2→pos, old_pos2);
COPY(ptr1→vel, old_vel1);
COPY(ptr2→vel, old_vel2);

old_dist = sqrt(cp→rel_pos_sq);
new_dist = ptr1→radius + ptr2→radius;

if (ERROR_CHECK || RunPar.conserve_total_energy) {
    if (ROTATING_FRAME)
        old_gpe = - ptr1→mass * ptr2→mass / old_dist;
    else
        old_gpe = Gpe();
}

/* One half overlap distance */

adj_dist = 0.5 * (new_dist / old_dist - 1);

fraction = old_dist / new_dist;

if (ERROR_CHECK && fraction < 0.9) {
    (void) sprintf(ErrorStr, "r / (R1 + R2) = %e", fraction);
    Error(WARNING2, "collision():  Large overlap.", ErrorStr);
}

/* New relative position vector = (1 + 2 adj_dist) times original */

for (cp→rel_pos_sq = rdotv = 0.0, k = 0; k < NUM_PHYS_DIM; k++) {
    ptr1→pos[k] -= adj_dist * rel_pos[k];
    cp→pos[k] += adj_dist * rel_pos[k];
    rel_pos[k] = cp→pos[k] - ptr1→pos[k];
    cp→rel_pos_sq += SQ(rel_pos[k]);
    rdotv += rel_pos[k] * (cp→vel[k] - ptr1→vel[k]);
}

COPY(cp→pos, ptr2→pos);
SUB_BOX_OFFSET(ptr2→pos, box2);

/*
 * Since positions have been adjusted, velocities must be corrected
 * for shear if applicable.
 *
 */

if (ROTATING_FRAME) {
    ptr1→vel[1] += 1.5 * (old_pos1[0] - ptr1→pos[0]);
    ptr2→vel[1] += 1.5 * (old_pos2[0] - ptr2→pos[0]);
    COPY(ptr2→vel, cp→vel);
    ADD_BOX_SHEAR(cp→vel, box2);
    for (rdotv = 0.0, k = 0; k < NUM_PHYS_DIM; k++)
        rdotv += rel_pos[k] * (cp→vel[k] - ptr1→vel[k]);
}
```

```
/* Check rdotv once again */

if (rdotv ≥ 0) {
    (void) sprintf(ErrorStr, "%i (%i) & %i (%i), rdotv = %e", *particle1,
        ptr1→orig_index, particle2, ptr2→orig_index, rdotv);
    Error(FATAL, "collide():  No longer colliding after adj.", ErrorStr);
}

/*
 * Get new GPE if applicable and adjust velocities so that total
 * energy remains constant (rdotv will not change sign). This
 * procedure is only approximate for the rotating frame as central
 * potential and other particles are not taken into account.
 *
 */

if (ERROR_CHECK || RunPar.conserve_total_energy) {
    if (ROTATING_FRAME)
        new_gpe = - ptr1→mass * ptr2→mass / new_dist;
    else
        new_gpe = Gpe();
}

if (RunPar.conserve_total_energy) {
    double inv_r2, vdotr, v1r[NUM_PHYS_DIM], v2r[NUM_PHYS_DIM], alpha, vfac;

    inv_r2 = 1 / cp→rel_pos_sq;

    vdotr = DOT(ptr1→vel, rel_pos) * inv_r2;

    for (k = 0; k < NUM_PHYS_DIM; k++)
        v1r[k] = vdotr * rel_pos[k];

    vdotr = DOT(cp→vel, rel_pos) * inv_r2;

    for (k = 0; k < NUM_PHYS_DIM; k++)
        v2r[k] = vdotr * rel_pos[k];

    alpha = (ptr1→mass * DOT(v1r, v1r) + ptr2→mass * DOT(v2r, v2r));

    vfac = 1 - 2 * (new_gpe - old_gpe) / alpha;

    if (vfac > 1 || vfac < 0.9) {
         if (ERROR_CHECK) {
             (void) sprintf(ErrorStr, "vfac = %f", vfac);
             Error(WARNING2, "collide():  Unable to adjust velocities.", "");
         }
         DebugPar.total_energy_adj -= new_gpe - old_gpe;
    }
    else {
        vfac = sqrt(vfac);

        for (rdotv = 0.0, k = 0; k < NUM_PHYS_DIM; k++) {
            ptr1→vel[k] += (vfac - 1) * v1r[k];
            cp→vel[k] += (vfac - 1) * v2r[k];
        }

        COPY(cp→vel, ptr2→vel);
        SUB_BOX_SHEAR(ptr2→vel, box2);
```

```
            for (rdotv = 0.0, k = 0; k < NUM_PHYS_DIM; k++)
                rdotv += rel_pos[k] * (cp→vel[k] - ptr1→vel[k]);
        }
    }
    else if (ERROR_CHECK)
        DebugPar.total_energy_adj -= new_gpe - old_gpe;

    /* Correct tzam if desired */

    if (ERROR_CHECK) {
        if (ROTATING_FRAME)
            DebugPar.tzam_adj -= ptr1→mass * (old_vel1[1] - ptr1→vel[1] +
                2 * (old_pos1[0] - ptr1→pos[1])) + ptr2→mass * (old_vel2[1] -
                ptr2→vel[1] + 2 * (old_pos2[0] - ptr2→pos[1]));
        else if (INERTIAL_FRAME)
            DebugPar.tzam_adj -= ptr1→mass * (CROSS_Z(ptr1→pos, ptr1→vel) -
                CROSS_Z(old_pos1, old_vel1)) + ptr2→mass * (CROSS_Z(ptr2→pos,
                ptr2→vel) - CROSS_Z(old_pos2, old_vel2));
        if (box2 ≠ CENTRE)
            DebugPar.tzam_adj -= ptr2→mass * BOX_SIZE * ptr2→vel[1];
    }

    /* Collision confirmed...increment counters */

    ++Counter[COLLISIONS];

    if (box2 ≠ CENTRE)
        ++Counter[GHOST_COLLISIONS];

    /*
     * Copy particle positions and velocities to working arrays for
     * easier manipulation. This was not done earlier because collision
     * was not yet confirmed but cp info was being updated.
     *
     */

    pos1 = ptr1→pos;
    pos2 = cp→pos;

    for (v2 = 0.0, k = 0; k < NUM_PHYS_DIM; k++) {
        init_vel1[k] = ptr1→vel[k];
        init_vel2[k] = cp→vel[k];                                  /* Ghost corrected */
        v2 += SQ(init_vel2[k] - init_vel1[k]);
        init_spin1[k] = ptr1→spin[k];
        init_spin2[k] = ptr2→spin[k];
    }

    r2 = cp→rel_pos_sq;

    r = sqrt(r2);                                         /* Distance between particles */
    v = sqrt(v2);                                       /* Magnitude of relative velocity */
    net_mass = ptr1→mass + ptr2→mass;
    semi = 1 / (2 / r - v2 / net_mass);                           /* Infinite for parabola!... */

    /* Output info */

    (void) printf(" %sCOLLISION: %i (%i) & %i (%i), t = %e v = %e\n",
        (box2 ≠ CENTRE ? "GHOST " : ""), *particle1, ptr1→orig_index,
        particle2, ptr2→orig_index, TIME, v);
```

```
/* Update flags for 1st body (2nd body updated later if applicable) */

if (ptr1→last_collider == ptr2→orig_index)
     ++ptr1→num_collisions;
else {
     ptr1→last_collider = ptr2→orig_index;
     ptr1→num_collisions = 1;
     ++Counter[FIRST_TIME_COLLISIONS];
}

/* Merge if small radial velocity (less than 1% escape velocity) */

if (RunPar.allow_mergers && v2 < 0.0002 * net_mass / r) {
     merge("RV < 1% esc vel", particle1, particle2, pos2, init_vel2);
     return TRUE;
}

/* Calculate and display more quantities if desired */

if (VERY_VERBOSE) {
     double x, ecc, peri, apog;

     x = 1 - r / semi;
     ecc = sqrt(SQ(x) + SQ(rdotv) / (net_mass * semi));
     peri = (1 - ecc) * semi;
     apog = (1 + ecc) * semi;

     (void) printf(" a = %g e = %g r_p = %g r_a = %g\n", semi, ecc,
          peri, apog);
}

/*
 * Calculate post-collision velocities. Note that if particle2 is a
 * ghost, the velocity of the REAL particle is adjusted, since it
 * would collide with the ghost of particle1 anyway, presumably at
 * the next time-step. The shear correction to the y-velocity of the
 * second particle is removed later (unless there is a forced merger).
 *
 */

Bounce(ptr1, ptr2, pos1, pos2, init_vel1, init_vel2, init_spin1, init_spin2,
     ptr1→vel, ptr2→vel, ptr1→spin, ptr2→spin);

/* Save non-local viscosity data if desired */

if (!EMPTY_STR(RunPar.nlv_filename)) {
     if (pos1[0] > pos2[0])
          OutputNlvData(ptr1→mass, (pos1[0] - pos2[0]) *
               (ptr1→vel[1] - init_vel1[1]));
     else
          OutputNlvData(ptr2→mass, (pos2[0] - pos1[0]) *
               (ptr2→vel[1] - init_vel2[1]));
}

/* Determine new semi-major axis for pair */

for (v2 = rdotv = 0.0, k = 0; k < NUM_PHYS_DIM; k++) {
     v2 += SQ(rel_vel[k] = ptr2→vel[k] - ptr1→vel[k]);
     rdotv += rel_pos[k] * rel_vel[k];
}
```

```
semi = 1 / (2 / r - v2 / net_mass);

/* Display more info */

if (VERY_VERBOSE)
    (void) printf(" Post-col'n rel vel %g, semi-major axis %g\n",
        sqrt(v2), semi);

/*
 * If particles are still bound (a > 0), force merger if semi-major axis
 * is inside sum of radii or if new pericentre distance is OUTSIDE sum
 * of radii and semi-major axis is less than 5 times sum of particle
 * radii. Skip this section if mergers aren't allowed.
 *
 */

if (RunPar.allow_mergers && semi > 0) {
    double sum_of_radii = ptr1→radius + ptr2→radius;

    if (sum_of_radii > semi) {
        merge("a < r1 + r2", particle1, particle2, pos2, ptr2→vel);
        ++Counter[FORCED_MERGERS];
        return TRUE;
    }

    if (semi < 5 * sum_of_radii) {                  /* Would perturbation check be better? */
        double x, new_peri;

        x = 1 - r / semi;
        new_peri = (1 - sqrt(SQ(x) + SQ(rdotv) / (net_mass * semi))) * semi;

        if (APPROX_GT(new_peri, sum_of_radii)) {
            merge("r1 + r2 < rp", particle1, particle2, pos2, ptr2→vel);
          ++Counter[FORCED_MERGERS];
          return TRUE;
        }
    }
}

/* Since second particle survived, update its counters */

if (ptr2→last_collider == ptr1→orig_index)
    ++ptr2→num_collisions;
else {
    ptr2→last_collider = ptr1→orig_index;
    ptr2→num_collisions = 1;
}

/* If 2nd particle is a ghost, must now remove offset and shear */

if (box2 ≠ CENTRE) {
    if (ERROR_CHECK) {                                          /* Correct tzam */
        double glz, lz;

        glz = ptr2→mass * CROSS_Z(pos2, ptr2→vel);
        SUB_BOX_SHEAR(ptr2→vel, box2);
        lz = ptr2→mass * CROSS_Z(ptr2→pos, ptr2→vel);
        DebugPar.tzam_adj -= lz - glz;
    }
    else
        SUB_BOX_SHEAR(ptr2→vel, box2);
}

/* Check for boundary crossings (both particles!) */

if (ApplyBndryCond(*particle1) == BC_TREE)
    return TRUE;
if (ApplyBndryCond(particle2) == BC_TREE)
    return TRUE;

/* Initialize force polynomials */

InitLoOrderPoly(*particle1);
InitLoOrderPoly( particle2);
InitHiOrderPoly(*particle1);
InitHiOrderPoly( particle2);

/*
 * Both particles must now be removed from tree (if applicable) and
 * replaced in order to fix c-o-m and quadrupole moments of cells.
 *
 */

if (ptr1→in_tree)
    RemoveFromTree(*particle1);
if (ptr2→in_tree)
    RemoveFromTree( particle2);
if (ptr1→in_tree)
    PlaceInTree(*particle1, UPDATE, Root);
if (ptr2→in_tree)
    PlaceInTree( particle2, UPDATE, Root);

/* Must now update tsl data for SECOND particle if necessary */

time2 = ptr2→t0 + ptr2→time_step;

for (i = Tsl.index + 1; i < Tsl.num_on_list; i++)
    if (Tsl.list[i] == particle2) {
        if (time2 < Clock.tsl_time)
            change_pos_on_tsl(particle2);
        else
            remove_from_tsl(i);
        break;
    }

if (i == Tsl.num_on_list && time2 < Clock.tsl_time) {
    for (i = Tsl.index + 1; i < Tsl.num_on_list; i++)
        if (time2 < Tsl.times[i])
            break;
    add_to_tsl(i, particle2, time2);
}

/* Finally, put first particle back on time-step list if necessary */

if (Clock.time + ptr1→time_step < Clock.tsl_time)
    change_pos_on_tsl(*particle1);
else
    ++Tsl.index;
    return TRUE;
}

static void merge(msg, particle1, particle2, pos2, vel2)
char *msg;
int *particle1, particle2;
double *pos2, *vel2;
{
    /*
     * Merges "particle1" and "particle2" and removes "particle2" from all
     * global lists. "pos2" and "vel2" should be the ghost-corrected
     * position and velocity, respecitvely, of the second particle. "msg"
     * should contain a message describing the reason why a merger occured.
     *
     */

    int i, k;
    DATA_T *ptr1 = Data[*particle1], *ptr2 = Data[particle2], *ptr;
    double mass1, mass2, mass, radius, inertia, com_pos[NUM_PHYS_DIM],
        com_vel[NUM_PHYS_DIM], lin_mom[NUM_PHYS_DIM], ang_mom[NUM_PHYS_DIM],
        old_ke = 0, old_gpe = 0;
    BOOLEAN particle2_in_tree;

    /* Properties of new particle */

    mass = (mass1 = ptr1→mass) + (mass2 = ptr2→mass);
    radius = Radius(mass);
    inertia = MomentOfInertia(mass, radius);

    /* Output info */

    (void) printf(" MERGER (%s):  %i (%i) & %i (%i), mass_%i / M = %g\n",
        msg, *particle1, ptr1→orig_index, particle2, ptr2→orig_index,
        *particle1, mass1 / mass);

    /* Increment counter */

    ++Counter[MERGERS];

    /* Remove both particles from tree now if applicable */

    if (ptr1→in_tree)
        RemoveFromTree(*particle1);
    if (ptr2→in_tree)
        RemoveFromTree( particle2);

    /* Get various quantities before merger */

    for (k = 0; k < NUM_PHYS_DIM; k++) {
        com_pos[k] = (mass1 * ptr1→pos[k] + mass2 * pos2[k]) / mass;
        lin_mom[k] = (mass1 * ptr1→vel[k] + mass2 * vel2[k]);
        com_vel[k] = lin_mom[k] / mass;
    }

    GetAngMom(mass1, ptr1→pos, ptr1→vel, ptr1→inertia, ptr1→spin, mass2,
        pos2, vel2, ptr2→inertia, ptr2→spin, ang_mom);

    if (ERROR_CHECK || RunPar.conserve_total_energy) {
        old_ke = 0.5 * (mass1 * DOT(ptr1→vel, ptr1→vel) +
            mass2 * DOT(vel2, vel2) +
            ptr1→inertia * DOT(ptr1→spin, ptr1→spin) +
            ptr2→inertia * DOT(ptr2→spin, ptr2→spin));
        old_gpe = Gpe();
    }

    /* Set position and velocity of first particle to be c-o-m values */

    COPY(com_pos, ptr1→pos);
    COPY(com_vel, ptr1→vel);

    /*
     * Set mass, radius, moment of inertia, drag factor, and spin, assuming
     * new particle becomes spheroid and angular momentum is conserved.
     *
     */

    ptr1→mass = mass;
    ptr1→radius = radius;
    ptr1→radius_sq = SQ(radius);
    ptr1→inertia = inertia;
    ptr1→drag_fac = DragFactor(mass);

    for (k = 0; k < NUM_PHYS_DIM; k++)
        ptr1→spin[k] = ang_mom[k] / inertia;

    /* Output some quantities if desired */

    if (VERBOSE) {
        (void) printf(" New mass %8.2e radius %8.2e ", mass, radius);
        (void) printf("spin x %+8.1e y %+8.1e z %+8.1e\n", ptr1→spin[0],
            ptr1→spin[1], ptr1→spin[2]);
    }

    particle2_in_tree = ptr2→in_tree;                    /* Flag if particle 2 was in tree */

    /* Deallocate memory held by particle2 */

    free((char *) ptr2);

    /* Decrement number of particles */

    --NumParticles;

    /*
     * Shift down Data array to fill in space just vacated and decrement
     * tree node indices (if applicable) that have index > particle2.
     *
     */

    if (*particle1 > particle2)
        --*particle1;

    for (i = particle2; i < NumParticles; i++) {
        ptr = Data[i] = Data[i + 1];
        if (particle2_in_tree && i ≠ *particle1)
            (ptr→node)→child[ptr→node_index].leaf = i;
    }

    /* Also remove particle2 from tsl if necessary and update */
    for (i = Tsl.index + 1; i < Tsl.num_on_list; i++)
        if (Tsl.list[i] == particle2) {
            remove_from_tsl(i);
            break;
        }

    for (i = Tsl.index; i < Tsl.num_on_list; i++)
        if (Tsl.list[i] > particle2)
            --Tsl.list[i];

    /* Energy adjustment */

    if (ERROR_CHECK || RunPar.conserve_total_energy) {
        double new_ke;

        new_ke = 0.5 * (DOT(lin_mom, lin_mom) / mass +
            DOT(ang_mom, ang_mom) / inertia);

        DebugPar.collision_dke += new_ke - old_ke;
        DebugPar.total_energy_adj -= (Gpe() - old_gpe) + (new_ke - old_ke);
    }

    /* Check boundary conditions for merged body */

    if (ApplyBndryCond(*particle1) == BC_TREE) {
        --Tsl.index;
        return;
    }

    /* Now set new force polynomial and time-step for merged body */

    InitLoOrderPoly(*particle1);
    InitHiOrderPoly(*particle1);

    /* Put merged particle in place of first particle in tree */

    if (ptr1→in_tree && particle2_in_tree)                  /* Both must have been in tree */
        PlaceInTree(*particle1, UPDATE, Root);

    /* Finally, put particle back on TSL if necessary */

    if (Clock.time + ptr1→time_step < Clock.tsl_time)
        change_pos_on_tsl(*particle1);
    else
        ++Tsl.index;
}

static void stop_run(msg)
char *msg;
{
    /* Temporarily halts integration */

    LongOutput();

    SaveRestartData();

    (void) printf("Restart data saved in \"%s\".\n", SaveFilename);

    Error(HALT, msg, "");
}
```

/ integrate.c */*

B.1.13 `make_tree.c`

The main routines for creating the tree and inserting new particles are in this file. There are three global functions: `MakeTree()`, which constructs a tree starting at `Root` given the size and geometric centre; `PlaceInTree()`, which inserts a given particle into a given node; and `GetIndex()`, which returns the index of the subnode containing a given particle in a given node. There are also three local functions: `make_new_node()`, used in conjunction with `MakeTree()` and `PlaceInTree()` for creating a new tree node; `pack()`, to initiate node packing (cf. §3.4.4); and `copy_info()`, to copy node information when turning a leaf into a branch. The comments in the code are fairly extensive, so the reader is referred to the source for further information.

```
/*
 * make_tree.c - DCR 91-04-25
 * ==========================
 * Routines for constructing tree.
 *
 * Global functions: MakeTree(), PlaceInTree(), GetIndex().
 *
 */

/* Include files */

#include "box_tree.h"

/* Local functions */

static NODE_T *make_new_node();
static void pack(), copy_info();

/* End of preamble */

void MakeTree(root_size, root_centre)
double root_size, *root_centre;
{
    /*
     * A call to this routine will build a particle tree, size "root_size",
     * centre "root_centre". Top tree node is returned as Root.
     *
     */

    int i;

    if (ERROR_CHECK && NumParticles == 0)
        Error(FATAL, "MakeTree(): No particles to place in tree!", "");

    /* Allocate memory for root node (also reset children data) */

    Root = make_new_node();

    /* Initialize root node */

    Root→parent = NULL;
    Root→tree_index = 0;
    Root→size = root_size;
    Root→half_size = 0.5 * root_size;

    COPY(root_centre, Root→centre);

    /* Place all requested particles in tree */

    for (i = 0; i < NumParticles; i++)
        if (Data[i]→in_tree)
            PlaceInTree(i, NO_UPDATE, Root);

    /* Fast moment calculation */

    CalcTreeMoments(Root);

    /* Check tree size and cumulative total of leaves */

    if (ERROR_CHECK) {
        if (Root→size ≠ TREE_SIZE) {
            (void) sprintf(ErrorStr, "Root->size = %g", Root→size);
            Error(FATAL, "MakeTree():  Inconsistent tree size.", ErrorStr);
        }
        if (Root→num_leaves ≠ NumParticles - TreePar.num_excluded) {
            (void) sprintf(ErrorStr, "Root->num_leaves = %i", Root→num_leaves);
            Error(FATAL, "MakeTree():  Missing leaves!", ErrorStr);
        }
    }
}

#define FORCE_UPDATE(flag) ((flag) == NO_UPDATE ? NO_UPDATE : UPDATE)

void PlaceInTree(particle, update, node)
int particle, update;
NODE_T *node;
{
    /*
     * Places "particle" in tree at position "Data[particle]->pos", starting
     * at branch "node". Branch moment updating is controlled through
     * "update": if NO_UPDATE, no updating is performed whatsoever (useful
     * for quick tree construction - see MakeTree()); if set to
     * UPDATE_CHILDREN, only children of "node" (on first call) are updated
     * (used when particle is moving completely within node - see
     * MoveInTree()); if UPDATE, the original node and all its affected
     * children are updated (see, for example, merge() in integrate.c).
     *
     */

    int child_index;
    CHILD_T *child;
    NODE_T *new_node;

    /* Determine subnode (child) to which particle belongs at this node */

    child_index = GetIndex(particle, Data[particle]→pos, node);

    if (ERROR_CHECK && child_index == -1) {
        (void) sprintf(ErrorStr, "particle %i (%i) %s", particle,
            Data[particle]→orig_index, NodeInfo(node));
        Error(FATAL, "PlaceInTree():  Particle outside its node.", ErrorStr);
    }
```

```
/* Assign pointer to child cell */

child = &node→child[child_index];

/* Decide what needs to be done to add particle to tree */

switch (node→child_type[child_index]) {

    case EMPTY:

        /* Child node is empty, so make it a leaf */

        node→child_type[child_index] = LEAF;
        child→leaf = particle;

        Data[particle]→node = node;
        Data[particle]→node_index = child_index;

        break;

    case BRANCH:

        /*
         * Child is a branch node, so move down hierarchy. Note that
         * moment updating is forced unless no updating at all has
         * been requested.
         *
         */

        PlaceInTree(particle, FORCE_UPDATE(update), child→branch);

        break;

    case LEAF:

        /*
         * Child is a leaf node already, so make a branch. First check,
         * however, if maximum node level will be exceeded (or if node
         * already packed), in which case particle should be packed in
         * CURRENT node. Updating is forced if packed particle is new
         * to this node, but should be off otherwise.
         *
         */

        if (node→packed || TreeLevel(node) == MAX_TREE_LEVEL) {
            if (!node→packed && MONITOR && VERY_VERBOSE) {
                (void) sprintf(ErrorStr, "%i (%i) time %g (%s)",
                    particle, Data[particle]→orig_index, TIME,
                    NodeInfo(node));
                Error(WARNING2, "PlaceInTree():  Node packing invoked.",
                    ErrorStr);
            }
            pack(node, particle);
            break;
        }

        new_node = make_new_node();

        /*
         * Copy branch information and place other particle in new
```

```
             * branch, using start-of-step position in case particle
             * has actually left node at current time.
             *
             */

            copy_info(node, new_node, child_index, update, child→leaf);

            /* Update child data for this node */

            node→child_type[child_index] = BRANCH;
            child→branch = new_node;

            /* Place original particle in new node */

            PlaceInTree(particle, FORCE_UPDATE(update), new_node);

            break;

        default:

            (void) sprintf(ErrorStr, "type = %i\n",
                node→child_type[child_index]);
            Error(FATAL, "PlaceInTree(): Unknown node type.", ErrorStr);

    } /* switch */

    /* Finally, update branch moments if desired for this node */

    if (update == UPDATE)
        UpdateBranchMoments(ADDM, particle, node);
}

#undef FORCE_UPDATE

int GetIndex(particle, pos, node)
int particle;
double *pos;
NODE_T *node;
{
    /*
     * Returns index of particle "particle" (pos'n "pos") in node "node".
     * Halts with error message if particle outside tree, but returns index
     * -1 if particle only outside node (used in MoveInTree()). Note
     * particle index actually used only in error message.
     *
     * In 2D, the node indexing is as follows (looking down on the xy-plane):
     *
     *     +—+—+
     *     | 2 | 3 |
     *     +—+—+
     *     | 0 | 1 |
     *     +—+—+
     *
     * In 3D, the bottom layer is as for 2D, with the top layer labeled
     * from 4 to 7 in an analogous manner.
     *
     * Note: problems can arise if PRECISION is not a sensible value...
     *
     */
    int k, index = 0;
    double rel_pos_k;

    for (k = 0; k < NUM_TREE_DIM; k++) {
        rel_pos_k = pos[k] - node→centre[k];
        if (APPROX_GT(ABS(rel_pos_k), node→half_size)) {
            if (ABS(pos[k] - Root→centre[k]) > Root→half_size) {
                (void) sprintf(ErrorStr, "%i (%i) x %g y %g z %g t %g",
                    particle, Data[particle]→orig_index, pos[0], pos[1],
                    (NUM_TREE_DIM == 3 ? pos[2] : 0), TIME);
                Error(FATAL, "GetIndex(): Particle outside tree.", ErrorStr);
            }
            else
                return -1;
        }
        if (rel_pos_k > 0)
            index += ChildIndexOffset[k];
    }

    return index;
}

static NODE_T *make_new_node()
{
    /* Allocates memory for one node, returning pointer to it */

    int i, k;
    NODE_T *new_node;

    /* Allocate memory */

    if ((new_node = (NODE_T *) malloc(sizeof(NODE_T))) == NULL)
        Error(FATAL, "make_new_node(): Unable to create new node.", "");

    /* Initialize EVERYTHING for safety */

    new_node→tree_index = new_node→node_index = -1;
    new_node→parent = NULL;

    new_node→size = new_node→half_size = new_node→max_ext =
        new_node→max_size = new_node→max_size_sq = new_node→mass = 0;

    for (k = 0; k < NUM_TREE_DIM; k++)
        new_node→centre[k] = 0;

    for (i = 0; i < MAX_NUM_CHILDREN; i++) {
        new_node→child_type[i] = EMPTY;
        new_node→child[i].leaf = -1;
    }

    new_node→num_leaves = 0;

    for (k = 0; k < NUM_PHYS_DIM; k++)
        new_node→pos0[k] = new_node→pos[k] = new_node→vel[k] =
            new_node→f[k] = new_node→f_dot[k] = 0;

    for (i = 0; i < NUM_QUAD_ELEM; i++)
        new_node→q_mom0[i] = new_node→q_mom[i] = new_node→q_dot[i] =
            new_node→q_2dot[i] = new_node→q_3dot[i] = 0;
    new_node→mt0 = new_node→qt0 = 0;
    new_node→mts = new_node→qts = HUGE_VAL;

    new_node→extended = new_node→packed = FALSE;

    /* Return pointer to new node */

    return new_node;
}

static void pack(node, particle)
NODE_T *node;
int particle;
{
    /*
     * Places "particle" in first available cell in "node", regardless of
     * particle position. Intended for use in (rare) situations when
     * required node level exceeds MAX_NODE_LEVEL (for example, when x and
     * y positions of 3D particle in 2D tree overlap to some extent.)
     *
     */

    int i;

    /* Set flag (stays on until node is destroyed) */

    node→packed = TRUE;

    /* Find first available cell */

    for (i = 0; i < MAX_NUM_CHILDREN; i++) {
        if (node→child_type[i] == EMPTY)
            break;
        if (ERROR_CHECK && node→child_type[i] == BRANCH) {
            (void) sprintf(ErrorStr, "%s", NodeInfo(node));
            Error(FATAL, "pack():  Found branch in node.", ErrorStr);
        }
    }

    /* Crash if node is full (hopefully this will never happen!) */

    if (i == MAX_NUM_CHILDREN) {
        (void) sprintf(ErrorStr, "%s", NodeInfo(node));
        Error(FATAL, "pack():  Node is full!", ErrorStr);
    }

    /* Place particle in cell */

    node→child_type[i] = LEAF;
    node→child[i].leaf = particle;

    /* Update particle data */

    Data[particle]→node = node;
    Data[particle]→node_index = i;

    /* Increment counter */

    ++Counter[PACKINGS];
}
```

```
static void copy_info(old_node, new_node, index, update, particle)
NODE_T *old_node, *new_node;
int index, update, particle;
{
    /*
     * Makes "new_node" a child branch of "old_node" at "index", and places
     * particle "particle" (start-of-step position "Data[particle]->pos0")
     * inside it. Updating is controlled through "update" (see PlaceInTree()).
     *
     */

    int k, particle_index;

    /* Copy/extrapolate information from parent node */

    new_node→parent = old_node;
    new_node→node_index = index;
    new_node→tree_index = old_node→tree_index * MAX_NUM_CHILDREN + index + 1;
    new_node→half_size = 0.5 * (new_node→size = old_node→half_size);

    /* Calculate centre of new node */

    for (k = 0; k < NUM_TREE_DIM; k++)
        new_node→centre[k] = old_node→centre[k] +
            ChildCoordOffset[k][index] * new_node→half_size;

    /* Set node and node index for particle in Data array */

    particle_index = GetIndex(particle, Data[particle]→pos0, new_node);

    if (ERROR_CHECK && particle_index == -1) {
        (void) sprintf(ErrorStr, "particle %i (%i) %s", particle,
            Data[particle]→orig_index, NodeInfo(new_node));
        Error(FATAL, "copy_info():  Particle outside its node.", ErrorStr);
    }

    Data[particle]→node = new_node;
    Data[particle]→node_index = particle_index;

    /* Place particle information in child leaf */

    new_node→child_type[particle_index] = LEAF;
    new_node→child[particle_index].leaf = particle;

    /*
     * Set leaf count and mass of new node if updating is requested. Note
     * that SECOND particle should be added before last return from
     * PlaceInTree().
     *
     */

    if (update ≠ NO_UPDATE) {
        new_node→num_leaves = 1;
        new_node→mass = Data[particle]→mass;
    }
}

/* make_tree.c */
```

B.1.14 misc.c

This file contains 30 global routines ranging from simple functions (e.g. `Radius()`) to more involved procedures (e.g. `SaveRestartData()`). There are a few system calling routines, such as a function to return the current date, so a number of system header files are included at the beginning. There are two local functions, `save_tree_data()` and `read_tree_data()` which are recursive procedures for use with `SaveRestartData()` and `ReadRestartData()`. The reader is referred to the source for descriptions of all the routines. Functions to note are `InitMassFunc()` for calculating equation (3.1), `Gpe()` for determining the gravitational potential energy, `InitCP()` for initializing closest-particle data, `Error()` for error handling, and `Terminate()` to cleanly exit `box_tree`. There are also various routines for finding closest particles (depending on whether interparticle gravity is included in the simulation) and for predicting particle and node positions and velocities to high order (the low order procedures are coded as in-line macros in `macros.h`). The routines for measuring the elapsed CPU are also found in this file.

```
/*
 * misc.c - DCR 91-05-03
 * =======================
 * Miscellaneous useful routines for box_tree code.
 *
 * Global functions: GetDate(), GetHost(), InitMassFunc(), EstMeanMass(),
 *     Radius(), Mass(), Density(), RocheRadius(), MomentOfInertia(),
 *     DragFactor(), Median(), Gpe(), Boolean(), MakeFilename(), CurrentIndex(),
 *     InitCp(), CheckForCp1(), CheckForCp2(), CheckForCp3(),
 *     PredictPosAndVelHi(), PredictPosAndVelHiAll(), PredictPosAndQMomAll(),
 *     ElapsedCpu(), TotalCpu(), TimeStamp(), BackupFile(), SaveRestartData(),
 *     ReadRestartData(), Error(), Terminate().
 *
 */

/* Include files */

#include "box_tree.h"
#ifndef SYSV
# include <sys/errno.h>                              /* For perror() */
# include <sys/param.h>                              /* For GetHost() */
# include <sys/time.h>                  /* For GetDate() & ElapsedCpu() */
# include <sys/resource.h>                /* Must be AFTER <sys/time.h> */
#endif

/* Local functions */

static void save_tree_data(), read_tree_data();

char *GetDate()
{
#ifndef SYSV
    /* Returns current date and time */

    static char date[MAX_STR_LEN];

    struct timeval tp;
    struct timezone tzp;

    /* Get seconds elapsed since 00:00 GMT, January 1, 1970 (zero hour) */

    if (gettimeofday(&tp, &tzp))
```

```
        return "unknown date";

    /* Convert to more conventional format, and strip off \n at end */

    (void) strcpy(date, asctime(localtime(&tp.tv_sec)));
    date[strlen(date) - 1] = '\0';

    return date;
#else
    return "unknown date";
#endif
}

char *GetHost()
{
#ifndef SYSV
    /* Returns host machine name if available */

    static char hostname[MAXHOSTNAMELEN];                  /* Defined in <sys/param.h> */

    if (gethostname(hostname, MAXHOSTNAMELEN))
        return "unknown host";

    return hostname;
#else
    return "unknown host";
#endif
}

#define M0 RunPar.init_min_mass                                  /* Useful abbreviations */
#define M1 RunPar.init_max_mass
#define GAMMA RunPar.mass_exponent

double InitMassFunc(frac)
double frac;
{
    /* Returns mass corresponding to fraction "frac" in mass function */

    double g1 = 1 + GAMMA;

    return M0 * pow(1 + frac * (pow(M1 / M0, g1) - 1), 1 / g1);
}

double EstMeanMass()
{
    /* Returns estimate of mean mass given mass function */

    double mm;

    if (M0 == M1)
        mm = M0;
    else if (GAMMA == -2)
        mm = (M0 * M1 / (M1 - M0)) * log(M1 / M0);
    else {
        double g1 = 1 + GAMMA, g2 = 2 + GAMMA;

        mm = (g1 / g2) * M0 * (pow(M1 / M0, g2) - 1) / (pow(M1 / M0, g1) - 1);
    }

    if (RunPar.seed_mass)
```

```
        mm = (RunPar.seed_mass + (NumParticles - 1) * mm) / NumParticles;

    return mm;
}

#undef M0
#undef M1
#undef GAMMA

double Radius(mass)
double mass;
{
    /* Returns radius corresponding to given spherical mass */

    return pow(mass / (FourThirds * PI * RunPar.density), OneThird);
}

double Mass(radius)
double radius;
{
    /* Returns inverse of Radius() */

    return FourThirds * PI * CUBE(radius) * RunPar.density;
}

double Density(mass, radius)
double mass, radius;
{
    /* Returns density of sphere */

    return mass / (FourThirds * PI * CUBE(radius));
}

double RocheRadius(mass)
double mass;
{
    /* Returns Roche radius for spherical mass */

    return pow(OneThird * mass, OneThird);
}

double MomentOfInertia(mass, radius)
double mass, radius;
{
    /* Returns moment of inertia of uniform solid sphere about any diameter */

    return 0.4 * mass * SQ(radius);
}

double DragFactor(mass)
double mass;
{
    /* Returns drag factor for uniform solid spherical mass */

    return pow(mass, - OneThird);
}

double Median(n, x)
int n;
double *x;
```

```
{
    /* Returns median of first n elements of double precision array x */

    int nm = n / 2;

    Sort(n, x);

    return (n % 2 ? x[nm] : 0.5 * (x[nm - 1] + x[nm]));
}

double Gpe()
{
    /*
     * Returns gravitational potential energy of system (central particles
     * only). Particle positions are assumed to be predicted to correct
     * order. Currently only INERTIAL_FRAME is supported.
     *
     */

    int i, j;
    DATA_T *ptri, *ptrj;
    double gpe, gpei, dx, dy, dz, r2;

    if (!RunPar.self_grav || !INERTIAL_FRAME)
        return 0.0;

    gpe = 0;

    for (i = 0; i < NumParticles - 1; i++) {
        ptri = Data[i];
        gpei = 0;
        for (j = i + 1; j < NumParticles; j++) {
            ptrj = Data[j];
            dx = ptrj→pos[0] - ptri→pos[0];
            dy = ptrj→pos[1] - ptri→pos[1];
            dz = ptrj→pos[2] - ptri→pos[2];
            r2 = SQ(dx) + SQ(dy) + SQ(dz);
            if (RunPar.use_softening)
                r2 += MAX(ptri→radius_sq, ptrj→radius_sq);
            gpei -= ptrj→mass / sqrt(r2);
        }
        gpe += ptri→mass * gpei;
    }

    return gpe;
}

char *Boolean(value)
BOOLEAN value;
{
    /* Returns address of string label corresponding to Boolean "value" */

    if (value == FALSE)
        return "FALSE";

    if (value == TRUE)
        return "TRUE";

    return "INVALID";
}
```

```
char *MakeFilename(basename, filenumber, extension)
char *basename, *extension;
int filenumber;
{
    /*
     * Returns address of character string formed by concatenation of
     * "basename", "filenumber", and "extension". Note filename is held in
     * static memory, so this routine can only be used to form one string
     * at a time. Also, "extension" should include prefix "." if desired.
     * If "filenumber" exceeds maximum number of allowed digits, NULL is
     * returned in place of string.
     *
     */

    static char filename[MAX_FILENAME_LEN];                    /* Storage */

    int i, j, num_digits;
    char filenumber_str[MAX_NUM_FILENUM_DIGITS + 1];

    if (filenumber < 0) {
        (void) sprintf(ErrorStr, "filenumber = %i", filenumber);
        Error(FATAL, "MakeFilename(): Negative filenumber passed.", ErrorStr);
    }

    if (filenumber ≥ EXP10(MAX_NUM_FILENUM_DIGITS)) {
        Error(WARNING2, "MakeFilename(): Filenumber exceeds maximum.", "");
        return NULL;
    }

    num_digits = (filenumber == 0 ? 1 : log10((double) filenumber) + 1);

    (void) sprintf(filenumber_str, "%i", filenumber);

    for (i = 0; i < MAX_NUM_FILENUM_DIGITS - num_digits; i++) {
        for (j = MAX_NUM_FILENUM_DIGITS; j > i; j--)
            filenumber_str[j] = filenumber_str[j - 1];
        filenumber_str[i] = '0';
    }

    (void) sprintf(filename, "%s%s%s", basename, filenumber_str, extension);

    return filename;
}

int CurrentIndex(orig_index)
int orig_index;
{
    /*
     * Returns current index of particle with original index "orig_index".
     * Returns -1 if particle not found (e.g. if particle has merged with
     * another). Note that if particles can only be removed, then search
     * could begin at orig_index and proceed DOWNWARDS until found...
     *
     */

    int i;

    for (i = 0; i < NumParticles; i++)
        if (Data[i]→orig_index == orig_index)
```

```
            return i;

    return -1;
}

void InitCp(particle)
int particle;
{
    /* Initializes closest-particle structure for "particle" */

    CP_T *cp = &(Data[particle]->cp);

    cp->index = cp->box = CP_UNDEF;
    cp->radius = 0;
    cp->rel_pos_sq = HUGE_VAL;

    ZERO(cp->pos);
    ZERO(cp->vel);
}

void CheckForCp1(particle0, particle, box, pos, r2)
int particle0, particle, box;
double *pos, r2;
{
    /*
     * Checks if "particle" ("box", "pos", square distance "r2") is closest
     * particle to "particle0", and updates cp info if so. This routine
     * should be used if interparticle gravity is switched on.
     *
     */

    CP_T *cp = &(Data[particle0]->cp);

    if (r2 < cp->rel_pos_sq) {
        DATA_T *ptr = Data[particle];

        cp->index = particle;
        cp->box = box;
        cp->radius = ptr->radius;
        cp->rel_pos_sq = r2;
        PREDICT_VEL_LO(ptr);
        COPY(pos, cp->pos);
        COPY(ptr->vel, cp->vel);
        ADD_BOX_SHEAR(cp->vel, box);
    }
}

void CheckForCp2(particle0, particle, pos, r2)
int particle0, particle;
double *pos, r2;
{
    /*
     * Same as CheckForCp1() but for case of no interparticle gravity
     * and approaching (central) particles (rdotv < 0). Intended for use
     * with InitLoOrderPoly() as a faster version of CheckForCp3().
     *
     */

    CP_T *cp = &(Data[particle0]->cp);
    if (r2 < cp→rel_pos_sq) {
        DATA_T *ptr = Data[particle];

        cp→index = particle;
        cp→box = CENTRE;
        cp→radius = ptr→radius;
        cp→rel_pos_sq = r2;
        COPY(pos, cp→pos);
        COPY(ptr→vel, cp→vel);
    }
}

void CheckForCp3(particle0, particle, box, pos, r2, rel_pos)
int particle0, particle, box;
double *pos, r2, *rel_pos;
{
    /*
     * Same as CheckForCp1() but for no interparticle gravity. Note that
     * the relative position ("rel_pos") should be passed to this routine,
     * for efficient calculation of rdotv.
     *
     */

    DATA_T *ptr0 = Data[particle0], *ptr = Data[particle];
    CP_T *cp = &(ptr0→cp);
    double vel[NUM_PHYS_DIM], rdotv;

    PREDICT_VEL_LO(ptr);
    COPY(ptr→vel, vel);
    ADD_BOX_SHEAR(vel, box);

    rdotv = rel_pos[0] * (ptr0→vel[0] - vel[0]) +
        rel_pos[1] * (ptr0→vel[1] - vel[1]) +
        rel_pos[2] * (ptr0→vel[2] - vel[2]);

    if (rdotv < 0 && r2 < cp→rel_pos_sq) {
        cp→index = particle;
        cp→box = box;
        cp→radius = ptr→radius;
        cp→rel_pos_sq = r2;
        COPY(pos, cp→pos);
        COPY(vel, cp→vel);
    }
}

void PredictPosAndVelHi(ptr)
DATA_T *ptr;
{
    /* Predicts pos. & vel. of particle pointed to by "ptr" to high order */

    int k;
    double f2dotk, dt1, dt;

    if ((ptr→pos_status == NO_PRED && ptr→vel_status == NO_PRED) ||
            (ptr→pos_status == HI_PRED && ptr→vel_status == HI_PRED))
        return;

    dt1 = (ptr→t0 - ptr→t1) + (ptr→t0 - ptr→t2);
    dt = Clock.time - ptr→t0;
    for (k = 0; k < NUM_PHYS_DIM; k++) {
        f2dotk = ptr→d3[k] * dt1 + ptr→d2[k];
        ptr→pos[k] = ((((0.05 * ptr→d3[k] * dt + OneTwelfth * f2dotk) * dt +
            ptr→f_dot[k]) * dt + ptr→f[k]) * dt + ptr→vel0[k]) * dt +
            ptr→pos0[k];
        ptr→vel[k] = (((0.25 * ptr→d3[k] * dt + OneThird * f2dotk) * dt +
            3 * ptr→f_dot[k]) * dt + 2 * ptr→f[k]) * dt + ptr→vel0[k];
    }

    ptr→pos_status = ptr→vel_status = HI_PRED;
}

void PredictPosAndVelHiAll()
{
    /* Predicts position and velocity of all particles to high order */

    static double last_time = 0;

    int i;

    if (last_time == Clock.time)
        return;

    for (i = 0; i < NumParticles; i++) {
        Data[i]→pos_status = Data[i]→vel_status = UN_PRED;
        PredictPosAndVelHi(Data[i]);
    }

    last_time = Clock.time;
}

void PredictPosAndQMomAll(node)
NODE_T *node;
{
    /* Predicts pos. and quad. mom's of tree nodes, starting at "node" */

    int i;

    if (TreePar.pred_mono)
        PREDICT_COM_POS(node);

    if (TreePar.use_quad && TreePar.pred_quad)
        PREDICT_Q_MOM(node);

    for (i = 0; i < MAX_NUM_CHILDREN; i++)
        if (node→child_type[i] == BRANCH)
            PredictPosAndQMomAll(node→child[i].branch);
}

#define ONE_SIXTIETH 0.01666666666666666667                /* Useful fraction */

double ElapsedCpu()
{
#ifndef SYSV
    /* Returns total CPU usage (in minutes) since program start */

    struct rusage tb;

    (void) getrusage(RUSAGE_SELF, &tb);
    return ONE_SIXTIETH * (tb.ru_utime.tv_sec + tb.ru_stime.tv_sec);
#else
    return 0.0;
#endif
}

#undef ONE_SIXTIETH

double TotalCpu()
{
    /*
     * Returns total CPU usage (in minutes) since start of first run.
     * (EvolPar.total_cpu is needed to keep track of CPU over restarts).
     *
     */

    static double last_time = 0;

    double time, elapsed_time;

    elapsed_time = (time = ElapsedCpu()) - last_time;
    last_time = time;

    return (EvolPar.total_cpu += elapsed_time);
}

void TimeStamp()
{
    /* Puts time stamp in log */

    (void) fprintf(Logfile, "%s TIME = %.3e CPU min = %.3e Nsteps = %li\n",
        GetDate(), TIME, ElapsedCpu(), Counter[TIME_STEPS]);
}

int BackupFile(filename, marker)
char *filename, marker;                                              /*ARGSUSED*/
{
#ifndef SYSV
    /*
     * Moves "filename" to "filename" + 'marker'. Returns 0 on success,
     * otherwise returns -1 and gives warning (plus system error message)
     * if I/O error occurs.
     *
     */

    char backup_filename[MAX_FILENAME_LEN];

    if (EMPTY_STR(filename))
        return 0;

    /* Reset external error flag */

    errno = 0;

    /* Append marker */

    (void) sprintf(backup_filename, "%s%c", filename, marker);

    /* Perform move if possible */
    if (rename(filename, backup_filename) ≠ 0 && errno ≠ ENOENT) {
        Error(IO, "BackupFile()", backup_filename);
        return -1;
    }

    if (errno ≠ ENOENT && VERBOSE)
        (void) printf("[Old \"%s\" moved to \"%s\".]\n", filename,
            backup_filename);

    return 0;
#else
    return -1;
#endif
}

#define WRITE(ptr, size, num) \
        if (fwrite((char *) ptr, size, num, save_file) ≠ num) {\
            Error(IO, msg, SaveFilename);\
            (void) fclose(save_file);\
            return;\
        }

void SaveRestartData()
{
    /*
     * Performs binary dump of all data needed for exact restart. Note
     * that all saved memory addresses are meaningless and must be
     * reassigned when reading (c.f. ReadRestartData()).
     *
     */

    char *msg = "SaveRestartData() (aborting save)";

    int i, dum;
    FILE *save_file;

    /* Get new CPU total (stored in EvolPar.total_cpu) */

    (void) TotalCpu();

    /*
     * If desired, first move existing save file (if any) to backup file
     * (SaveFilename with RunPar.save_file_index appended) and increment
     * index if no error occurs. The old file will be overwritten if it
     * cannot be moved.
     *
     */

    if (RunPar.num_save_files > 0 &&
            BackupFile(SaveFilename, '0' + RunPar.save_file_index) == 0 &&
            ++RunPar.save_file_index == RunPar.num_save_files)
        RunPar.save_file_index = 0;

    /* Open save file */

    (void) printf("Saving restart data to \"%s\" (program time %f).\n",
        SaveFilename, TIME);

    if ((save_file = fopen(SaveFilename, "w")) == NULL) {
        Error(IO, "SaveRestartData()", SaveFilename);
        return;
    }

    /* Save unalterable params.h parameters for error check on restart */

    dum = NUM_PHYS_DIM;
    WRITE(&dum, sizeof(int), 1);
    dum = NUM_TREE_DIM;
    WRITE(&dum, sizeof(int), 1);
    dum = MAX_NUM_PARTICLES;
    WRITE(&dum, sizeof(int), 1);
    dum = MAX_NUM_ON_TSL;
    WRITE(&dum, sizeof(int), 1);
    dum = MAX_NUM_TO_TRACK;
    WRITE(&dum, sizeof(int), 1);
    dum = MAX_NUM_TO_EXCLUDE;
    WRITE(&dum, sizeof(int), 1);
    dum = MAX_NUM_FILENUM_DIGITS;
    WRITE(&dum, sizeof(int), 1);
    dum = MAX_FILENAME_LEN;
    WRITE(&dum, sizeof(int), 1);
    dum = MAX_STR_LEN;
    WRITE(&dum, sizeof(int), 1);
    dum = WORKSPACE_SIZE;
    WRITE(&dum, sizeof(int), 1);

    /* Save all run-time parameters and counters */

    WRITE(&NumParticles, sizeof(int), 1);
    WRITE(&NumBoxes, sizeof(int), 1);
    WRITE(&RunPar, sizeof(RUN_PAR_T), 1);
    WRITE(&EvolPar, sizeof(EVOL_PAR_T), 1);
    WRITE(&TreePar, sizeof(TREE_PAR_T), 1);
    WRITE(&MoviePar, sizeof(MOVIE_PAR_T), 1);
    WRITE(&DebugPar, sizeof(DEBUG_PAR_T), 1);
    WRITE(&Clock, sizeof(CLOCK_T), 1);
    WRITE(Counter, sizeof(int), NUM_COUNTERS);
    WRITE(&Tsl, sizeof(TSL_T), 1);

    /* Save particle data */

    for (i = 0; i < NumParticles; i++)
        WRITE(Data[i], sizeof(DATA_T), 1);

    /* Save tree data using recursive procedure */

    if (RunPar.use_tree)
        save_tree_data(Root, save_file);

    /* Close save file */

    if (fclose(save_file) ≠ 0)
        Error(IO, "SaveRestartData()", SaveFilename);
}

static void save_tree_data(node, save_file)
NODE_T *node;
FILE *save_file;
{
    /* Recursively saves "node" data to "save_file" */
    char *msg = "save_tree_data() (aborting save)";

    int i;

    WRITE(node, sizeof(NODE_T), 1);

    for (i = 0; i < MAX_NUM_CHILDREN; i++)
        if (node→child_type[i] == BRANCH)
            save_tree_data(node→child[i].branch, save_file);
}

#undef WRITE

void ReadRestartData()
{
    /* Restores data from binary dump, allocating new memory as required */

    int i, dum;
    FILE *save_file;
    char *msg = "ReadRestartData():  params.h changed since last save.";

    /* Open save file */

    if ((save_file = fopen(SaveFilename, "r")) == NULL)
        Error(FATAL_IO, "ReadRestartData()", SaveFilename);

    /* Check whether program parameters have changed (see params.h) */

    (void) fread((char *) &dum, sizeof(int), 1, save_file);
    if (dum ≠ NUM_PHYS_DIM) {
        (void) sprintf(ErrorStr, "NUM_PHYS_DIM was %i now %i", dum,
            NUM_PHYS_DIM);
        Error(FATAL, msg, ErrorStr);
    }
    (void) fread((char *) &dum, sizeof(int), 1, save_file);
    if (dum ≠ NUM_TREE_DIM) {
        (void) sprintf(ErrorStr, "NUM_TREE_DIM was %i now %i", dum,
            NUM_TREE_DIM);
        Error(FATAL, msg, ErrorStr);
    }
    (void) fread((char *) &dum, sizeof(int), 1, save_file);
    if (dum ≠ MAX_NUM_PARTICLES) {
        (void) sprintf(ErrorStr, "MAX_NUM_PARTICLES was %i now %i", dum,
            MAX_NUM_PARTICLES);
        Error(FATAL, msg, ErrorStr);
    }
    (void) fread((char *) &dum, sizeof(int), 1, save_file);
    if (dum ≠ MAX_NUM_ON_TSL) {
        (void) sprintf(ErrorStr, "MAX_NUM_ON_TSL was %i now %i", dum,
            MAX_NUM_ON_TSL);
        Error(FATAL, msg, ErrorStr);
    }
    (void) fread((char *) &dum, sizeof(int), 1, save_file);
    if (dum ≠ MAX_NUM_TO_TRACK) {
        (void) sprintf(ErrorStr, "MAX_NUM_TO_TRACK was %i now %i", dum,
            MAX_NUM_TO_TRACK);
        Error(FATAL, msg, ErrorStr);
    }
    (void) fread((char *) &dum, sizeof(int), 1, save_file);
```

```
if (dum ≠ MAX_NUM_TO_EXCLUDE) {
    (void) sprintf(ErrorStr, "MAX_NUM_TO_EXCLUDE was %i now %i", dum,
        MAX_NUM_TO_EXCLUDE);
    Error(FATAL, msg, ErrorStr);
}
(void) fread((char *) &dum, sizeof(int), 1, save_file);
if (dum ≠ MAX_NUM_FILENUM_DIGITS) {
    (void) sprintf(ErrorStr, "MAX_NUM_FILENUM_DIGITS was %i now %i", dum,
        MAX_NUM_FILENUM_DIGITS);
    Error(FATAL, msg, ErrorStr);
}
(void) fread((char *) &dum, sizeof(int), 1, save_file);
if (dum ≠ MAX_FILENAME_LEN) {
    (void) sprintf(ErrorStr, "MAX_FILENAME_LEN was %i now %i", dum,
        MAX_FILENAME_LEN);
    Error(FATAL, msg, ErrorStr);
}
(void) fread((char *) &dum, sizeof(int), 1, save_file);
if (dum ≠ MAX_STR_LEN) {
    (void) sprintf(ErrorStr, "MAX_STR_LEN was %i now %i", dum,
        MAX_STR_LEN);
    Error(FATAL, msg, ErrorStr);
}
(void) fread((char *) &dum, sizeof(int), 1, save_file);
if (dum ≠ WORKSPACE_SIZE) {
    (void) sprintf(ErrorStr, "WORKSPACE_SIZE was %i now %i", dum,
        WORKSPACE_SIZE);
    Error(FATAL, msg, ErrorStr);
}

/* Read all parameters and counters */

(void) fread((char *) &NumParticles, sizeof(int), 1, save_file);
(void) fread((char *) &NumBoxes, sizeof(int), 1, save_file);
(void) fread((char *) &RunPar, sizeof(RUN_PAR_T), 1, save_file);
(void) fread((char *) &EvolPar, sizeof(EVOL_PAR_T), 1, save_file);
(void) fread((char *) &TreePar, sizeof(TREE_PAR_T), 1, save_file);
(void) fread((char *) &MoviePar, sizeof(MOVIE_PAR_T), 1, save_file);
(void) fread((char *) &DebugPar, sizeof(DEBUG_PAR_T), 1, save_file);
(void) fread((char *) &Clock, sizeof(CLOCK_T), 1, save_file);
(void) fread((char *) Counter, sizeof(int), NUM_COUNTERS, save_file);
(void) fread((char *) &Tsl, sizeof(TSL_T), 1, save_file);

/* Read particle data */

for (i = 0; i < NumParticles; i++) {
    Data[i] = (DATA_T *) malloc(sizeof(DATA_T));
    (void) fread((char *) Data[i], sizeof(DATA_T), 1, save_file);
}

/* Read tree data */

if (RunPar.use_tree) {
    Root = (NODE_T *) malloc(sizeof(NODE_T));
    read_tree_data(Root, save_file);
}

/* Close save file */

(void) fclose(save_file);
```

```
}

static void read_tree_data(node, save_file)
NODE_T *node;
FILE *save_file;
{
    /* Recursively reads "node" data from "save_file" */

    int i;
    CHILD_T *child;

    (void) fread((char *) node, sizeof(NODE_T), 1, save_file);

    for (i = 0; i < MAX_NUM_CHILDREN; i++) {
        child = &node→child[i];
        if (node→child_type[i] == BRANCH) {
            child→branch = (NODE_T *) malloc(sizeof(NODE_T));
            read_tree_data(child→branch, save_file);
            (child→branch)→parent = node;
        }
        else if (node→child_type[i] == LEAF) {
            Data[child→leaf]→node = node;
            Data[child→leaf]→node_index = i;
        }
    }
}

void Error(flag, msg1, msg2)
int flag;
char *msg1, *msg2;
{
    /*
     * Displays error message "msg1" on stdout (and optionally stderr and
     * the log) and, depending on the severity ("flag"), may terminate
     * execution (c.f. Terminate()). Extra information ("msg2") is printed
     * if desired.
     *
     */

    char error[MAX_STR_LEN], *msg = "";
    BOOLEAN fatal = (flag ≠ WARNING1 && flag ≠ WARNING2 && flag ≠ IO);

    if (EMPTY_STR(msg1))
        msg = "unknown error";
    else
        switch (flag) {
            case WARNING1:
                msg = "major warning in ";
                ++Counter[WARNINGS];
                break;
            case WARNING2:
                msg = "minor warning in ";
                ++Counter[WARNINGS];
                break;
            case IO:
                msg = "I/O error warning in ";
                ++Counter[IO_ERRORS];
                break;
            case FATAL_IO:
                (void) printf("\n");
                msg = "fatal I/O error in ";
                break;
            case FATAL:
                (void) printf("\n\07");                /* (\07 = <CTRL><G> audio beep) */
                msg = "fatal error in ";
                break;
            case HALT:
            case SYS_ERR:
                (void) printf("\n\07");
                msg = "";
                break;
            default:
                Terminate(ERROR);
        }

    /* Use "error" here because "Workspace" may be in use in calling routine */

    (void) sprintf(error, "box_tree -- %s%s\n", msg, msg1);

    (void) printf("@%s", error);

    if (fatal || flag == WARNING1) {
        (void) fprintf(stderr, " %s", error);
        if (Logfile)
        (void) fprintf(Logfile, "@%s", error);
    }

    if (!EMPTY_STR(msg2)) {
        if (flag == IO || flag == FATAL_IO) {
            (void) sprintf(error, " system message -- %s", msg2);
            perror(error);
        }
        else {
            (void) printf(" (%s).\n", msg2);
            if (fatal || flag == WARNING1) {
                (void) fprintf(stderr, " (%s).\n", msg2);
                if (Logfile)
                    (void) fprintf(Logfile, " (%s).\n", msg2);
            }
        }
    }

    if (fatal)
        Terminate(flag == HALT ? USER_HALT : ERROR);
}

void Terminate(status)
int status;
{
    /*
     * Terminates execution, returning 1 if program terminated abnormally
     * (i.e. with fatal error) or 0 if program terminated normally.
     *
     */

    if (status == ERROR) {
        char *msg = "*** FATAL ERROR in box_tree\n";

        (void) fprintf(stderr, "\n\07%s", msg);
        if (Logfile)
```

```
            (void) fprintf(Logfile, "%s", msg);
    }

    /* OK to use "Workspace" here since program is terminating */

    if (status == ALL_DONE)
        (void) sprintf(Workspace, "Run completed %s.\n", GetDate());
    else
        (void) sprintf(Workspace,
            "Program halted at t = %.5e (CPU %.3e min this run, %i steps).\n",
            TIME, ElapsedCpu(), Counter[TIME_STEPS]);

    (void) printf("%s", Workspace);

    if (Logfile)
        (void) fprintf(Logfile, "%s", Workspace);

    if (status == ERROR)                                          /* Try to dump core */
        abort();

    if (Logfile)
        (void) fclose(Logfile);

    exit(status == ERROR ? 1 : 0);
}

/* misc.c */
```

B.1.15 `output.c`

With the exception of `DisplayParams()` and various short messages and warnings found throughout the code, all of the `box_tree` output is generated by the routines in this file. The various forms of periodic output described in §A.5 are all represented: `LongOutput()` generates the main periodic output; `CalcEvolPar()` calculates and displays the evolving parameters; `OutputStats()` appends to the stats file; `OutputDat()` creates particle data files; `MakeMovieFrame()` outputs movie frames; and `OutputNlvData()` appends to the NLV file. The function `OpenStatsFile()` is used to open the stats file initially for writing. The only local function, `calc_data()`, is called by `LongOutput()`, `CalcEvolPar()` and `OutputStats()`. The procedure calculates all of the statistical data used in the output routines and stores it in a large static structure (called `store`) which is local to the file and therefore accessible by the output routines. A check of the simulation clock is made each time `calc_data()` is called to make sure the statistical data is not recalculated unnecessarily.

```
/*
 * output.c - DCR 91-05-03
 * ========================
 * Various output routines for box_tree code.
 *
 * Global functions: LongOutput(), CalcEvolPar(), OpenStatsFile(),
 *     OutputStats(), OutputDat(), MakeMovieFrame(), OutputNlvData().
 */

/* Include files */

#include "box_tree.h"
```

```
/* Local variables */

static struct {                                        /* For storage of useful statistics */
    double com_pos[NUM_PHYS_DIM];
    double com_vel[NUM_PHYS_DIM];
    double com_pv[NUM_PHYS_DIM];
    double vel_disp[NUM_PHYS_DIM];
    double total_mass;
    double max_mass;
    int max_mass_index;
    double mean_dist;
    double mean_min_dist;
    double ke;
    double re;
    double max_z;
    int max_z_index;
    double scale_height;
    double h_osc;
    double tzam;
    double mean_ecc;
    double ff0;
    double lv;
    double mean_mass;
    double median_mass;
    double mean_radius;
    double mean_roche_radius;
    double vel_disp_mag;
    double gpe;
    double total_energy;
    double tzam_diff;
    double total_energy_diff;
    double cpo;
    double mfp;
} store;

/* Local functions */

static void calc_data();

/* End of preamble */

#define MAX_Z(ptr) sqrt(SQ((ptr)→pos[2]) + SQ((ptr)→vel[2]))

void LongOutput()
{
    /* Performs verbose output to stdout */

    DATA_T *ptr;

    /* Fill "store" structure */

    calc_data();

    /* Start printing... */

    (void) printf("\n");

    if (GHOSTS)
        (void) printf("Centre b");
    else
    (void) printf("B");
(void) printf("ox stats at t = %g (total CPU = %.2e min) ", TIME,
      TotalCpu());
(void) printf("after %i steps:\n", Counter[TIME_STEPS]);

(void) printf(
    " Com pos:  x = %+12.5e y = %+12.5e z = %+12.5e (mag %.3e)\n",
    store.com_pos[0], store.com_pos[1], store.com_pos[2],
    MAG(store.com_pos));

(void) printf(
    " >initial:  x = %+12.5e y = %+12.5e z = %+12.5e (mag %.3e)\n",
    DebugPar.com_pos[0], DebugPar.com_pos[1], DebugPar.com_pos[2],
    MAG(DebugPar.com_pos));

(void) printf(
    " Com vel:  x = %+12.5e y = %+12.5e z = %+12.5e (mag %.3e)\n",
    store.com_vel[0], store.com_vel[1], store.com_vel[2],
    MAG(store.com_vel));

(void) printf(
    " >initial:  x = %+12.5e y = %+12.5e z = %+12.5e (mag %.3e)\n",
    DebugPar.com_vel[0], DebugPar.com_vel[1], DebugPar.com_vel[2],
    MAG(DebugPar.com_vel));

(void) printf(
    " P.v.  err:  x = %12.5e y = %12.5e z = %12.5e (mag %.3e)\n",
    ABS(store.com_pv[0]), ABS(store.com_pv[1]), ABS(store.com_pv[2]),
    MAG(store.com_pv));

(void) printf(
    " Vel disp:  x = %12.5e y = %12.5e z = %12.5e (mag %.3e)\n",
    store.vel_disp[0], store.vel_disp[1], store.vel_disp[2],
    store.vel_disp_mag);

ptr = Data[store.max_mass_index];
(void) printf(
    " Max mass:  %i (%i) mass %.2e (frac.  of tot.  %.1e) max z %.5e\n",
    store.max_mass_index, ptr→orig_index, store.max_mass,
    store.max_mass / store.total_mass, MAX_Z(ptr));

(void) printf(" Mean mass:  %.2e (radius:  %.2e Roche radius:  %.2e)\n",
    store.mean_mass, store.mean_radius, store.mean_roche_radius);

(void) printf(" Mean dist:  %.2e (%.1f RR", store.mean_dist,
    store.mean_dist / store.mean_roche_radius);
if (RunPar.bc_opt ≠ UNBOUNDED)
    (void) printf(", %.2f box_size)", store.mean_dist / BOX_SIZE);
else
    (void) printf(")");

(void) printf(" Mean min dist %.2e (%.1f RR)\n", store.mean_min_dist,
    store.mean_min_dist / store.mean_roche_radius);

ptr = Data[store.max_z_index];
(void) printf(" Max z:  %i (%i) mass %.2e z %+12.5e max z %.5e\n",
    store.max_z_index, ptr→orig_index, ptr→mass, ptr→pos[2],
    MAX_Z(ptr));

(void) printf(" tzam/M: init %.4e adj cur %.4e diff %.2e rms %.2e\n",
    DebugPar.tzam / store.total_mass, store.tzam / store.total_mass,
    store.tzam_diff / store.total_mass,
    sqrt(DebugPar.tzam_rms_err / DebugPar.num_calls) / store.total_mass);

(void) printf(" energy:  init %.4e adj cur %.4e diff %.2e rms %.2e\n",
    DebugPar.total_energy, store.total_energy, store.total_energy_diff,
    sqrt(DebugPar.total_energy_rms_err / DebugPar.num_calls));

(void) printf(" (KE %.5e (coll %.5e) RE %.5e GPE %.5e)\n",
    store.ke, - DebugPar.collision_dke, store.re, store.gpe);

(void) printf(" Other:  <ecc> %.5e hgt %.5e <osc> %.4e\n",
    store.mean_ecc, store.scale_height, store.h_osc);
(void) printf(" FF(0) %.2e <c/p/o> %.3e mfp %.3e lv %.3e\n",
        store.ff0, store.cpo, store.mfp, store.lv);

/* Display counters */

(void) printf(" Counters:\n");
(void) printf(" Time-steps:  %i\n", Counter[TIME_STEPS]);
(void) printf(" Min time-steps:  %i\n", Counter[MIN_TIME_STEPS]);
(void) printf(" Max time-steps:  %i\n", Counter[MAX_TIME_STEPS]);
(void) printf(" Collisions:  %i\n", Counter[COLLISIONS]);
(void) printf(" First-time col'ns:  %i\n",
    Counter[FIRST_TIME_COLLISIONS]);
(void) printf(" Ghost collisions:  %i\n", Counter[GHOST_COLLISIONS]);
(void) printf(" Mergers:  %i\n", Counter[MERGERS]);
(void) printf(" Forced mergers:  %i\n", Counter[FORCED_MERGERS]);
(void) printf(" Box boundary xings:  %i\n", Counter[BNDRY_XINGS]);
(void) printf(" L-R boundary xings:  %i\n",
    Counter[LATERAL_BNDRY_XINGS]);
(void) printf(" Ghost box xings:  %i\n",
    Counter[GHOST_BOX_BNDRY_XINGS]);
(void) printf(" Total mono updates:  %i\n",
    Counter[TOTAL_MONO_UPDATES]);
(void) printf(" Recur mono updates:  %i\n",
    Counter[RECUR_MONO_UPDATES]);
(void) printf(" Total quad updates:  %i\n",
    Counter[TOTAL_QUAD_UPDATES]);
(void) printf(" Recur quad updates:  %i\n",
    Counter[RECUR_QUAD_UPDATES]);
(void) printf(" Node packings:  %i\n", Counter[PACKINGS]);
(void) printf(" Force errors:  %i\n", Counter[FORCE_ERRORS]);
(void) printf(" Warnings:  %i\n", Counter[WARNINGS]);
(void) printf(" I/O errors:  %i\n", Counter[IO_ERRORS]);

/* Display any collision statistics */

if (Counter[COLLISIONS]) {
    int i;

    (void) printf("\n Collision statistics:\n\n");
    (void) printf(
        " Particle Last collider Nc M/min z_max/<hgt>\n");
    (void) printf(
        " ----------- ------------- ---- ----- -----------\n");
    for (i = 0; i < NumParticles; i++) {
        ptr = Data[i];
        if (ptr→last_collider ≠ -1) {
            if (ERROR_CHECK && ptr→num_collisions ≤ 0)
```

```
                    Error(FATAL, "LongOutput(): Bad collision stats.", "");
                (void) printf(" %4i (%4i) %4i (%4i) %4i %.3f", i,
                    ptr→orig_index, CurrentIndex(ptr→last_collider),
                    ptr→last_collider, ptr→num_collisions, ptr→mass /
                    RunPar.init_min_mass);
                if (store.scale_height)
                    (void) printf(" %.3f", MAX_Z(ptr) / store.scale_height);
                else
                    (void) printf(" (undef)");
                (void) printf("\n");
            }
        }
    }

    (void) printf("\n[END OUTPUT]\n");
}

void CalcEvolPar()
{
    /* Calculates and outputs evolving parameters */

    static BOOLEAN first_call = TRUE;

    EVOL_PAR_T *ptr = &EvolPar;

    calc_data();

    ptr→mean_mass = store.mean_mass;
    ptr→vel_disp = store.vel_disp_mag;
    ptr→median_mass = store.median_mass;
    ptr→mean_radius = store.mean_radius;
    ptr→mean_roche_radius = store.mean_roche_radius;

    /* Calculate radial coefficient of restitution velocity cutoff */

    ptr→min_rad_vel = (RunPar.no_slide ? 0.01 * ptr→vel_disp : 0);

    /* Calculate closest-particle detection zone */

    ptr→cp_zone_sq = SQ(100 * ptr→mean_roche_radius);

    /* Calculate self-gravity check zone if tree is active */

    if (RunPar.use_tree)
        ptr→self_grav_r2 = SQ(0.01 * TREE_SIZE);

    /* Display new parameters if desired */

    if (VERBOSE && (!first_call || Clock.time > RunPar.init_clock_time ||
            !Clock.timer[EVOL])) {
        (void) printf("\n");
        (void) printf("Status of evolving parameters, t = %g:\n", TIME);
        (void) printf(" Mag of vel disp = %e\n", ptr→vel_disp);
        (void) printf(" Mean mass = %e\n", ptr→mean_mass);
        (void) printf(" Median mass = %e\n", ptr→median_mass);
        (void) printf(" Mean radius = %e\n", ptr→mean_radius);
        (void) printf(" Mean Roche radius = %e\n", ptr→mean_roche_radius);
        (void) printf(" CP min radial vel = %e\n", ptr→min_rad_vel);
        (void) printf(" CP check zone = %e\n", sqrt(ptr→cp_zone_sq));
        if (RunPar.use_tree) {
                (void) printf(" Current tree size = %e\n", TreePar.tree_size);
                (void) printf(" Self-grav check zone = %e\n",
                    sqrt(ptr→self_grav_r2));
            }
            (void) printf(" Total CPU (min) = %e\n", TotalCpu());
            (void) printf("\n");
        }

        first_call = FALSE;
}

#define WRITE(addr, size, num) \
        if (fwrite((char *) addr, size, num, stats_file) ≠ num) {\
            Error(IO, msg, RunPar.stats_filename);\
            (void) fclose(stats_file);\
            return;\
        }

void OpenStatsFile()
{
    /* Opens new statistics file and writes out comment line */

    static char *msg = "OpenStatsFile() (aborting)";

    FILE *stats_file;

    if ((stats_file = fopen(RunPar.stats_filename, "w")) == NULL) {
        Error(IO, msg, RunPar.stats_filename);
        return;
    }

    WRITE(RunPar.comment_line, sizeof(char), MAX_STR_LEN);

    if (fclose(stats_file))
        Error(IO, msg, RunPar.stats_filename);
}

void OutputStats()
{
    /* Appends summary to statistics file */

    static char *msg = "OutputStats() (aborting save)";

    FILE *stats_file;
    DATA_T *ptr;
    double dum_dbl;

    calc_data();

    if ((stats_file = fopen(RunPar.stats_filename, "a")) == NULL) {
        Error(IO, msg, RunPar.stats_filename);
        return;
    }

    dum_dbl = TIME;
    WRITE(&dum_dbl, sizeof(double), 1);
    (void) TotalCpu();                                 /* (store in EvolPar.total_cpu) */
    WRITE(&EvolPar.total_cpu, sizeof(double), 1);
    WRITE(&Counter[TIME_STEPS], sizeof(int), 1);
    WRITE(&NumParticles, sizeof(int), 1);
    WRITE(&Counter[COLLISIONS], sizeof(int), 1);
    WRITE(&Counter[FIRST_TIME_COLLISIONS], sizeof(int), 1);
    WRITE(store.com_pos, sizeof(double), NUM_PHYS_DIM);
    WRITE(store.com_vel, sizeof(double), NUM_PHYS_DIM);
    WRITE(store.vel_disp, sizeof(double), NUM_PHYS_DIM);
    WRITE(&store.total_mass, sizeof(double), 1);
    WRITE(&store.max_mass, sizeof(double), 1);
    ptr = Data[store.max_mass_index];
    WRITE(ptr→pos, sizeof(double), NUM_PHYS_DIM);
    SUB_SHEAR(ptr);
    WRITE(ptr→vel, sizeof(double), NUM_PHYS_DIM);
    WRITE(ptr→spin, sizeof(double), NUM_PHYS_DIM);
    WRITE(&store.tzam, sizeof(double), 1);
    WRITE(&store.tzam_diff, sizeof(double), 1);
    dum_dbl = sqrt(DebugPar.tzam_rms_err / DebugPar.num_calls);
    WRITE(&dum_dbl, sizeof(double), 1);
    WRITE(&store.total_energy, sizeof(double), 1);
    WRITE(&DebugPar.collision_dke, sizeof(double), 1);
    WRITE(&store.total_energy_diff, sizeof(double), 1);
    dum_dbl = sqrt(DebugPar.total_energy_rms_err / DebugPar.num_calls);
    WRITE(&dum_dbl, sizeof(double), 1);
    WRITE(&store.mean_ecc, sizeof(double), 1);
    WRITE(&store.scale_height, sizeof(double), 1);
    WRITE(&store.ff0, sizeof(double), 1);
    WRITE(&store.cpo, sizeof(double), 1);
    WRITE(&store.mfp, sizeof(double), 1);
    WRITE(&store.lv, sizeof(double), 1);

    /* Close file */

    if (fclose(stats_file))
        Error(IO, msg, RunPar.stats_filename);
}

#undef WRITE

#define WRITE(addr, size, num) \
        if (fwrite((char *) addr, size, num, dat_file) ≠ num) {\
            Error(IO, msg, dat_filename);\
            (void) fclose(dat_file);\
            return;\
        }

void OutputDat()
{
    /* Saves particle data to current output file */

    static char *msg = "OutputDat() (aborting save)";

    int i, dum_int;
    char *dat_filename;
    FILE *dat_file;
    DATA_T *ptr;
    double dum_dbl;

    /* Construct filename and prepare file */

    if ((dat_filename = MakeFilename(RunPar.dat_basename,
            RunPar.dat_number++, ".dat")) == NULL) {
        Error(WARNING2, "OutputDat():  Output skipped.", "");
        return;
    }

    (void) printf("\nWriting particle data to \"%s\", time %f...\n",
        dat_filename, TIME);

    if (BackupFiles)
        (void) BackupFile(dat_filename, BACKUP_MARKER);

    if ((dat_file = fopen(dat_filename, "w")) == NULL) {
        Error(IO, msg, dat_filename);
        return;
    }

    /* Output data */

    WRITE(RunPar.comment_line, sizeof(char), MAX_STR_LEN);
    dum_dbl = TIME;
    WRITE(&dum_dbl, sizeof(double), 1);

    for (i = 0; i < NumParticles; i++) {
        ptr = Data[i];
        WRITE(&ptr→orig_index, sizeof(int), 1);
        WRITE(&ptr→mass, sizeof(double), 1);
        WRITE(&ptr→radius, sizeof(double), 1);
        WRITE(ptr→pos, sizeof(double), NUM_PHYS_DIM);
        WRITE(ptr→vel, sizeof(double), NUM_PHYS_DIM);
        SUB_SHEAR(ptr);
        WRITE(&ptr→vel[1], sizeof(double), 1);
        ADD_SHEAR(ptr);
        WRITE(ptr→spin, sizeof(double), NUM_PHYS_DIM);
        dum_int = ptr→color;
        WRITE(&dum_int, sizeof(int), 1);
    }

    (void) fclose(dat_file);
}

#undef WRITE

void MakeMovieFrame()
{
#ifndef SYSV
# ifndef ALPHA
    /* Makes a movie frame */

    int options;

    PredictPosAndVelHiAll();

    if (ROTATING_FRAME && GHOSTS)
        UpdateBoxPos();

    options = PLOT_POS;                                  /* Always plot particle positions */

    if (RunPar.use_tree && MoviePar.draw_tree)
        options |= DRAW_TREE | PLOT_COM | COM_LINES;

    if (GHOSTS && RunPar.bc_opt ≠ UNBOUNDED)
        options |= DRAW_BOXES;
    if (MoviePar.plot_vel)
        options |= PLOT_VEL;

    Draw(options);
# endif
#endif
}

#define WRITE(addr, size, num) \
        if (fwrite((char *) addr, size, num, nlv_file) ≠ num) {\
            Error(IO, msg, RunPar.nlv_filename);\
            (void) fclose(nlv_file);\
            return;\
        }

void OutputNlvData(m, x)
double m, x;
{
    /* Saves non-local viscosity data */

    static BOOLEAN first_call = TRUE;
    static FILE *nlv_file = (FILE *) NULL;

    char *msg = "OutputNlvData() (aborting save)";
    float tf, mf, xf;                                   /* Use floats to reduce file size */

    if (first_call) {
        nlv_file = fopen(RunPar.nlv_filename, "a");
        if (nlv_file == NULL) {
            Error(IO, "OutputNlvData()", RunPar.nlv_filename);
            return;
        }
    }

    if (!nlv_file)
        return;

    tf = (float) Clock.time;
    mf = (float) m;
    xf = (float) x;

    /* Output data */

    WRITE(&tf, sizeof(float), 1);
    WRITE(&mf, sizeof(float), 1);
    WRITE(&xf, sizeof(float), 1);

    /* Flush after every write because file is never closed properly */

    (void) fflush(nlv_file);
}

#undef WRITE

static void calc_data()
{
    /* Calculates various useful statistics */

    static BOOLEAN first_call = TRUE;
static double last_time = 0;

int i, j, k;
DATA_T *ptr;
double *mass_array, mass, rel_pos[NUM_PHYS_DIM], dist, min_dist, ecc_sq,
    err;

if (!first_call && Clock.time == last_time)
    return;

/* Predict all positions and velocities to high order */

PredictPosAndVelHiAll();

/* Initialize arrays */

ZERO(store.com_pos);
ZERO(store.com_vel);
ZERO(store.com_pv);
ZERO(store.vel_disp);

/* Initialize other quantities */

store.total_mass = store.max_mass = store.mean_dist = store.mean_min_dist =
    store.ke = store.re = store.scale_height = store.h_osc = store.tzam =
    store.mean_ecc = store.ff0 = store.lv = 0;

mass_array = (double *) malloc((unsigned) NumParticles * sizeof(double));

/*
 * Calculate various statistical/diagnostic quantities. Note that
 * energies and dispersions are calculated using PECULIAR velocities
 * w.r.t. Keplerian rotation, so shearing motion in y direction is
 * removed.
 *
 */

for (i = 0; i < NumParticles; i++) {
    ptr = Data[i];
    mass = ptr→mass;
    mass_array[i] = mass;
    store.total_mass += mass;

    if (mass > store.max_mass || (mass == store.max_mass &&
            MAX_Z(ptr) > MAX_Z(Data[store.max_mass_index]))) {
        store.max_mass = mass;
        store.max_mass_index = i;
    }

    for (k = 0; k < NUM_PHYS_DIM; k++) {
        store.com_pos[k] += mass * ptr→pos[k];
        store.com_vel[k] += mass * ptr→vel[k];
    }

    for (min_dist = HUGE_VAL, j = i + 1; j < NumParticles; j++) {
        SUB(ptr→pos, Data[j]→pos, rel_pos);
        dist = MAG(rel_pos);
        min_dist = MIN(min_dist, dist);
        store.mean_dist += dist;
    }
```

```
if (min_dist < HUGE_VAL)
    store.mean_min_dist += min_dist;

SUB_SHEAR(ptr);

for (k = 0; k < NUM_PHYS_DIM; k++) {
    if (RunPar.bc_opt ≠ UNBOUNDED)
        store.com_pv[k] += mass * ptr→vel[k];
    store.vel_disp[k] += mass * SQ(ptr→vel[k]);
}

if (INERTIAL_FRAME) {
    store.ke += mass * DOT(ptr→vel, ptr→vel);
    store.re += ptr→inertia * DOT(ptr→spin, ptr→spin);
}

if (ABS(ptr→pos[2]) > store.max_z) {
    store.max_z = ABS(ptr→pos[2]);
    store.max_z_index = i;
}

store.scale_height += mass * SQ(ptr→pos[2]);
store.h_osc += mass * (SQ(ptr→pos[2]) + SQ(ptr→vel[2]));

/* Calculate total z angular momentum and mean eccentricity */

ecc_sq = 0;

if (ROTATING_FRAME) {
    double lz;

    /* (r x v) - 1 to first order */

    lz = ptr→vel[1] + 2 * ptr→pos[0];

    /* Total z ang mom, correct to 1st order, within add. const. */

    store.tzam += mass * lz + ptr→inertia * ptr→spin[2];

    /* Eccentricity w.r.t. central mass, to first order */

    ecc_sq = 1 - SQ(1 + lz) * (1 - 4 * ptr→pos[0] - 2 * ptr→vel[1]);
}
else if (INERTIAL_FRAME)
    store.tzam += mass * CROSS_Z(ptr→pos, ptr→vel) +
        ptr→inertia * ptr→spin[2];

store.mean_ecc += (ecc_sq < 0 ? 0 : ecc_sq);

/* Add contribution to mid-plane filling factor */

if (RunPar.bc_opt ≠ UNBOUNDED && ABS(ptr→pos[2]) < ptr→radius)
    store.ff0 += PI * (ptr→radius_sq - SQ(ptr→pos[2]));

/* Local viscosity */

store.lv += mass * ptr→vel[0] * ptr→vel[1];

ADD_SHEAR(ptr);
```

```
}

store.mean_mass = store.total_mass / NumParticles;

store.median_mass = Median(NumParticles, mass_array);
free((char *) mass_array);

/* Calculate radius and Roche radius of mean mass */

store.mean_radius = Radius(store.mean_mass);
store.mean_roche_radius = RocheRadius(store.mean_mass);

NORM(store.com_pos, store.total_mass);
NORM(store.com_vel, store.total_mass);

if (NumParticles > 1)
    store.mean_dist /= (0.5 * NumParticles * (NumParticles - 1));

store.mean_min_dist /= NumParticles;

if (RunPar.bc_opt ≠ UNBOUNDED)
    NORM(store.com_pv, store.total_mass * BOX_SIZE);

for (k = 0; k < NUM_PHYS_DIM; k++)
    store.vel_disp[k] = sqrt(store.vel_disp[k] / store.total_mass);

store.vel_disp_mag = MAG(store.vel_disp);

store.ke * = 0.5;
store.re * = 0.5;
store.gpe = Gpe();

store.total_energy = store.ke + store.re + store.gpe;

store.scale_height = sqrt(store.scale_height / store.total_mass);
store.h_osc = sqrt(0.5 * store.h_osc / store.total_mass);

if (first_call) {                          /* (note error stats reset on restart currently) */

    /* Initialize com, tzam, and total energy error data */

    COPY(store.com_pos, DebugPar.com_pos);
    COPY(store.com_vel, DebugPar.com_vel);
    DebugPar.tzam = store.tzam;
    if (ERROR_CHECK && ABS(DebugPar.tzam) ≤ PRECISION && !GALAXY_FRAME)
        Error(WARNING1, "calc_data(): tzam ∼ 0; error unnormalized.", "");
    DebugPar.tzam_adj = DebugPar.tzam_rms_err = 0;
    DebugPar.total_energy = store.total_energy;
    if (ERROR_CHECK && ABS(DebugPar.total_energy) ≤ PRECISION &&
            INERTIAL_FRAME)
        Error(WARNING1, "calc_data(): TE ∼ 0; error unnormalized.", "");
    DebugPar.collision_dke = DebugPar.total_energy_adj =
        DebugPar.total_energy_rms_err = 0;
    DebugPar.num_calls = 0;
}
else if (!APPROX_EQ(store.total_mass, RunPar.total_mass))
    Error(WARNING2, "calc_data(): Poor mass conservation.", "");

/* Get tzam/TE errors */
```

```
    store.tzam += DebugPar.tzam_adj;
    store.tzam_diff = (ERROR_CHECK ? store.tzam - DebugPar.tzam : 0);
    err = store.tzam_diff;
    if (ABS(DebugPar.tzam) > PRECISION)
        err /= DebugPar.tzam;
    DebugPar.tzam_rms_err += SQ(err);
    store.total_energy += DebugPar.total_energy_adj;
    store.total_energy_diff = (ERROR_CHECK || RunPar.conserve_total_energy ?
        store.total_energy - DebugPar.total_energy : 0);
    err = store.total_energy_diff;
    if (ABS(DebugPar.total_energy) > PRECISION)
        err /= DebugPar.total_energy;
    DebugPar.total_energy_rms_err += SQ(err);
    ++DebugPar.num_calls;

    /* Remaining items... */

    store.mean_ecc = sqrt(store.mean_ecc / NumParticles);

    if (RunPar.bc_opt ≠ UNBOUNDED)
        store.ff0 /= SQ(BOX_SIZE);

    /* Calculate mean collisions/particle/orbit and mean free path */

    store.cpo = (Clock.time == 0 ? 0 :
        Counter[COLLISIONS] / (NumParticles * TIME));
    store.mfp = (Counter[COLLISIONS] == 0 ? 0 : (store.vel_disp_mag *
        Clock.time / Counter[COLLISIONS]) / store.mean_radius);

    store.lv * = (TwoThirds / store.total_mass);

    last_time = Clock.time;
    first_call = FALSE;
}

/* output.c */
```

B.1.16 params.c

This file consists of only two routines, both global and both called only from `main()`: `GetParams()` and `DisplayParams()`. The former routine reads the `box_tree` parameter file using the `rdpar` parsing functions and performs extensive checks on the input. The latter routine displays the parameters on the screen (`stdout`) once they have been loaded. Both routines are self-explanatory. Note that the box, view, and size parameters are set at the end of `GetParams()` if they were not set explicitly in the parameter file.

```
/*
 * params.c – DCR 93-06-18
 * ========================
 * Routines for reading and displaying user-supplied box_tree parameters.
 *
 * Global functions: GetParams(), DisplayParams().
 *
 */

/* Include files */

#include "box_tree.h"
#include <rdpar.h>
```

```
/* Additional definitions */

#define TWO_PI_SCALE (ROTATING_FRAME ? TWO_PI : 1.0)

/* End of preamble */

void GetParams(par_filename, restart)
char *par_filename;
BOOLEAN restart;
{
    /*
     * Reads all or just changeable parameters from "par_filename",
     * depending on "restart". This routine makes heavy use of rdpar.
     *
     */

    char *msg = "GetParams():  Invalid choice/bad syntax.";

    int i, ic_opt, bc_opt, dum_int, dum_int_array[2];
    SHAPE_T dum_shape;
    FILE *dum_file;
    RUN_PAR_T *ptr = &RunPar;
    REST_COEF_T *ptrc = &ptr->rest_coef;
    DRAG_COEF_T *ptrd = &ptr->drag_coef;
    TREE_PAR_T *ptrt = &TreePar;
    MOVIE_PAR_T *ptrm = &MoviePar;
    DEBUG_PAR_T *ptre = &DebugPar;
    double dum_dbl;
    BOOLEAN dum_boolean;

    /*
     * Check whether parameter file exists. If it doesn't and this is a
     * restart, return now and use old parameters, otherwise abort.
     *
     */

    if ((dum_file = fopen(par_filename, "r")) == NULL) {
        if (restart) {
            Error(IO, "GetParams()", par_filename);
            (void) printf("[using old parameters for restart]\n");
            return;
        }
        else
            Error(FATAL_IO, "GetParams()", par_filename);
    }
    (void) fclose(dum_file);

    (void) printf("\nReading parameter file \"%s\"...\n", par_filename);

    OpenPar(par_filename);

    /* Read header line (changeable) */

    ReadStr("Comment line", ptr->comment_line, MAX_STR_LEN);

    /* Get fixed/initial parameters if required */

    if (!restart) {
```

```
/* Reference frame */

ReadInt("Reference frame", &ptr→ref_frame);
if (ptr→ref_frame ≠ ROTATING && ptr→ref_frame ≠ INERTIAL &&
        ptr→ref_frame ≠ GALAXY)
    Error(FATAL, msg, "unknown reference frame");

/* Unit conversions to mks */

ReadDbl("Length scale", &ptr→length_scale);
if (ptr→length_scale ≤ 0)
    Error(FATAL, msg, "length scale must be positive");

ReadDbl("Mass scale", &ptr→mass_scale);
if (ptr→mass_scale ≤ 0)
    Error(FATAL, msg, "mass scale must be positive");

ReadDbl("Time scale", &ptr→time_scale);
if (ptr→time_scale ≤ 0)
    Error(FATAL, msg, "time scale must be positive");

/* Calculate velocity scale and density conversion factor */

ptr→velocity_scale = TWO_PI_SCALE * ptr→length_scale /
    ptr→time_scale;

ptr→density_conv = CUBE(ptr→length_scale) / ptr→mass_scale;

/* Random number generator seed */

ReadInt("Random number seed", &ptr→ran.seed);
if (ptr→ran.seed < 0)
    Error(FATAL, msg, "random number seed must be non-negative");

/* Initial conditions option */

ReadInt("Init cond option", &ic_opt);
if (ic_opt ≠ ALIGNED_COM && ic_opt ≠ UNIFORM_RAN && ic_opt ≠ WT &&
        ic_opt ≠ CLOSE_PACKED && ic_opt ≠ SUPPLIED)
    Error(FATAL, msg, "unknown initial condition option");
if ((INERTIAL_FRAME || GALAXY_FRAME) && ic_opt ≠ SUPPLIED)
    Error(FATAL, msg, "must use supplied IC's with chosen ref frame");
ptr→ic_opt = ic_opt;

ReadInt("Bdry cond option", &bc_opt);
if (bc_opt ≠ PERIODIC && bc_opt ≠ UNBOUNDED && bc_opt ≠ DISABLED)
    Error(FATAL, msg, "unknown boundary condition option");
if (ROTATING_FRAME && bc_opt ≠ PERIODIC)
    Error(FATAL, msg, "need periodic BC's for rotating frame");
ptr→bc_opt = bc_opt;

/* Following parameters apply to all options... */

ReadInt("Use ghost particles?", &dum_boolean);
if (dum_boolean)
    NumBoxes = MAX_NUM_BOXES;
else
    NumBoxes = 1;
if (NumBoxes > 1 && bc_opt == UNBOUNDED)
    Error(FATAL, msg, "cannot use unbounded BC's with ghosts");
```

```
ReadDbl("Box size", &ptr→box_size);
if (ptr→box_size == 0 && ic_opt ≠ WT && bc_opt ≠ UNBOUNDED)
     Error(FATAL, msg, "box size cannot be zero");
if (ptr→box_size < 0)
     ptr→box_size * = - ptr→length_scale;

ReadDbl("Initial clock time", &ptr→init_clock_time);
if (ptr→init_clock_time < 0)
     ptr→init_clock_time * = - ptr→time_scale;
ptr→init_clock_time * = TWO_PI_SCALE;

ReadDbl("Initial x vel disp", &ptr→init_x_vel_disp);
if (ptr→init_x_vel_disp < 0)
     ptr→init_x_vel_disp * = - ptr→velocity_scale;

ReadDbl("Initial y vel disp", &ptr→init_y_vel_disp);
if (ptr→init_y_vel_disp < 0)
     ptr→init_y_vel_disp * = - ptr→velocity_scale;

ReadDbl("Initial z vel disp", &ptr→init_z_vel_disp);
if (ptr→init_z_vel_disp < 0)
     ptr→init_z_vel_disp * = - ptr→velocity_scale;

ReadInt("Use small dispersions?", &ptr→small_disp);

/* Get option-dependent parameters */

if (ic_opt == ALIGNED_COM || ic_opt == UNIFORM_RAN ||
          ic_opt == WT || ic_opt == CLOSE_PACKED) {

     ReadInt("Number of particles", &NumParticles);
     if (NumParticles < 1)
          Error(FATAL, msg, "number of particles must be positive");
     if (NumParticles > MAX_NUM_PARTICLES)
          Error(FATAL, msg, "too many particles");

     ReadDbl("Smallest initial mass", &ptr→init_min_mass);
     if (ptr→init_min_mass < 0)
          ptr→init_min_mass * = - ptr→mass_scale;

     ReadDbl("Largest initial mass", &ptr→init_max_mass);
     if (ptr→init_max_mass < 0)
          ptr→init_max_mass * = - ptr→mass_scale;
     if (ptr→init_max_mass < ptr→init_min_mass)
          Error(FATAL, msg, "max mass must be >= min mass");

     ReadDbl("Particle density", &ptr→density);
     if (ptr→density == 0)
          Error(FATAL, msg, "particle density cannot be zero");
     if (ptr→density < 0)
          ptr→density = - ptr→density;
     else
          ptr→density * = DENSITY_CGS_TO_MKS;
     ptr→density * = ptr→density_conv;

     ReadDbl("Smallest initial radius", &dum_dbl);
     if (dum_dbl < 0)
          dum_dbl * = - ptr→length_scale;
     if (dum_dbl ≠ 0)
```

```
            ptr→init_min_mass = Mass(dum_dbl);
        else if (ptr→init_min_mass == 0)
            Error(FATAL, msg, "minimum radius cannot be zero");

        ReadDbl("Largest initial radius", &dum_dbl);
        if (dum_dbl < 0)
            dum_dbl ∗ = - ptr→length_scale;
        if (dum_dbl ≠ 0) {
            if (APPROX_LT(dum_dbl, Radius(ptr→init_min_mass)))
                Error(FATAL, msg, "max radius must be >= min rad");
            else
                ptr→init_max_mass = Mass(dum_dbl);
        }

        if (ic_opt == CLOSE_PACKED &&
                ptr→init_min_mass ≠ ptr→init_max_mass)
            Error(FATAL, msg, "need m_min == m_max for close packing");
    }

    if (ic_opt == ALIGNED_COM || ic_opt == UNIFORM_RAN || ic_opt == WT) {

        ReadDbl("Mass function exponent", &ptr→mass_exponent);
        if (ptr→mass_exponent == -1)
            Error(FATAL, msg, "mass function exponent cannot be -1");

        ReadDbl("Seed mass", &ptr→seed_mass);
        if (ptr→seed_mass < 0)
            ptr→seed_mass ∗ = - ptr→mass_scale;
    }


    if (ic_opt == ALIGNED_COM || ic_opt == UNIFORM_RAN ||
            ic_opt == SUPPLIED) {

        ReadInt("Use softening?", &ptr→use_softening);
        if (GALAXY_FRAME && !ptr→use_softening)
            Error(FATAL, msg, "must use softening in galaxy simulation");
    }

    if (ic_opt == ALIGNED_COM || ic_opt == UNIFORM_RAN) {

        ReadInt("Reject init.  binaries?", &ptr→rej_init_bin);

        ReadDbl("Vertical scale height", &ptr→init_scale_height);
        if (ptr→init_scale_height < 0)
            Error(FATAL, msg, "vertical scale height must be non-negative");
    }

    if (ic_opt == UNIFORM_RAN) {

        ReadInt("Number of x divisions", &ptr→num_x_div);
        if (ptr→num_x_div < 1)
            Error(FATAL, msg, "number of x divisions must be positive");

        ReadInt("Number of y divisions", &ptr→num_y_div);
        if (ptr→num_y_div < 1)
            Error(FATAL, msg, "number of y divisions must be positive");

        if (ptr→num_x_div ∗ ptr→num_y_div > NumParticles)
            Error(FATAL, msg, "number of divisions > number of particles");
        }

        if (ic_opt == WT) {

            ReadDbl("Optical depth", &ptr→optical_depth);
            if (ptr→optical_depth ≤ 0 && ptr→box_size == 0)
                Error(FATAL, msg, "optical depth must be positive");

            ReadDbl("Disk thickness", &ptr→init_disk_thickness);
            if (ptr→init_disk_thickness < 0)
                Error(FATAL, msg, "initial disk thickness must be non-neg.");
        }

        if (ic_opt == CLOSE_PACKED) {

            ReadInt("Number of layers", &ptr→num_layers);
            if (ptr→num_layers < 1)
                Error(FATAL, msg, "must have one or more layers");
            dum_int = sqrt((double) NumParticles / ptr→num_layers);
            if (NumParticles ≠ SQ(dum_int) ∗ ptr→num_layers)
                Error(FATAL, msg, "N / N_l must be perfect square for packing");

            ReadInt("Expand radii?", &ptr→expand_radii);

            ReadInt("Stagger in z?", &ptr→stagger_in_z);
        }

        if (ic_opt == SUPPLIED) {

            ReadStr("Init cond filename", ptr→init_cond_filename,
                MAX_FILENAME_LEN);

            ReadInt("No.  header lines", &ptr→num_header_lines);
            if (ptr→num_header_lines < 0)
                Error(FATAL, msg, "number of header lines must be non-neg.");

            ReadInt("Add shear?", &ptr→add_shear);
            if (ptr→add_shear && !ROTATING_FRAME)
                Error(FATAL, msg, "must be in rotating frame to add shear");
        }

    }  /∗ if not restart ∗/

    /∗ Get changeable parameters ∗/

    ReadInt("Verbosity level", &ptr→verbosity_level);
    if (ptr→verbosity_level < 0)
        Error(FATAL, msg, "verbosity level must be non-negative");

    ReadInt("Debug level", &ptr→debug_level);
    if (ptr→debug_level < 0)
        Error(FATAL, msg, "debug level must be non-negative");

    ReadLng("Stop check", &ptr→stop_check);
    if (ptr→stop_check < 0)
        Error(FATAL, msg, "stop check interval must be non-negative");

    ReadLng("CPU check", &ptr→cpu_check);
    if (ptr→cpu_check < 0)
        Error(FATAL, msg, "CPU check interval must be non-negative");
```

```
ReadLng("Safety dump", &ptr→safety_dump);
if (ptr→safety_dump < 0)
    Error(FATAL, msg, "safety dump interval must be non-negative");

ReadLng("Log time stamp", &ptr→time_stamp);
if (ptr→time_stamp < 0)
    Error(FATAL, msg, "time stamp interval must be non-negative");

ReadDbl("Output interval", &dum_dbl);
if (dum_dbl < 0)
    dum_dbl ∗ = - ptr→time_scale;
dum_dbl ∗ = TWO_PI_SCALE;
if (restart && dum_dbl ≠ 0) {
    Clock.timer[OUTPUT] = (int) (Clock.time / dum_dbl) ∗ dum_dbl;
    if (Clock.timer[OUTPUT] ≤ Clock.time)
        Clock.timer[OUTPUT] += dum_dbl;
}
ptr→interval[OUTPUT] = dum_dbl;

ReadDbl("Stats interval", &dum_dbl);
if (dum_dbl < 0)
    dum_dbl ∗ = - ptr→time_scale;
dum_dbl ∗ = TWO_PI_SCALE;
if (restart && dum_dbl ≠ 0) {
    Clock.timer[STATS] = (int) (Clock.time / dum_dbl) ∗ dum_dbl;
    if (Clock.timer[STATS] ≤ Clock.time)
        Clock.timer[STATS] += dum_dbl;
}
ptr→interval[STATS] = dum_dbl;

ReadDbl("Dat interval", &dum_dbl);
if (dum_dbl < 0)
    dum_dbl ∗ = - ptr→time_scale;
dum_dbl ∗ = TWO_PI_SCALE;
if (restart && dum_dbl ≠ 0) {
    Clock.timer[DAT] = (int) (Clock.time / dum_dbl) ∗ dum_dbl;
    if (Clock.timer[DAT] ≤ Clock.time)
        Clock.timer[DAT] += dum_dbl;
}
ptr→interval[DAT] = dum_dbl;

ReadDbl("Evol par interval", &dum_dbl);
if (dum_dbl < 0)
    dum_dbl ∗ = - ptr→time_scale;
dum_dbl ∗ = TWO_PI_SCALE;
if (restart && dum_dbl ≠ 0) {
    Clock.timer[EVOL] = (int) (Clock.time / dum_dbl) ∗ dum_dbl;
    if (Clock.timer[EVOL] ≤ Clock.time)
        Clock.timer[EVOL] += dum_dbl;
}
ptr→interval[EVOL] = dum_dbl;

ReadDbl("Movie interval", &dum_dbl);
if (dum_dbl < 0)
    dum_dbl ∗ = - ptr→time_scale;
dum_dbl ∗ = TWO_PI_SCALE;
if (restart && dum_dbl ≠ 0) {
    Clock.timer[MOVIE] = (int) (Clock.time / dum_dbl) ∗ dum_dbl;
    if (Clock.timer[MOVIE] ≤ Clock.time)
```

```
          Clock.timer[MOVIE] += dum_dbl;
}
ptr→interval[MOVIE] = dum_dbl;

if (ERROR_CHECK) {
     ReadDbl("Debug/check interval", &dum_dbl);
     if (dum_dbl < 0)
          dum_dbl ∗ = - ptr→time_scale;
     dum_dbl ∗ = TWO_PI_SCALE;
     if (restart && dum_dbl ≠ 0) {
          Clock.timer[CHECK] = (int) (Clock.time / dum_dbl) ∗ dum_dbl;
          if (Clock.timer[CHECK] ≤ Clock.time)
               Clock.timer[CHECK] += dum_dbl;
     }
     ptr→interval[CHECK] = dum_dbl;
}
else
     ptr→interval[CHECK] = 0;

ReadDbl("Termination time", &ptr→termination_time);
if (ptr→termination_time < 0)
     ptr→termination_time ∗ = - ptr→time_scale;
ptr→termination_time ∗ = TWO_PI_SCALE;

if (ptr→cpu_check) {
     ReadDbl("Run time", &ptr→run_time);
     if (ptr→run_time < 0)
         Error(FATAL, msg, "run time must be non-negative");
}
else
     ptr→run_time = 0.0;

if (ptr→safety_dump) {
     ReadInt("No.  backup save files", &ptr→num_save_files);
     if (ptr→num_save_files < 0 || ptr→num_save_files > MAX_NUM_SAVE_FILES)
          Error(FATAL, msg, "invalid number of backup save files");
}
else
     ptr→num_save_files = 0;
ptr→save_file_index = 0;                              /∗ (regardless of restart) ∗/

ReadInt("TSF option", &ptr→tsf_opt);
if (ptr→tsf_opt ≠ RV_ONLY && ptr→tsf_opt ≠ RV_AND_F &&
          ptr→tsf_opt ≠ F_ONLY)
     Error(FATAL, msg, "invalid TSF option");
if (ptr→use_softening && ptr→tsf_opt ≠ F_ONLY)
     Error(FATAL, msg, "must use F_ONLY TSF option for softening");

ReadDbl("Time-step coefficient", &ptr→time_step_coef);
if (ptr→time_step_coef ≤ 0)
     Error(FATAL, msg, "time-step coefficient must be positive");

ReadDbl("Minimum time-step", &ptr→min_time_step);
if (ptr→min_time_step < 0)
     ptr→min_time_step ∗ = - ptr→time_scale;
ptr→min_time_step ∗ = TWO_PI_SCALE;

ReadDbl("Maximum time-step", &ptr→max_time_step);
if (ptr→max_time_step < 0)
     ptr→max_time_step ∗ = - ptr→time_scale;
```

```
ptr→max_time_step ∗ = TWO_PI_SCALE;
if (ptr→max_time_step && ptr→max_time_step < ptr→min_time_step)
     Error(FATAL, msg, "max time-step must be >= min step (or 0)");

ReadInt("Include self-gravity?", &dum_boolean);
if (restart && dum_boolean && !ptr→self_grav) {
     (void) printf("[Self-gravity turned on...(re)initializing...");
     PredictPosAndVelHiAll();
     if (ROTATING_FRAME && GHOSTS)
          UpdateBoxPos();
     for (i = 0; i < NumParticles; i++) {
          (void) ApplyBndryCond(i);
          InitLoOrderPoly(i);
     }
     for (i = 0; i < NumParticles; i++)
          InitHiOrderPoly(i);
     InitTsl();
     if (ptr→use_tree) {
          DeallocTree(Root);
          MakeTree(TREE_SIZE, TREE_CENTRE);
     }
     (void) printf("done]\n");
}
ptr→self_grav = dum_boolean;
if (ptr→tsf_opt ≠ RV_ONLY && !ptr→self_grav && INERTIAL_FRAME)
     Error(FATAL, msg, "TSF option selected requires particle gravity");

if (ROTATING_FRAME) {
     ReadDbl("Z grav enhance factor", &dum_dbl);
     if (dum_dbl < 1)
          Error(FATAL, msg, "gravity enhance factor must be 1 or greater");
     if (restart && SQ(dum_dbl) ≠ ptr→g_factor_sq)
          Error(FATAL, msg, "cannot change Z grav enhance factor on restart");
     ptr→g_factor_sq = SQ(dum_dbl);
}
else
     ptr→g_factor_sq = 0;

ReadDbl("CP sum of radii factor", &dum_dbl);
if (dum_dbl < 0 || dum_dbl > 1)
     Error(FATAL, msg, "CP sum of radii factor must be in [0,1]");
if (ptr→tsf_opt == RV_ONLY && dum_dbl == 1)
     Error(FATAL, msg, "CP factor must be < 1 for RV_ONLY TSF option");
ptr→cp_fac_sq = SQ(dum_dbl);

ReadDbl("Radial restitut'n coef.", &ptrc→radial);
if ((ptrc→radial < 0 || ptrc→radial > 1) && ptrc→radial ≠ BHL_FLAG)
     Error(FATAL, msg, "radial rest.  coef.  must lie in [0,1]");

ReadDbl("Trans.  restitut'n coef.", &ptrc→transverse);
if ((ptrc→transverse < - 1 || ptrc→transverse > 1) &&
          ptrc→transverse ≠ BHL_FLAG)
     Error(FATAL, msg, "transverse rest.  coef.  must lie in [-1,1]");

ReadInt("Inhibit sliding phase?", &ptr→no_slide);

if (!ptr→use_softening)
     ReadInt("Apply coll'n velo adj?", &ptr→conserve_total_energy);
else
     ptr→conserve_total_energy = FALSE;
```

```
ReadInt("Include gas drag?", &dum_boolean);
if (restart && dum_boolean && !ptr→include_drag)
    Error(FATAL, msg, "cannot activate gas drag on restart");
ptr→include_drag = dum_boolean;

if (ptr→include_drag) {
    ReadDbl("Drag coef in x", &ptrd→x);
    ReadDbl("Drag coef in y", &ptrd→y);
    ReadDbl("Drag coef in z", &ptrd→z);
    if (ptrd→x < 0 || ptrd→y < 0 || ptrd→z < 0)
        Error(FATAL, msg, "drag coefficients cannot be negative");
    ReadDbl("Constant drag in y", &ptrd→hdot);
    if (ptrd→hdot < 0)
       Error(FATAL, msg, "constant drag in y cannot be negative");
}
else
    ptrd→x = ptrd→y = ptrd→z = ptrd→hdot = 0;

ReadInt("Allow mergers?", &ptr→allow_mergers);
if (ptr→allow_mergers && !ptr→self_grav)
    Error(FATAL, msg, "must have interparticle gravity for mergers");

ReadInt("Use tree?", &dum_boolean);
if (restart && dum_boolean && !ptr→use_tree)
    Error(FATAL, msg, "cannot invoke tree on restart");
ptr→use_tree = dum_boolean;
if (ptr→use_tree && !ptr→self_grav)
    Error(WARNING1, "GetParams():  Tree invoked without gravity.", "");

/* Particle tracking parameters */

ptr→num_to_track = 0;

while (ReadNInt("Track particle", dum_int_array, 2) ≠ NEND) {
    if (dum_int_array[0] < 0)
        Error(FATAL, msg, "invalid track particle");
    dum_int = ptr→num_to_track;
    if (dum_int == MAX_NUM_TO_TRACK)
        Error(FATAL, msg, "too many particles to track");
    ptr→track_list[dum_int] = dum_int_array[0];
    ptr→track_colors[dum_int] = dum_int_array[1];
    ++ptr→num_to_track;
}

for (i = ptr→num_to_track; i < MAX_NUM_TO_TRACK; i++)
    ptr→track_list[i] = -1;

/* Miscellaneous output parameters */

if (ptr→interval[STATS]) {
    ReadStr("Stats filename", ptr→stats_filename, MAX_FILENAME_LEN);
    if (EMPTY_STR(ptr→stats_filename))
        Error(FATAL, msg, "must specify stats filename");
}
else
    ptr→stats_filename[0] = '\0';

if (ptr→interval[DAT]) {
    ReadStr("Dat file basename", ptr→dat_basename,
        MAX_FILENAME_LEN - MAX_NUM_FILENUM_DIGITS - 4);
    if (EMPTY_STR(ptr→dat_basename))
        Error(FATAL, msg, "must specify dat basename");
    ReadInt("Starting dat file no.", &dum_int);
    if (dum_int < -1)
        Error(FATAL, msg, "invalid starting dat output number");
    if (dum_int == -1 && !restart)
        ptr→dat_number = 0;
    else if (dum_int > -1)
        ptr→dat_number = dum_int;
    if (ptr→dat_number ≥ EXP10(MAX_NUM_FILENUM_DIGITS))
        Error(FATAL, msg, "starting dat output number too large");
}
else
    ptr→dat_basename[0] = '\0';

ReadStr("NLV output filename", ptr→nlv_filename, MAX_FILENAME_LEN);
if (!EMPTY_STR(ptr→nlv_filename) && ptr→allow_mergers)
    Error(FATAL, msg, "NLV algorithm cannot handle mergers");

/* Tree parameters */

if (ptr→use_tree) {

    if (!restart) {
        ReadDbl("Tree size", &ptrt→tree_size);
        if (ptrt→tree_size < 0)
            ptrt→tree_size *= - ptr→length_scale;
    }

    ReadDbl("Expansion factor", &ptrt→expansion);
    if (ptrt→expansion < 1)
        Error(FATAL, msg, "tree expansion factor must be >= 1");

    ReadDbl("Maximum opening angle", &dum_dbl);
    if (dum_dbl < 0)
        Error(FATAL, msg, "max opening angle must be non-negative");
    if ((ptrt→theta_sq = SQ(dum_dbl)) > 1.0 / NUM_TREE_DIM)
        Error(WARNING1, "GetParams():  Large angle.", "");

    ReadInt("Use quadrupole?", &dum_boolean);
    if (restart && dum_boolean && !ptrt→use_quad)
        Error(FATAL, msg, "cannot invoke quadrupole on restart");
    ptrt→use_quad = dum_boolean;

    ReadInt("Use minimum repair?", &ptrt→use_move);

    ReadInt("Use hi-ord.  prediction?", &ptrt→use_high_order);

    ReadInt("Predict monopole?", &ptrt→pred_mono);

    if (ptrt→use_quad && ptrt→pred_mono)
        ReadInt("Predict quadrupole?", &ptrt→pred_quad);
    else
        ptrt→pred_quad = FALSE;

    if (ptrt→pred_mono)
        ReadInt("Check update times?", &ptrt→check_update_times);
    else
        ptrt→check_update_times = FALSE;
        if (ptrt→check_update_times) {

            ReadDbl("Mono time-step coef", &ptrt→mtsc);
            if (ptrt→mtsc ≤ 0)
                Error(FATAL, msg, "mono tsc must be positive");

            if (ptrt→use_quad && ptrt→pred_quad) {
                ReadDbl("Quad time-step coef", &ptrt→qtsc);
                if (ptrt→qtsc < 0)
                    Error(FATAL, msg, "quad tsc must be positive");
            }
            else
                ptrt→qtsc = 0;
        }
        else
            ptrt→mtsc = ptrt→qtsc = 0;

        /* Particle exclude list (initial conditions only) */

        if (!restart) {
            ptrt→num_excluded = 0;

            while (ReadNInt("Exclude particle", dum_int_array, 1) ≠ NEND) {
                dum_int = ptrt→num_excluded;
                if (dum_int == MAX_NUM_TO_EXCLUDE)
                    Error(FATAL, msg, "too many particle exclusions");
                ptrt→exclude_list[dum_int] = dum_int_array[0];
                ++ptrt→num_excluded;
            }
        }
    }
    else {                                              /* Initialize for completeness */
        ptrt→tree_size = ptrt→half_tree_size = ptrt→expansion =
            ptrt→theta_sq = ptrt→mtsc = ptrt→qtsc = 0;
        ptrt→use_quad = ptrt→use_move = ptrt→use_high_order =
            ptrt→pred_mono = ptrt→pred_quad =
            ptrt→check_update_times = FALSE;
        ptrt→num_excluded = 0;
    }

    /* Movie parameters */

    if (ptr→interval[MOVIE]) {

        ReadStr("File basenames", ptrm→basename,
            MAX_FILENAME_LEN - MAX_NUM_FILENUM_DIGITS - 4);

        ReadInt("Starting frame number", &dum_int);
        if (dum_int < -1)
            Error(FATAL, msg, "invalid starting frame number");
        if (dum_int == -1 && !restart)
            ptrm→frame_number = 0;
        else if (dum_int > -1)
            ptrm→frame_number = dum_int;
        if (ptrm→frame_number ≥ EXP10(MAX_NUM_FILENUM_DIGITS))
            Error(FATAL, msg, "starting frame number too large");

        ReadInt("Frame size", &ptrm→frame_size);
        if (ptrm→frame_size ≤ 0)
```

```
            Error(FATAL, msg, "frame size must be non-negative");

        ReadDbl("View size", &ptrm→view_size);                    /* (0/neg val handled later) */

        (void) ReadNDbl("View centre", ptrm→view_centre, 2);

        if (ptr→use_tree)
            ReadInt("Draw tree?", &ptrm→draw_tree);
        else
            ptrm→draw_tree = FALSE;

        ReadInt("Particle shape", &dum_int);
        dum_shape = (SHAPE_T) dum_int;
        if (dum_shape ≠ POINT && dum_shape ≠ CIRCLE &&
                dum_shape ≠ SQUARE && dum_shape ≠ DIAMOND &&
                dum_shape ≠ DISK && dum_shape ≠ SPHERE)
            Error(FATAL, msg, "unknown particle shape");
        ptrm→particle_shape = dum_shape;

        ReadDbl("Radius magnification", &ptrm→radius_mag);
        if (ptrm→radius_mag ≤ 0)
            Error(FATAL, msg, "radius magnification must be non-negative");

        ReadDbl("Viewing distance", &ptrm→distance);
        if (ptrm→distance == 0)
            Error(FATAL, msg, "viewing distance cannot be zero");
        if (ptrm→distance < 0)
            ptrm→distance *= - ptr→length_scale;

        ReadDbl("Z magnification", &ptrm→z_mag);
        if (ptrm→z_mag ≤ 0)
            Error(FATAL, msg, "z magnification must be non-negative");

        ReadInt("Hide blocked objects?", &ptrm→hide_blocked_objects);

        ReadInt("Draw velocity vectors?", &ptrm→plot_vel);

        ReadInt("Default color", &dum_int);
        ptrm→dflt_color = dum_int;
    }
    else {
        ptrm→basename[0] = '\0';
        ptrm→frame_number = ptrm→frame_size = 0;
        ptrm→view_size = ptrm→half_view_size = ptrm→view_centre[0] =
            ptrm→view_centre[1] = ptrm→radius_mag =
            ptrm→distance = ptrm→z_mag = 0;
        ptrm→draw_tree = ptrm→hide_blocked_objects = ptrm→plot_vel = FALSE;
        ptrm→particle_shape = POINT;
        ptrm→dflt_color = WHITE;
    }

    /* Debug parameters */

    if (ptr→interval[CHECK]) {

        if (ptr→use_tree) {                          /* (all tests currently only apply to tree) */

            ReadInt("Check tree?", &ptre→check_tree);

            ReadInt("Check multipoles?", &ptre→check_multipoles);
            ReadInt("Check force?", &ptre→check_force);
            if (ptre→check_force && ptr→use_softening)
                Error(FATAL, msg, "cannot check softened forces");
        }
        else
            ptre→check_tree = ptre→check_multipoles =
                ptre→check_force = FALSE;

        if (!ptre→check_tree && !ptre→check_multipoles &&
                !ptre→check_force) {
              Error(WARNING2, "GetParams():  No debug flags.", "");
              ptr→interval[CHECK] = 0;
        }
    }
    else
        ptre→check_tree = ptre→check_multipoles = ptre→check_force = FALSE;

    /* DebugPar initializations (remainder performed in calc_data()) */

    if (!restart) {
        ptre→num_force_checks = 0;
        ptre→avg_force = ptre→max_force = ptre→total_err = ptre→max_err = 0;
    }

    /* Close parameter file */

    ClosePar();

    /* Set various quantities derived from input parameters... */

    /*
     * If optical depth specified, compute new box size assuming smooth
     * particle size distribution.
     *
     */

    if (ptr→optical_depth) {
        int nn;
        double r, sum_r2 = 0;

        nn = (ptr→seed_mass > 0);

        for (i = 0; i < NumParticles; i++)
            if (nn && i == 0) {
                r = Radius(ptr→seed_mass);
                sum_r2 += SQ(r);
            }
            else {
                r = Radius(InitMassFunc((double) (i - nn) /
                    (NumParticles - nn - 1)));
                sum_r2 += SQ(r);
            }

        ptr→box_size = sqrt(PI * sum_r2 / ptr→optical_depth);
        (void) printf("[box size set to %e]\n", ptr→box_size);
    }

    /* Box size info */
    ptr→half_box_size = 0.5 * ptr→box_size;
    ptr→sys_size = (GHOSTS ? 3 : 1) * ptr→box_size;
    ptr→half_sys_size = 0.5 * ptr→sys_size;

    /* Tree size info */

    if (ptr→use_tree) {
        if (ptrt→tree_size == 0) {
            if (ptr→box_size == 0)
                Error(FATAL, msg, "invalid box and/or tree size");
            ptrt→tree_size = ptr→box_size;
            if (bc_opt == PERIODIC && ptrt→tree_size < ptr→box_size)
                Error(FATAL, msg, "tree must be >= box size for periodic BC's");
        }
        ptrt→half_tree_size = 0.5 * ptrt→tree_size;
    }

    /* Movie view size info */

    if (ptr→interval[MOVIE]) {
        if (ptrm→view_size == 0)
            ptrm→view_size = (ptr→use_tree ? ptrt→tree_size : ptr→box_size);
        else if (ptrm→view_size < 0)
            ptrm→view_size * = - ptr→box_size;
        if (ptrm→view_size ≤ 0)
            Error(FATAL, msg, "invalid box and/or view size");
        ptrm→half_view_size = 0.5 * ptrm→view_size;
    }

    (void) printf("Done.\n");
}

void DisplayParams()
{
    /* Displays current run parameters */

    int i;
    RUN_PAR_T *ptr = &RunPar;

    (void) printf("\nComment line:  \"%s\"\n", RunPar.comment_line);

    (void) printf("\nSave file name = \"%s\"\n", SaveFilename);

    (void) printf("\nFixed/initial (read once) parameters:\n\n");

    (void) printf(" Reference frame = %i ", ptr→ref_frame);
    switch (ptr→ref_frame) {
        case ROTATING:
            (void) printf("(ROTATING)");
            break;
        case INERTIAL:
            (void) printf("(INERTIAL)");
            break;
        case GALAXY:
            (void) printf("(GALAXY)");
            break;
        default:
            (void) printf("(UNKNOWN)");
    }
    (void) printf("\n");
```

```
(void) printf("\n");
(void) printf(" Length scale = %g m\n", ptr→length_scale);
(void) printf(" Mass scale = %g kg\n", ptr→mass_scale);
(void) printf(" Time scale = %g s\n", ptr→time_scale);
(void) printf(" Velocity scale = %g m/s\n", ptr→velocity_scale);

(void) printf("\n");
(void) printf(" [all subsequent output will be in these units ");
(void) printf("unless otherwise indicated]\n");

(void) printf("\n");
(void) printf(" Random number seed = %i\n", ptr→ran.seed);
(void) printf(" Initial conditions opt = %i ", ptr→ic_opt);
switch (ptr→ic_opt) {
    case ALIGNED_COM:
        (void) printf("(ALIGN COM)");
        break;
    case UNIFORM_RAN:
        (void) printf("(UNIF RAN)");
        break;
    case WT:
        (void) printf("(WT)");
        break;
    case CLOSE_PACKED:
        (void) printf("(CLOSE PACKED)");
        break;
    case SUPPLIED:
        (void) printf("(SUPPLIED)");
        break;
    default:
        (void) printf("(UNKNOWN)");
}
(void) printf("\n");
(void) printf(" Boundary cond option = %i ", ptr→bc_opt);
switch (ptr→bc_opt) {
    case PERIODIC:
        (void) printf("(PERIODIC)");
        break;
    case UNBOUNDED:
        (void) printf("(UNBOUNDED)");
        break;
    case DISABLED:
        (void) printf("(DISABLED)");
        break;
    default:
        (void) printf("(UNKNOWN)");
}
(void) printf("\n");

(void) printf("\n");
(void) printf(" Number of boxes = %i\n", NumBoxes);
(void) printf(" Box size = %g\n", ptr→box_size);
(void) printf(" Initial clock time = %g\n",
    ptr→init_clock_time / TWO_PI_SCALE);
(void) printf(" Initial x-vel disp = %g\n", ptr→init_x_vel_disp);
(void) printf(" Initial y-vel disp = %g\n", ptr→init_y_vel_disp);
(void) printf(" Initial z-vel disp = %g\n", ptr→init_z_vel_disp);
(void) printf(" Small dispersions flag = %s\n", Boolean(ptr→small_disp));
```

```
(void) printf("\nOther fixed/initial physical parameters:\n");

if (ptr→ic_opt == ALIGNED_COM || ptr→ic_opt == UNIFORM_RAN ||
        ptr→ic_opt == WT || ptr→ic_opt == CLOSE_PACKED) {
    (void) printf("\n");
    (void) printf(" Number of particles = %i\n", NumParticles);
    (void) printf(" Initial min mass = %g\n", ptr→init_min_mass);
    (void) printf(" Initial max mass = %g\n", ptr→init_max_mass);
    (void) printf(" Particle density = %g g/cc\n",
        ptr→density / ptr→density_conv / DENSITY_CGS_TO_MKS);
}

if (ptr→ic_opt == ALIGNED_COM || ptr→ic_opt == UNIFORM_RAN ||
        ptr→ic_opt == WT) {
    (void) printf("\n");
    (void) printf(" Mass function exp = %g\n", ptr→mass_exponent);
    (void) printf(" Seed mass = %g", ptr→seed_mass);
    if (ptr→seed_mass == 0)
        (void) printf(" (none)");
    (void) printf("\n");
}

if (ptr→ic_opt == ALIGNED_COM || ptr→ic_opt == UNIFORM_RAN ||
        ptr→ic_opt == SUPPLIED) {
    (void) printf("\n");
    (void) printf(" Softening flag = %s\n", Boolean(ptr→use_softening));
}

if (ptr→ic_opt == ALIGNED_COM || ptr→ic_opt == UNIFORM_RAN) {
    (void) printf("\n");
    (void) printf(" Initial binary rejection = %s\n",
        Boolean(ptr→rej_init_bin));
    (void) printf(" Vertical scale height = %g ",
        ptr→init_scale_height);
    if (ptr→init_scale_height == 0)
        (void) printf("(ignored)");
    else
        (void) printf("(Roche radii)");
    (void) printf("\n");
}

if (ptr→ic_opt == UNIFORM_RAN) {
    (void) printf("\n");
    (void) printf(" No.  of x divisions = %i\n", ptr→num_x_div);
    (void) printf(" No.  of y divisions = %i\n", ptr→num_y_div);
}

if (ptr→ic_opt == WT) {
    (void) printf("\n");
    (void) printf(" Optical depth = %g\n", ptr→optical_depth);
    (void) printf(" Disk thickness = %g radii\n",
        ptr→init_disk_thickness);
}

if (ptr→ic_opt == CLOSE_PACKED) {
    (void) printf("\n");
    (void) printf(" Number of layers = %i\n", ptr→num_layers);
    (void) printf(" Expand radii = %s\n",
        Boolean(ptr→expand_radii));
    (void) printf(" Small dispersions = %s\n",
        Boolean(ptr→small_disp));
    (void) printf(" Stagger in z = %s\n",
        Boolean(ptr→stagger_in_z));
}

if (ptr→ic_opt == SUPPLIED) {
    (void) printf("\n");
    (void) printf(" Init cond filename = \"%s\"\n",
        ptr→init_cond_filename);
    (void) printf(" No.  header lines = %i\n", ptr→num_header_lines);
    if (ROTATING_FRAME)
        (void) printf(" Add shear = %s\n",
            Boolean(ptr→add_shear));
}

/* Variable parameters */

(void) printf("\nVariable (restart) parameters:\n");

(void) printf("\n");
(void) printf(" Verbosity level = %i\n", ptr→verbosity_level);
(void) printf(" Debug level = %i\n", ptr→debug_level);
(void) printf(" Stop check = %li\n", ptr→stop_check);
(void) printf(" CPU check = %li\n", ptr→cpu_check);
(void) printf(" Safety dump = %li\n", ptr→safety_dump);
(void) printf(" Time stamp = %li\n", ptr→time_stamp);

(void) printf(" Output interval = %g",
        ptr→interval[OUTPUT] / TWO_PI_SCALE);
if (ptr→interval[OUTPUT] == 0)
    (void) printf(" (no output)");
(void) printf("\n");

(void) printf(" Stats interval = %g",
        ptr→interval[STATS] / TWO_PI_SCALE);
if (ptr→interval[STATS] == 0)
    (void) printf(" (no stats)");
(void) printf("\n");

(void) printf(" Dat interval = %g",
        ptr→interval[DAT] / TWO_PI_SCALE);
if (ptr→interval[DAT] == 0)
    (void) printf(" (no dat output)");
(void) printf("\n");

(void) printf(" Evol par interval = %g",
        ptr→interval[EVOL] / TWO_PI_SCALE);
if (ptr→interval[EVOL] == 0)
    (void) printf(" (no par recalc)");
(void) printf("\n");

(void) printf(" Movie interval = %g",
        ptr→interval[MOVIE] / TWO_PI_SCALE);
if (ptr→interval[MOVIE] == 0)
    (void) printf(" (no movie)");
(void) printf("\n");

if (ERROR_CHECK) {
    (void) printf(" Debug/check interval = %g",
            ptr→interval[CHECK] / TWO_PI_SCALE);
        if (ptr→interval[CHECK] == 0)
            (void) printf(" (no checking)");
        (void) printf("\n");
    }

    (void) printf(" Termination time = %g\n",
            ptr→termination_time / TWO_PI_SCALE);

    if (ptr→cpu_check)
        (void) printf(" Run time = %g\n", ptr→run_time);

    if (ptr→safety_dump)
        (void) printf(" Number of save files = %i\n", ptr→num_save_files);

    (void) printf(" TSF option = %i ", ptr→tsf_opt);
    switch(ptr→tsf_opt) {
        case RV_ONLY:
            (void) printf("(R/V only)");
            break;
        case RV_AND_F:
            (void) printf("(R/V and F)");
            break;
        case F_ONLY:
            (void) printf("(F only)");
            break;
        default:
            (void) printf("(UNKNOWN)");
    }
    (void) printf("\n");
    (void) printf(" Time-step coefficient = %g\n", ptr→time_step_coef);
    (void) printf(" Minimum time-step = %g",
        ptr→min_time_step / TWO_PI_SCALE);
    if (!ptr→min_time_step)
        (void) printf(" (NONE)");
    (void) printf("\n");
    (void) printf(" Maximum time-step = %g",
        ptr→max_time_step / TWO_PI_SCALE);
    if (!ptr→max_time_step)
        (void) printf(" (NONE)");
    (void) printf("\n");
    (void) printf(" Self-gravity flag = %s\n", Boolean(ptr→self_grav));
    if (ROTATING_FRAME)
        (void) printf(" Z grav enhance factor = %g\n",
            sqrt(ptr→g_factor_sq));
    (void) printf(" CP sum of radii factor = %g\n", sqrt(ptr→cp_fac_sq));
    (void) printf(" Radial restit'n coef = %g", ptr→rest_coef.radial);
    if (ptr→rest_coef.radial == BHL_FLAG)
        (void) printf(" (BHL)");
    (void) printf("\n");
    (void) printf(" Trans.  restit'n coef = %g",
        ptr→rest_coef.transverse);
    if (ptr→rest_coef.transverse == BHL_FLAG)
        (void) printf(" (BHL)");
    (void) printf("\n");
    (void) printf(" No sliding phase flag = %s\n",
        Boolean(ptr→no_slide));
    (void) printf(" Apply velo adj flag = %s\n",
        Boolean(ptr→conserve_total_energy));
    (void) printf(" Include gas drag flag = %s\n",
        Boolean(ptr→include_drag));
```

```
if (ptr→include_drag) {
    (void) printf(" Drag coef.  in x = %g\n", ptr→drag_coef.x);
    (void) printf(" Drag coef.  in y = %g\n", ptr→drag_coef.y);
    (void) printf(" Drag coef.  in z = %g\n", ptr→drag_coef.z);
    (void) printf(" Constant drag in y = %g\n", ptr→drag_coef.hdot);
}
(void) printf(" Allow mergers flag = %s\n",
     Boolean(ptr→allow_mergers));
(void) printf(" Use tree flag = %s\n", Boolean(ptr→use_tree));

(void) printf("\n");
(void) printf(" Particle tracking:");
if (ptr→num_to_track == 0)
    (void) printf(" NONE");
else {
    COLOR_T color;

    for (i = 0; i < ptr→num_to_track; i++) {
        (void) printf(" %i", ptr→track_list[i]);
        if ((color = ptr→track_colors[i]) == BLACK)
            (void) printf(" (M)");
        else
            (void) printf(" [%i]", color);
    }
}
(void) printf("\n");

(void) printf("\nMiscellaneous output parameters:\n\n");

if (ptr→interval[STATS])
    (void) printf(" Stats filename:  \"%s\"\n", ptr→stats_filename);

if (ptr→interval[DAT]) {
    (void) printf(" Dat file basename:  \"%s\"\n", ptr→dat_basename);
    (void) printf(" Starting dat no.  = %i\n", ptr→dat_number);
}

if (!EMPTY_STR(ptr→nlv_filename))
    (void) printf(" NLV output filename:  \"%s\"\n", ptr→nlv_filename);

if (ptr→use_tree) {
    TREE_PAR_T *ptrt = &TreePar;

    (void) printf("\nTree parameters:\n\n");

    (void) printf(" Tree size = %g\n", ptrt→tree_size);
    (void) printf(" Expansion factor = %g", ptrt→expansion);
    if (ptrt→expansion == 1)
        (void) printf(" (NO EXPANSIONS)");
    (void) printf("\n");
    (void) printf(" Max opening angle (theta) = %g",
        sqrt(ptrt→theta_sq));
    if (ptrt→theta_sq > 1.0 / NUM_TREE_DIM)
        (void) printf(" (LARGE ANGLE)");
    (void) printf("\n");
    (void) printf(" Use quadrupole flag = %s\n",
        Boolean(ptrt→use_quad));
    (void) printf(" Use mimimum repair flag = %s\n",
        Boolean(ptrt→use_move));
    (void) printf(" Use high-order pred flag = %s\n",
        Boolean(ptrt→use_high_order));
    (void) printf(" Check update times flag = %s\n",
        Boolean(ptrt→check_update_times));
    (void) printf(" Predict monopole flag = %s\n",
        Boolean(ptrt→pred_mono));

    if (ptrt→use_quad)
        (void) printf(" Predict quadrupole flag = %s\n",
            Boolean(ptrt→pred_quad));

    if (ptrt→check_update_times) {
        (void) printf(" Mono time-step coef = %g\n", ptrt→mtsc);
        if (ptrt→use_quad)
            (void) printf(" Quad time-step coef = %g\n",
                ptrt→qtsc);
    }

    (void) printf("\n Exclusion list:");
    if (ptrt→num_excluded == 0)
        (void) printf(" (EMPTY)");
    else
        for (i = 0; i < ptrt→num_excluded; i++)
            (void) printf(" %i", ptrt→exclude_list[i]);

    (void) printf("\n\n Maximum allowed tree level = %i\n",
            MAX_TREE_LEVEL);
}

if (ptr→interval[MOVIE]) {
    MOVIE_PAR_T *ptrm = &MoviePar;

    (void) printf("\nMovie parameters:\n\n");

    (void) printf(" File basenames:  \"%s\"\n\n", ptrm→basename);
    (void) printf(" Starting frame number = %i\n", ptrm→frame_number);
    (void) printf(" Frame size = %i\n", ptrm→frame_size);
    (void) printf(" View size = %g\n", ptrm→view_size);
    (void) printf(" View centre = (%g,%g)\n",
        ptrm→view_centre[0], ptrm→view_centre[1]);
    if (ptr→use_tree)
        (void) printf(" Draw tree = %s\n",
            Boolean(ptrm→draw_tree));
    (void) printf(" Particle shape = %i", ptrm→particle_shape);
    switch (ptrm→particle_shape) {
        case POINT:
            (void) printf(" (POINT)");
            break;
        case CIRCLE:
            (void) printf(" (CIRCLE)");
            break;
        case SQUARE:
            (void) printf(" (SQUARE)");
            break;
        case DIAMOND:
            (void) printf(" (DIAMOND)");
            break;
        case DISK:
            (void) printf(" (DISK)");
            break;
        case SPHERE:
                (void) printf(" (SPHERE)");
                break;
            default:
                (void) printf(" (UNKNOWN)");
        }
        (void) printf("\n");
        (void) printf(" Radius magnification = %g\n", ptrm→radius_mag);
        (void) printf(" Viewing distance = %g\n", ptrm→distance);
        (void) printf(" Z magnification = %g\n", ptrm→z_mag);
        (void) printf(" Hide blocked objects = %s\n",
            Boolean(ptrm→hide_blocked_objects));
        (void) printf(" Draw velocity vectors = %s\n",
            Boolean(ptrm→plot_vel));
    }

    if (ptr→interval[CHECK]) {
        DEBUG_PAR_T *ptre = &DebugPar;

        (void) printf("\nDebug/checking parameters:\n\n");
        (void) printf(" Check tree flag = %s\n",
                Boolean(ptre→check_tree));
        (void) printf(" Check multipoles flag = %s\n",
                Boolean(ptre→check_multipoles));
        (void) printf(" Check force flag = %s\n",
                Boolean(ptre→check_force));
    }

    (void) printf("\n");
}

/* params.c */
```

B.1.17 `recipes.c`

This file contains a few routines borrowed from *Numerical Recipes in C* (Press et al. 1988) and modified slightly for use with `box_tree`. The routines (all global) consist of a binary search method (`Locate()`), a uniform random number generator (`Ran()`), a Gaussian deviate generator (`Gasdev()`), and two versions of a sort method, one for doubles (`Sort`), and one for doubles with a matching integer array (`Sort2`).

```
/*
 * recipes.c - DCR 91-06-24
 * ========================
 *
 * Various useful routines from "Numerical Recipes in C" (Press, et al 1988).
 * Modifications by DCR (note in particular that input zero-offset arrays are
 * converted to unit-offset arrays for compatability).
 *
 * Global functions: Locate(), Ran(), Gasdev(), Sort(), Sort2().
 *
 * The original Numerical Recipes routines are Copyright (C) 1987, 1988
 * Numerical Recipes Software, reproduced by permission, from the book
 * Numerical Recipes: The Art of Scientific Computing, published by
 * Cambridge University Press.
 *
 */

/* Include files */
```

```
#include "box_tree.h"

/* End of preamble */

void Locate(xx,n,x,j)                                   /* Based on locate(), NRiC 3.4 */
double xx[],x;
int n,*j;
{
    /* (returns j+1 (zero offset) using def'n of j in NRiC) */

    int ascnd,ju,jm,jl;

    --xx;                                                  /* Make unit-offset array */

    /*
     * Treat special cases (short lists, technically non-monotonic) and
     * allow for range of constant values (assumed to be "increasing").
     *
     */

    if (n == 0)
        *j = 0;                                                      /* Zero offset */
    else if (n == 1)
        *j = (x < xx[1] ? 0 : 1);                                    /* Zero offset */
    else {
        jl=0;
        ju=n+1;
        ascnd=xx[n] ≥ xx[1];                                          /* Note "=" */
        while (ju-jl > 1) {
            jm=(ju+jl) ≫ 1;
            if ((x ≥ xx[jm]) == ascnd)                                   /* Ditto */
                jl=jm;
            else
                ju=jm;
        }
        *j=jl;
    }
}

#define M1 259200
#define IA1 7141
#define IC1 54773
#define RM1 (1.0/M1)
#define M2 134456
#define IA2 8121
#define IC2 28411
#define RM2 (1.0/M2)
#define M3 243000
#define IA3 4561
#define IC3 51349

double Ran()                                             /* Based on ran1(), NRiC 7.1 */
{
    /* (ix1, ix2, ix3, r stored in RunPar.ran struct for restarts) */

    int j;
    RAN_T *ran = &RunPar.ran;
    double temp;

    if (ran→seed < 0) {
        ran→ix1=(IC1-(ran→seed)) % M1;
        ran→ix1=(IA1*ran→ix1+IC1) % M1;
        ran→ix2=ran→ix1 % M2;
        ran→ix1=(IA1*ran→ix1+IC1) % M1;
        ran→ix3=ran→ix1 % M3;
        for (j=1;j≤97;j++) {
            ran→ix1=(IA1*ran→ix1+IC1) % M1;
            ran→ix2=(IA2*ran→ix2+IC2) % M2;
            ran→r[j]=(ran→ix1+ran→ix2*RM2)*RM1;
        }
        ran→seed = - ran→seed;
    }
    ran→ix1=(IA1*ran→ix1+IC1) % M1;
    ran→ix2=(IA2*ran→ix2+IC2) % M2;
    ran→ix3=(IA3*ran→ix3+IC3) % M3;
    j=1 + ((97*ran→ix3)/M3);
    if (ERROR_CHECK && (j > 97 || j < 1))
        Error(FATAL, "Ran(): Invalid array index.", "");
    temp=ran→r[j];
    ran→r[j]=(ran→ix1+ran→ix2*RM2)*RM1;
    return temp;
}

#undef M1
#undef IA1
#undef IC1
#undef RM1
#undef M2
#undef IA2
#undef IC2
#undef RM2
#undef M3
#undef IA3
#undef IC3

double Gasdev()                                        /* Based on gasdev(), NRiC 7.2 */
{
    RAN_T *ptr = &RunPar.ran;
    double fac,r,v1,v2;

    if (ptr→iset == 0) {
        do {
            v1=2*Ran()-1;
            v2=2*Ran()-1;
            r=v1*v1+v2*v2;
        } while (r ≥ 1);
        fac=sqrt(-2*log(r)/r);
        ptr→gset=v1*fac;
        ptr→iset=1;
        return v2*fac;
    } else {
        ptr→iset=0;
        return ptr→gset;
    }
}

void Sort(n,ra)                                          /* Based on sort(), NRiC 8.2 */
int n;
double ra[];
{
    int l,j,ir,i;
    double rra;

    if (n == 1)
        return;

    --ra;                                              /* Make unit-offset array */

    l=(n >> 1)+1;
    ir=n;
    for (;;) {
        if (l > 1)
            rra=ra[--l];
        else {
            rra=ra[ir];
            ra[ir]=ra[1];
            if (--ir == 1) {
                ra[1]=rra;
                return;
            }
        }
        i=l;
        j=l << 1;
        while (j <= ir) {
            if (j < ir && ra[j] < ra[j+1]) ++j;
            if (rra < ra[j]) {
                ra[i]=ra[j];
                j += (i=j);
            }
            else j=ir+1;
        }
        ra[i]=rra;
    }
}

void Sort2(n,ra,rb)                                    /* Based on sort2(), NRiC 8.2 */
int n;
double ra[];
int rb[];
{
    int l,j,ir,i,rrb;
    double rra;

    if (n == 1)
        return;

    --ra;                                              /* Make unit-offset arrays */
    --rb;

    l=(n >> 1)+1;
    ir=n;
    for (;;) {
        if (l > 1) {
            rra=ra[--l];
            rrb=rb[l];
        } else {
            rra=ra[ir];
            rrb=rb[ir];
            ra[ir]=ra[1];
            rb[ir]=rb[1];
            if (--ir == 1) {
                ra[1]=rra;
                rb[1]=rrb;
                return;
            }
        }
        i=l;
        j=l ≪ 1;
        while (j ≤ ir) {
            if (j < ir && ra[j] < ra[j+1]) ++j;
            if (rra < ra[j]) {
                ra[i]=ra[j];
                rb[i]=rb[j];
                j += (i=j);
            }
            else j=ir+1;
        }
        ra[i]=rra;
        rb[i]=rrb;
    }
}

/* recipes.c */
```

B.1.18 repair_tree.c

This file contains the routines for performing tree repair (cf. §3.4.1). There are two global functions, `MoveInTree()` and `RemoveFromTree()`. The former implements the algorithm shown pictorially in Fig. 3.2, performing the minimum repair required to move a particle from one location in the tree to another. The latter function removes the particle concerned entirely from the tree and does not replace it. This is needed in the `collision()` and `merge()` routines (cf. `integrate.c`) to update all the ancestor nodes of the affected particles. The `MoveInTree()` function makes use of the local routines `other_particle()` for locating orphans and `repair_tree()` for performing the actual repair.

```
/*
 * repair_tree.c - DCR 91-04-30
 * ============================
 * Routines to move particles in tree and perform any necessary repairs.
 *
 * Global functions: MoveInTree(), RemoveFromTree().
 *
 */

/* Include files */

#include "box_tree.h"

/* Local functions */

static LEAF_T other_particle();
static void repair_tree();

/* End of preamble */

#define INSIDE_SUBNODE(particle, pos, node, index)\
    GetIndex(particle, pos, node) == (index)
```

```
void MoveInTree(particle, repair_root)
int particle;
BOOLEAN repair_root;
{
    /*
     * Moves particle "particle" in tree to position Data[particle]->pos.
     * Any necessary tree repair will be performed automatically. If
     * particle trajectory from previous to current time is discontinous
     * (e.g. boundary condition), "repair_root" should be set TRUE, so that
     * multipole moments of entire tree are properly adjusted.
     *
     */

    DATA_T *ptr = Data[particle];
    NODE_T *node = ptr→node;
    int index = ptr→node_index;
    double *pos = ptr→pos;

    /*
     * No corrections are needed if movement is all within leaf cell
     * (regardless of "repair_root"). However, updating is forced in rare
     * case of "packed" cell (c.f. PlaceInTree()), since node indices of
     * particles are not meaningful in such a situation.
     *
     */

    if (!node→packed && INSIDE_SUBNODE(particle, pos, node, index))
        return;

    /*
     * If particle leaves node and there are only two particles in node,
     * then identify particle left behind and repair tree. Otherwise, if
     * movement is between cells in node or outside node, empty node cell.
     * (Full tree repair is only necessary if there are exactly two leaves
     * total in the node and one of them moves out, so that there will be
     * only one leaf remaining. In this case the branch must become a leaf
     * of the parent or higher ancestor.) No repair is performed if the
     * branch is actually the root node.
     *
     */

    if (node→num_leaves == 2 && OUTSIDE_NODE(pos, node))
        repair_tree(pos, other_particle(particle, node), &node);
    else {
        if (ERROR_CHECK && node→child_type[index] ≠ LEAF) {
            (void) sprintf(ErrorStr, "index %i %s", index, NodeInfo(node));
            Error(FATAL, "MoveInTree():  Expected leaf.", ErrorStr);
        }
        node→child_type[index] = EMPTY;
        node→child[index].leaf = -1;
    }

    /*
     * Now subtract contribution of particle to multipole moments for
     * current node and each ancestor up the tree until movement is
     * contained in one entire node. Note that "pos", the true final
     * position, is used for determining whether the particle is contained
     * in "node", so that the particle will ALWAYS be in Root.
     *
     */

    while (OUTSIDE_NODE(pos, node)) {
        if (ERROR_CHECK && node == Root) {
            (void) sprintf(ErrorStr, "particle %i (%i)", particle,
                ptr→orig_index);
            Error(FATAL, "MoveInTree():  Particle has left tree!", ErrorStr);
        }
        UpdateBranchMoments(SUBM, particle, node);
        node = node→parent;
    }

    /*
     * Finally, put particle back in tree, first testing whether particle
     * actually left tree and boundary conditions were applied, in which
     * case the root node needs to be updated. Otherwise no update to the
     * current node is required.
     *
     */

    if (repair_root) {
        if (ERROR_CHECK && node ≠ Root) {
            (void) sprintf(ErrorStr, "particle %i (%i)", particle,
                ptr→orig_index);
            Error(WARNING2, "MoveInTree():  Redundant root repair?", ErrorStr);
        }
        UpdateBranchMoments(SUBM, particle, Root);
        PlaceInTree(particle, UPDATE, Root);
    }
    else
        PlaceInTree(particle, UPDATE_CHILDREN, node);
}

#undef INSIDE_SUBNODE

void RemoveFromTree(particle)
int particle;
{
    /*
     * Removes particle "particle" entirely from tree, repairing as
     * necessary and updating moments right up to root node. This routine
     * is needed for collisions and boundary conditions, but can also be
     * used in place of MoveInTree() if a call to this routine is
     * immediately followed by a call to PlaceInTree() (with UPDATE).
     * This will result in more accurate moments but increased CPU time.
     *
     */

    DATA_T *ptr = Data[particle];
    NODE_T *node = ptr→node;
    int index = ptr→node_index;

    /* Check whether tree repair is needed; otherwise, empty cell */

    if (node→num_leaves == 2 && node ≠ Root)
        repair_tree((double *) NULL, other_particle(particle, node), &node);
    else {
        if (ERROR_CHECK && node→child_type[index] ≠ LEAF) {
            (void) sprintf(ErrorStr, "index %i %s", index, NodeInfo(node));
            Error(FATAL, "RemoveFromTree():  Expected leaf.", ErrorStr);
        }
        node→child_type[index] = EMPTY;
        node→child[index].leaf = -1;
    }

    /* Update moments of all ancestors, including Root */

    do {
        UpdateBranchMoments(SUBM, particle, node);
        node = node→parent;
    } while (node ≠ NULL);

    /* Reset particle tree indices */

    ptr→node = NULL;
    ptr→node_index = -1;
}

static LEAF_T other_particle(particle, node)
int particle;
NODE_T *node;
{
    /*
     * Returns index of particle that ISN'T "particle" in the two-leaf
     * branch "node". This particle becomes an "orphan" temporarily
     * during tree repair. This function is intended for use with calls to
     * repair_tree().
     *
     */

    int i;
    LEAF_T other_leaf = -1;

    /* Examine children until OTHER particle is found */

    for (i = 0; i < MAX_NUM_CHILDREN; i++)
        if (node→child_type[i] == LEAF &&
                (other_leaf = node→child[i].leaf) ≠ particle)
            break;

    /* Error check */

    if (ERROR_CHECK && other_leaf == -1) {
        (void) sprintf(ErrorStr, "particle %i (%i) %s", particle,
            Data[particle]→orig_index, NodeInfo(node));
        Error(FATAL, "other_particle():  Unable to find other particle.",
            ErrorStr);
    }

    /* Return other particle */

    return other_leaf;
}

static void repair_tree(pos, leaf, node)
double *pos;
LEAF_T leaf;
NODE_T **node;
{
    /*
```

```
 * Removes branch nodes, beginning with "node" and continuing with its
 * ancestors, until "pos" lies inside the ancestral node (if "pos" is
 * not NULL) or until an ancestor with more than two leaves (before
 * repair) is found (or the ancestor is root and "pos" is NULL). The
 * address of this final node is returned in "node". Also, the former
 * branch entry in this node is replaced by "leaf".
 *
 */

NODE_T *parent_node;
int index;

/* Keep removing nodes until we find an appropriate ancestor */

do {
    if (*node == Root)
        Error(FATAL, "repair_tree(): Attempt to repair root.", "");
    parent_node = (*node)->parent;
    index = (*node)->node_index;
    free((char *) *node);
    *node = parent_node;
} while ((pos != (double *) NULL ? OUTSIDE_NODE(pos, (*node)) :
    *node != Root) && (*node)->num_leaves == 2);

/* Error check */

if (ERROR_CHECK && (*node)->child_type[index] != BRANCH) {
    (void) sprintf(ErrorStr, "index %i %s", index, NodeInfo(*node));
    Error(FATAL, "repair_tree(): Expected branch.", ErrorStr);
}

/* Replace branch with leaf and update Data array */

(*node)->child_type[index] = LEAF;
(*node)->child[index].leaf = leaf;

Data[leaf]->node = *node;
Data[leaf]->node_index = index;
}

/* repair_tree.c */
```

B.1.19 tree_util.c

This file consists of miscellaneous functions that operate on the tree: `DeallocTree()` to deallocate all memory associated with the tree; `TreeLevel()` to return the level (generation) of a given node; `NodeInfo()` to return a string containing information regarding a given node (useful for error messages); `GetOffspring()` to return a list of all the leaves contained in a given node and all its descendants; `NotOffspring()` to test whether a particle is a leaf of a given node or one of its descendants' branches (used by `add_tree_force()` to check for self-gravity problems); and `Node()` to return a pointer to a node with the given tree index. The `Node()` function makes use of the recursive local function `search_for_node()`.

```
/*
 * tree_util.c - DCR 91-09-12
 * ==========================
 * Various miscellaneous tree handling routines.
 *
 * Global functions(): DeallocTree(), TreeLevel(), NodeInfo(), GetOffspring(),
 *     NotOffspring(), Node().
 *
 */

/* Include files */

#include "box_tree.h"

/* Local functions */

static void search_for_node();

/* End of preamble */

void DeallocTree(node)
NODE_T *node;
{
    /* Deallocates tree, starting from "node" (should be Root on first call) */

    int i;

    for (i = 0; i < MAX_NUM_CHILDREN; i++)
        if (node->child_type[i] == BRANCH)
            DeallocTree(node->child[i].branch);

    free((char *) node);
}

int TreeLevel(node)
NODE_T *node;
{
    /* Returns depth of node in tree (number of "generations") */

    int level = 0;

    /* Move up hierarchy until Root is reached */

    while (node != Root) {
        node = node->parent;
        ++level;
    }

    return level;
}

char *NodeInfo(node)
NODE_T *node;
{
    /* Returns information concerning "node" in a character array */

    static char workspace[MAX_STR_LEN];                 /* (private to avoid conflicts) */

    (void) sprintf(workspace, "node lvl %i idx %i sz %.1e max %.1e lvs %i",
        TreeLevel(node), node->tree_index, node->size, node->max_size,
        node->num_leaves);

    return workspace;
}
```

```
void GetOffspring(node, num_leaves, offspring)
NODE_T *node;
int *num_leaves;
LEAF_T *offspring;
{
    /*
     * Returns pointer to list of leaves belonging to "node" and its
     * descendants at current time. Before calling, num_leaves should be
     * set to zero. Also, offspring should point to an array of
     * MAX_NUM_PARTICLES elements of type LEAF_T. It is the responsibility
     * of the calling routine to ensure the returned values are correct.
     *
     */

    int i;
    CHILD_T *child;

    /* Error check */

    if (ERROR_CHECK && (node == NULL || offspring == NULL))
        Error(FATAL, "GetOffspring():  Invalid argument(s).", "");

    /* Examine children: record leaves and descend branches */

    for (i = 0; i < MAX_NUM_CHILDREN; i++) {
        child = &node->child[i];
        if (node->child_type[i] == BRANCH)
            GetOffspring(child->branch, num_leaves, offspring);
        else if (node->child_type[i] == LEAF)
            offspring[(*num_leaves)++] = child->leaf;
    }
}

BOOLEAN NotOffspring(particle, node)
int particle;
NODE_T *node;
{
    /*
     * Returns TRUE if "particle" is NOT a leaf in "node" or its children.
     * Otherwise returns FALSE.
     *
     */

    int i;
    CHILD_T *child;
    BOOLEAN not_offspring = TRUE;

    for (i = 0; i < MAX_NUM_CHILDREN; i++) {
        child = &node->child[i];
        if (node->child_type[i] == BRANCH) {
            if (!(not_offspring = NotOffspring(particle, child->branch)))
                break;
        }
        else if (node->child_type[i] == LEAF && child->leaf == particle) {
            not_offspring = FALSE;
            break;
        }
    }
    return not_offspring;
}

NODE_T *Node(tree_index)
int tree_index;
{
    /* Returns node corresponding to "tree_index" (NULL if not found) */

    NODE_T *node = NULL;

    if (ERROR_CHECK && tree_index < 0) {
        (void) sprintf(ErrorStr, "tree index = %i", tree_index);
        Error(WARNING2, "Node():  negative index (overflow?).", ErrorStr);
    }

    search_for_node(tree_index, Root, &node);

    return node;
}

static void search_for_node(tree_index, search_node, node)
int tree_index;
NODE_T *search_node, **node;
{
    /* Recursive search routine for Node() */

    int i;

    if (search_node→tree_index == tree_index)
        *node = search_node;
    else
        for (i = 0; i < MAX_NUM_CHILDREN; i++) {
            if (search_node→child_type[i] == BRANCH)
                search_for_node(tree_index, search_node→child[i].branch, node);
            if (*node ≠ NULL)
                break;
        }
}

/* tree_util.c */
```

B.1.20 update_tree.c

The last source file contains all the routines for calculating the multipole moments of tree cells. There are two sets of routines, one for fast moment calculation immediately after tree construction, and the other for updates at later times. A call to `CalcTreeMoments()` invokes the fast calculation, while updates are performed through `UpdateBranchMoments()`. An update consists of adding or subtracting the contribution of a given particle to a node's multipole moments, depending on whether the particle is being inserted into or removed from the tree. The monopole and quadrupole of a branch can be updated separately as well using `UpdateMonopole()` and `UpdateQuadrupole()`. These functions are called by the tree force routines if the corresponding moment is found to be out of date. The functions are recursive because the children of a node may need to be updated before the node itself can be updated. Liberal use is made of long function names to distinguish between the various operations that need to be performed when calculating tree moments, such as adding the centre-of-mass contribution of a particle to a node (`add_leaf_to_monopole()`) or predicting the centre-of-mass position, velocity, force, and first derivative (`pred_branch_mono()`). The routines that assign node time-steps and determine the maximum extension of a node (cf. §3.4.5) are also found in this file.

```
/*
 * update_tree.c - DCR 91-05-10
 * ==============================
 * Routines for calculating tree multipole moments.
 *
 * Global functions: CalcTreeMoments(), UpdateBranchMoments(), UpdateMonopole(),
 *     UpdateQuadrupole().
 *
 */

/* Include files */

#include "box_tree.h"

/* Local functions */

static void
    add_leaf_to_monopole(),
    add_branch_to_monopole(),
    add_branch_quadrupole(),
    pred_leaf_mono(),
    pred_branch_mono(),
    pred_and_add_quad(),
    add_to_quadrupole(),
    set_mono_time_step(),
    set_quad_time_step(),
    get_max_size();

/* End of preamble */

void CalcTreeMoments(branch)
BRANCH_T *branch;
{
    /*
     * Calculates monopole (and quadrupole if desired) moments of "branch",
     * calculating moments of its children as needed. Thus to recompute
     * all tree moments, call this routine with Root as its argument. Note
     * that it is assumed all moments are ZEROED before call. Also note
     * that no prediction is performed - all data are assumed to be up to
     * date. Computation time is reduced by these shortcuts.
     *
     */

    int i;
    DATA_T *ptr;
    CHILD_T *child;
    BRANCH_T *child_branch;
    double norm;

    /* Calculate monopole contribution of children */

    for (i = 0; i < MAX_NUM_CHILDREN; i++) {
        child = &branch→child[i];
        if (branch→child_type[i] == LEAF)
            add_leaf_to_monopole(branch, child→leaf);
        else if (branch→child_type[i] == BRANCH) {
            CalcTreeMoments(child→branch);
        add_branch_to_monopole(branch, child→branch);
    }
}

/* Error check */

if (ERROR_CHECK) {
    if (branch→num_leaves < 1) {
        (void) sprintf(ErrorStr, "%s", NodeInfo(branch));
        Error(FATAL, "CalcTreeMoments():  Branch has no leaves.", ErrorStr);
    }
    if (branch→num_leaves == 1 && branch ≠ Root) {
        (void) sprintf(ErrorStr, "%s", NodeInfo(branch));
        Error(FATAL, "CalcTreeMoments():  Branch has only one leaf.",
            ErrorStr);
    }
    if (branch→mass == 0) {
        (void) sprintf(ErrorStr, "%s", NodeInfo(branch));
        Error(FATAL, "CalcTreeMoments():  Branch has zero mass.", ErrorStr);
    }
}

/* Divide through by branch mass to obtain monopole moment */

norm = 1 / branch→mass;

branch→pos[0] *= norm;
branch→pos[1] *= norm;
branch→pos[2] *= norm;

if (TreePar.pred_mono) {
    int k;

    for (k = 0; k < NUM_PHYS_DIM; k++) {
        branch→pos0[k] = branch→pos[k];
        branch→vel[k] *= norm;
        branch→f[k] *= norm;
        branch→f_dot[k] *= norm;
    }
}

/* Set update time and time-step */

branch→mt0 = Clock.time;

if (TreePar.check_update_times)
    set_mono_time_step(branch);

/* Check node size */

get_max_size(branch);

/* Return now if not using quadrupole */

if (!TreePar.use_quad)
    return;

/* Otherwise add contribution of children to quadrupole */

for (i = 0; i < MAX_NUM_CHILDREN; i++) {
        child = &branch→child[i];
        if (branch→child_type[i] == LEAF) {
            ptr = Data[child→leaf];
            add_to_quadrupole(branch, ptr→mass, ptr→pos, ptr→vel, ptr→f,
                ptr→f_dot);
        }
        else if (branch→child_type[i] == BRANCH) {
            child_branch = child→branch;
            add_branch_quadrupole(branch, child_branch);
            add_to_quadrupole(branch, child_branch→mass, child_branch→pos,
                child_branch→vel, child_branch→f, child_branch→f_dot);
        }
    }

    /* Save start-of-step quadrupole moment tensor */

    if (TreePar.pred_quad)
        for (i = 0; i < NUM_QUAD_ELEM; i++)
            branch→q_mom0[i] = branch→q_mom[i];

    /* Set update time and time-step */

    branch→qt0 = Clock.time;

    if (TreePar.check_update_times)
        set_quad_time_step(branch);
}

static void add_leaf_to_monopole(parent, leaf)
BRANCH_T *parent;
LEAF_T leaf;
{
    /* Adds contribution of "leaf" to monopole of "parent" */

    DATA_T *ptr = Data[leaf];
    double particle_mass = ptr→mass;

    /* Increment number of leaves and add to mass */

    ++parent→num_leaves;
    parent→mass += particle_mass;

    /* Add contribution */

    parent→pos[0] += particle_mass * ptr→pos[0];
    parent→pos[1] += particle_mass * ptr→pos[1];
    parent→pos[2] += particle_mass * ptr→pos[2];

    if (TreePar.pred_mono) {
        int k;

        for (k = 0; k < NUM_PHYS_DIM; k++) {
            parent→vel[k] += particle_mass * ptr→vel[k];
            parent→f[k] += particle_mass * ptr→f[k];
            parent→f_dot[k] += particle_mass * ptr→f_dot[k];
        }
    }
}

static void add_branch_to_monopole(parent, branch)
```

```
BRANCH_T *parent, *branch;
{
    /* Adds contribution of "branch" to "parent" monopole */

    double branch_mass = branch→mass;

    parent→num_leaves += branch→num_leaves;
    parent→mass += branch_mass;

    parent→pos[0] += branch_mass * branch→pos[0];
    parent→pos[1] += branch_mass * branch→pos[1];
    parent→pos[2] += branch_mass * branch→pos[2];

    if (TreePar.pred_mono) {
        int k;

        for (k = 0; k < NUM_PHYS_DIM; k++) {
            parent→vel[k] += branch_mass * branch→vel[k];
            parent→f[k] += branch_mass * branch→f[k];
            parent→f_dot[k] += branch_mass * branch→f_dot[k];
        }
    }
}

static void add_branch_quadrupole(parent, branch)
BRANCH_T *parent, *branch;
{
    /* Adds contribution of "branch" to "parent" quadrupole */

    int i;

    for (i = 0; i < NUM_QUAD_ELEM; i++)
        parent→q_mom[i] += branch→q_mom[i];

    if (TreePar.pred_quad)
        for (i = 0; i < NUM_QUAD_ELEM; i++) {
            parent→q_dot[i] += branch→q_dot[i];
            parent→q_2dot[i] += branch→q_2dot[i];
            parent→q_3dot[i] += branch→q_3dot[i];
        }
}

void UpdateBranchMoments(op, particle, branch)
int op, particle;
BRANCH_T *branch;
{
    /*
     * Adds or subtracts, depending on "op", contribution of "particle"
     * to "branch" monopole (and quadrupole if desired). Note that, when
     * adding, "particle" should already be in tree.
     *
     */

    /* Error check */

    if (ERROR_CHECK && ((op ≠ ADDM && op ≠ SUBM) || particle < 0 ||
            particle ≥ MAX_NUM_PARTICLES || branch == (BRANCH_T *) NULL))
        Error(FATAL, "UpdateBranchMoments():  Invalid argument(s).", "");

    /* Adjust number of leaves and branch mass */
    branch→num_leaves += op;
    branch→mass += op ∗ Data[particle]→mass;

    /∗ Update monopole data ∗/

    UpdateMonopole(branch);

    /∗ If desired, also update quadrupole ∗/

    if (TreePar.use_quad)
        UpdateQuadrupole(branch);
}

void UpdateMonopole(branch)
BRANCH_T ∗branch;
{
    /∗ Recalculates "branch" monopole data from children ∗/

    int i, k;
    CELL_T child_type;
    CHILD_T ∗child;
    double mass, pos[NUM_PHYS_DIM], vel[NUM_PHYS_DIM], f[NUM_PHYS_DIM],
        f_dot[NUM_PHYS_DIM], norm, ∗ppos;

    /∗ Zero branch monopole data ∗/

    ZERO(branch→pos);
    ZERO(branch→vel);
    ZERO(branch→f);
    ZERO(branch→f_dot);

    /∗ Predict monopoles (c-o-m pos'ns) of children and add contributions ∗/

    for (i = 0; i < MAX_NUM_CHILDREN; i++) {
        child_type = branch→child_type[i];
        if (child_type ≠ EMPTY) {
            child = &branch→child[i];
            if (TreePar.pred_mono) {
                if (child_type == LEAF)
                    pred_leaf_mono(child→leaf, &mass, pos, vel, f, f_dot);
                else
                    pred_branch_mono(child→branch, &mass, pos, vel, f, f_dot);
                for (k = 0; k < NUM_PHYS_DIM; k++) {
                    branch→pos[k] += mass ∗ pos[k];
                    branch→vel[k] += mass ∗ vel[k];
                    branch→f[k] += mass ∗ f[k];
                    branch→f_dot[k] += mass ∗ f_dot[k];
                }
            }
            else {
                if (child_type == LEAF) {
                    mass = Data[child→leaf]→mass;
                    ppos = Data[child→leaf]→pos;
                }
                else {
                    mass = child→branch→mass;
                    ppos = child→branch→pos;
                }
                branch→pos[0] += mass ∗ ppos[0];
                branch→pos[1] += mass * ppos[1];
                branch→pos[2] += mass * ppos[2];
            }
        }
    }

    /* Divide through by branch mass */

    norm = (branch→mass == 0 ? 0 : 1 / branch→mass);                    /* For N=2 case */

    branch→pos[0] * = norm;
    branch→pos[1] * = norm;
    branch→pos[2] * = norm;

    if (TreePar.pred_mono)
        for (k = 0; k < NUM_PHYS_DIM; k++) {
            branch→pos0[k] = branch→pos[k];
            branch→vel[k] * = norm;
            branch→f[k] * = norm;
            branch→f_dot[k] * = norm;
        }

    /* Set update time and time-step */

    branch→mt0 = Clock.time;

    if (TreePar.check_update_times)
        set_mono_time_step(branch);

    /* Check node size */

    get_max_size(branch);
}

static void pred_leaf_mono(leaf, mass, pos, vel, f, f_dot)
LEAF_T leaf;
double *mass, *pos, *vel, *f, *f_dot;
{
    /*
     * Predicts "leaf" monopole data to high or low order as desired.
     * Note that for maximum efficiency, the particle prediction routines
     * PredictPosAndVelHi(), PREDICT_POS_LO(), and PREDICT_VEL_LO() are
     * not called, but instead written out explicitly below.
     *
     */

    int k;
    DATA_T *ptr = Data[leaf];
    double dt = Clock.time - ptr→t0;

    /* Error check */

    if (ERROR_CHECK && APPROX_GT(dt, ptr→time_step)) {
        (void) sprintf(ErrorStr, "%i (%i), t %g, %.16f > %.16f", leaf,
            ptr→orig_index, TIME, dt, ptr→time_step);
        Error(WARNING2, "pred_leaf_mono():  Particle out of date.", ErrorStr);
    }

    /* Calculate monopole data due to leaf */
    *mass = ptr→mass;

    if (TreePar.use_high_order) {
        double dtp = (ptr→t0 - ptr→t1) + (ptr→t0 - ptr→t2), f2dotk;

        if ((ptr→pos_status == UN_PRED && ptr→vel_status == UN_PRED) ||
                (ptr→pos_status == LO_PRED && ptr→vel_status == LO_PRED)) {
            for (k = 0; k < NUM_PHYS_DIM; k++) {
                f2dotk = ptr→d3[k] * dtp + ptr→d2[k];
                ptr→pos[k] = ((((0.05 * ptr→d3[k] * dt + OneTwelfth *
                        f2dotk) * dt + ptr→f_dot[k]) * dt + ptr→f[k]) * dt +
                        ptr→vel0[k]) * dt + ptr→pos0[k];
                ptr→vel[k] = (((0.25 * ptr→d3[k] * dt + OneThird *
                        f2dotk) * dt + 3 * ptr→f_dot[k]) * dt +
                        2 * ptr→f[k]) * dt + ptr→vel0[k];
            }
            ptr→pos_status = ptr→vel_status = HI_PRED;
        }
        for (k = 0; k < NUM_PHYS_DIM; k++) {
            pos[k] = ptr→pos[k];
            vel[k] = ptr→vel[k];
            f2dotk = ptr→d3[k] * dtp + ptr→d2[k];
            f[k] = (0.5 * (ptr→d3[k] * dt + f2dotk) * dt +
                3 * ptr→f_dot[k]) * dt + ptr→f[k];
            f_dot[k] = (0.5 * ptr→d3[k] * dt + OneThird * f2dotk) * dt +
                ptr→f_dot[k];
        }
    }
    else {
        if (ptr→pos_status == UN_PRED) {
            for (k = 0; k < NUM_PHYS_DIM; k++)
                ptr→pos[k] = ((ptr→f_dot[k] * dt + ptr→f[k]) * dt +
                    ptr→vel0[k]) * dt + ptr→pos0[k];
            ptr→pos_status = LO_PRED;
        }
        if (ptr→vel_status == UN_PRED) {
            for (k = 0; k < NUM_PHYS_DIM; k++)
                ptr→vel[k] = (3 * ptr→f_dot[k] * dt + 2 * ptr→f[k]) *
                    dt + ptr→vel0[k];
            ptr→vel_status = LO_PRED;
        }
        for (k = 0; k < NUM_PHYS_DIM; k++) {
            pos[k] = ptr→pos[k];
            vel[k] = ptr→vel[k];
            f[k] = 3 * ptr→f_dot[k] * dt + ptr→f[k];
            f_dot[k] = ptr→f_dot[k];
        }
    }
}

static void pred_branch_mono(branch, mass, pos, vel, f, f_dot)
BRANCH_T *branch;
double *mass, *pos, *vel, *f, *f_dot;
{
    /* Predicts "branch" monopole data (low order) */

    int k;
    double dt = Clock.time - branch→mt0;

    /* Check if monopole needs updating first */
    if (TreePar.check_update_times && dt > branch→mts) {
        UpdateMonopole(branch);
        ++Counter[TOTAL_MONO_UPDATES];
        ++Counter[RECUR_MONO_UPDATES];
        dt = 0;
    }

    /* Calculate monopole data due to branch */

    *mass = branch→mass;

    for (k = 0; k < NUM_PHYS_DIM; k++) {
        pos[k] = ((branch→f_dot[k] * dt + branch→f[k]) * dt +
            branch→vel[k]) * dt + branch→pos0[k];
        vel[k] = (3 * branch→f_dot[k] * dt + 2 * branch→f[k]) * dt +
            branch→vel[k];
        f[k] = 3 * branch→f_dot[k] * dt + branch→f[k];
        f_dot[k] = branch→f_dot[k];
    }
}

void UpdateQuadrupole(branch)
BRANCH_T *branch;
{
    /* Recalculates "branch" quadrupole data from children */

    int i;
    CELL_T child_type;
    CHILD_T *child;
    double mass, pos[NUM_PHYS_DIM], vel[NUM_PHYS_DIM], f[NUM_PHYS_DIM],
        f_dot[NUM_PHYS_DIM], *ppos;

    /* Zero quadrupole moments of branch */

    for (i = 0; i < NUM_QUAD_ELEM; i++)
        branch→q_mom[i] = branch→q_dot[i] = branch→q_2dot[i] =
            branch→q_3dot[i] = 0.0;

    /*
     * Predict monopoles of children and quadrupoles of any branches and
     * add contributions to quadrupole of this branch. Note that together
     * add_to_quadrupole() and pred_and_add_quad() constitute the parallel
     * axis theorem for quadrupole moments (e.g. Hernquist 1987).
     *
     */

    for (i = 0; i < MAX_NUM_CHILDREN; i++) {
        child_type = branch→child_type[i];
        if (child_type ≠ EMPTY) {
            child = &branch→child[i];
            if (TreePar.pred_quad) {
                if (child_type == LEAF)
                    pred_leaf_mono(child→leaf, &mass, pos, vel, f, f_dot);
                else {
                    pred_branch_mono(child→branch, &mass, pos, vel, f, f_dot);
                    pred_and_add_quad(branch, child→branch);
                }
                add_to_quadrupole(branch, mass, pos, vel, f, f_dot);
            }
```

```
                else {
                    if (child_type == LEAF) {
                        mass = Data[child→leaf]→mass;
                        ppos = Data[child→leaf]→pos;
                    }
                    else {
                        mass = child→branch→mass;
                        ppos = child→branch→pos;
                    }
                    /* (vel, f, f_dot ignored) */
                    add_to_quadrupole(branch, mass, ppos, vel, f, f_dot);
                }
            }
        }

        /* Save start-of-step quadrupole */

        if (TreePar.pred_quad)
            for (i = 0; i < NUM_QUAD_ELEM; i++)
                branch→q_mom0[i] = branch→q_mom[i];

        /* Set update time and time-step */

        branch→qt0 = Clock.time;

        if (TreePar.check_update_times)
            set_quad_time_step(branch);
}

static void pred_and_add_quad(parent, branch)
BRANCH_T *parent, *branch;
{
        /* Predicts "branch" quadrupole data and adds it in to "parent" data */

        int i;
        double dt = Clock.time - branch→qt0;

        /* Check if quadrupole needs updating first */

        if (TreePar.check_update_times && dt > branch→qts) {
            UpdateQuadrupole(branch);
            ++Counter[TOTAL_QUAD_UPDATES];
            ++Counter[RECUR_QUAD_UPDATES];
            dt = 0;
        }

        /* Add in child branch contribution to quadrupole */

        for (i = 0; i < NUM_QUAD_ELEM; i++) {
            parent→q_mom[i] += ((branch→q_3dot[i] * dt + branch→q_2dot[i]) * dt +
                branch→q_dot[i]) * dt + branch→q_mom0[i];
            parent→q_dot[i] += (3 * branch→q_3dot[i] * dt +
                2 * branch→q_2dot[i]) * dt + branch→q_dot[i];
            parent→q_2dot[i] += 3 * branch→q_3dot[i] * dt + branch→q_2dot[i];
            parent→q_3dot[i] += branch→q_3dot[i];
        }
}

static void add_to_quadrupole(branch, mass, pos, vel, f, f_dot)
BRANCH_T *branch;
```

```
double mass, *pos, *vel, *f, *f_dot;
{
    /* Adds contribution of "mass" ("pos", "vel", etc.) to "branch" quad */

    int k, m, n;
    double rel_pos[NUM_PHYS_DIM], rel_vel[NUM_PHYS_DIM], rel_f[NUM_PHYS_DIM],
        rel_f_dot[NUM_PHYS_DIM], w0, w1, w2, w3, m2 = 2 * mass, m3 = 3 * mass;

    /* Fast calculation if there is no quadrupole prediction */

    if (!TreePar.pred_quad) {
        SUB(pos, branch→pos, rel_pos);
        w0 = DOT(rel_pos, rel_pos);
        branch→q_mom[0] += mass * (3 * SQ(rel_pos[0]) - w0);
        branch→q_mom[1] += m3 * rel_pos[0] * rel_pos[1];
        branch→q_mom[2] += m3 * rel_pos[0] * rel_pos[2];
        branch→q_mom[3] += mass * (3 * SQ(rel_pos[1]) - w0);
        branch→q_mom[4] += m3 * rel_pos[1] * rel_pos[2];
        return;
    }

    /*
     * Calculate quantities relative to branch (recall f and f_dot contain
     * hidden factors of 1/2 and 1/6, respectively).
     *
     */

    SUB(pos, branch→pos, rel_pos);
    SUB(vel, branch→vel, rel_vel);
    SUB(f, branch→f, rel_f);
    SUB(f_dot, branch→f_dot, rel_f_dot);

    w0 = mass * DOT(rel_pos, rel_pos);
    w1 = m2 * DOT(rel_pos, rel_vel);
    w2 = mass * (DOT(rel_vel, rel_vel) + 2 * DOT(rel_pos, rel_f));
    w3 = m2 * (DOT(rel_pos, rel_f_dot) + DOT(rel_vel, rel_f));

    /* Add in contribution to branch quadrupole (optimize?) */

    for (k = 0; k < MIN(2, NUM_PHYS_DIM); k++)
        for (m = k; m < NUM_PHYS_DIM; m++) {
            n = k * (NUM_PHYS_DIM - 1) + m;
            branch→q_mom[n] += m3 * rel_pos[k] * rel_pos[m];
            branch→q_dot[n] += m3 * (rel_vel[k] * rel_pos[m] +
                rel_pos[k] * rel_vel[m]);
            branch→q_2dot[n] += m3 * (rel_f[k] * rel_pos[m] +
                rel_vel[k] * rel_vel[m] + rel_pos[k] * rel_f[m]);
            branch→q_3dot[n] += m3 * (rel_f_dot[k] * rel_pos[m] +
                rel_f[k] * rel_vel[m] + rel_vel[k] * rel_f[m] +
                rel_pos[k] * rel_f_dot[m]);
            if (k == m) {
                branch→q_mom[n] -= w0;
                branch→q_dot[n] -= w1;
                branch→q_2dot[n] -= w2;
                branch→q_3dot[n] -= w3;
            }
        }
}

#define MONO_DT_STAB 2                /* Maximum monopole time-step increase factor */
```

```
static void set_mono_time_step(branch)
BRANCH_T *branch;
{
    /* Sets update time and step for "branch" monopole */

    int k;
    double w0, w1, w2, w3, v2, f2, ft, fb;

    w0 = branch->size;
    w1 = w2 = w3 = v2 = f2 = 0;

    for (k = 0; k < NUM_PHYS_DIM; k++) {
        w1 += ABS(branch->vel[k]);
        w2 += ABS(branch->f[k]);
        w3 += ABS(branch->f_dot[k]);
        v2 += SQ(branch->vel[k]);
        f2 += SQ(branch->f[k]);
    }

    if ((ft = w0 * w2 + v2) == 0 || (fb = w1 * w3 + f2) == 0) {
        (void) sprintf(ErrorStr, "%s time %g", NodeInfo(branch), TIME);
        Error(WARNING2, "set_mono_time_step(): Using max mono step.", ErrorStr);
        branch->mts = (RunPar.max_time_step ? RunPar.max_time_step : 1);
    }
    else
        branch->mts =
            MIN(MONO_DT_STAB * branch->mts, TreePar.mtsc * sqrt(ft / fb));
}

#undef MONO_DT_STAB

#define QUAD_DT_STAB 2                  /* Maximum quadrupole time-step increase factor */

static void set_quad_time_step(branch)
BRANCH_T *branch;
{
    /* Sets update time and time-step for "branch" quadrupole */

    int i;
    double w0, w1, w2, w3, q12, q22, ft, fb;

    w0 = w1 = w2 = w3 = q12 = q22 = 0;

    for (i = 0; i < NUM_QUAD_ELEM; i++) {
        w0 += ABS(branch->q_mom[i]);
        w1 += ABS(branch->q_dot[i]);
        w2 += ABS(branch->q_2dot[i]);
        w3 += ABS(branch->q_3dot[i]);
        q12 += SQ(branch->q_dot[i]);
        q22 += SQ(branch->q_2dot[i]);
    }

    if ((ft = w0 * w2 + q12) == 0 || (fb = w1 * w3 + q22) == 0) {
        (void) sprintf(ErrorStr, "%s time %g", NodeInfo(branch), TIME);
        Error(WARNING2, "set_quad_time_step(): Using max quad step.", ErrorStr);
        branch->qts = (RunPar.max_time_step ? RunPar.max_time_step : 1);
    }
    else
        branch->qts =
            MIN(QUAD_DT_STAB * branch→qts, TreePar.qtsc * sqrt(ft / fb));
}

#undef QUAD_DT_STAB

static void get_max_size(node)
NODE_T *node;
{
    /*
     * Assigns max size to "node" based on actual position of its children.
     * Note that branch extensions are defined around the centre of mass
     * of each node, so if the centre of mass of Root is not at the origin
     * (for example), the root node will be extended to encompass the whole
     * system from its non-zero centre posision.
     *
     */

    int i;
    DATA_T *ptrd;
    NODE_T *ptrn;
    CHILD_T *child;
    double ext, max_ext = 0;

    /*
     * Maximum sizes are currently only used in the rotating frame
     * (preferably flattened). However, this presupposes that the
     * particles are well-behaved and do not undergo extreme changes
     * in position or velocity over short intervals.
     *
     */

    if (!ROTATING_FRAME) {
        node→max_ext = node→half_size;
        node→max_size = node→size;
        node→max_size_sq = SQ(node→size);
        return;
    }

    for (i = 0; i < MAX_NUM_CHILDREN; i++) {
        child = &node→child[i];
        switch (node→child_type[i]) {
        case EMPTY:
            continue;
        case LEAF:
            ptrd = Data[child→leaf];
            PREDICT_POS_LO(ptrd);
            ext = ABS(ptrd→pos[1] - node→pos[1]);
            max_ext = MAX(max_ext, ext);
            ext = ABS(ptrd→pos[2] - node→pos[2]);
            max_ext = MAX(max_ext, ext);
            break;
        case BRANCH:
            ptrn = child→branch;
            ext = ABS(ptrn→pos[1] - node→pos[1]) + ptrn→max_ext;
            max_ext = MAX(max_ext, ext);
            ext = ABS(ptrn→pos[2] - node→pos[2]) + ptrn→max_ext;
            max_ext = MAX(max_ext, ext);
        }
    }
    node→max_ext = max_ext;
    node→max_size = MAX(2 * max_ext, node→size);
    node→max_size_sq = SQ(node→max_size);

    if (MONITOR && VERY_VERBOSE && node→max_size > node→size)
        (void) printf("MONITOR -- ext node %s (time %g max size %g)\n",
            NodeInfo(node), TIME, node→max_size);
}

/* update_tree.c */
```

B.2 rdpar

The `rdpar` parser was written to replace a more cumbersome version of the same name available in the public domain (author unknown). The parser consists of a set of functions that can be called by an application to read parameters from a file with ease. The use of `rdpar` is not restricted to `box_tree`. Indeed, `make_movie`, `stats_read`, and `dat_read` all make use of it. The code itself contains a lengthy comment at the beginning that explains the calling syntax. There is a small header file as well which must be included by applications that wish to use the parsing routines. Note that `rdpar` has its own error handling routines.

B.2.1 rdpar.h

```
/* rdpar.h - Definitions for rdpar routines DCR 93-03-23 */

#define NEND -1                              /* Flags for terminating multiple reads */
#define NEND_LABEL "NULL"

#define MAX_RDPAR_SIZE 1000              /* Maximum number of lines in parameter file */

#ifndef MAX_STR_LEN                                      /* Maximum string length */
# define MAX_STR_LEN 256
#endif

extern void                                                      /* External voids */
        OpenPar(),
        ReadInt(),
        ReadLng(),
        ReadFlo(),
        ReadDbl(),
        ReadStr(),
        ClosePar();

extern int                                                        /* External ints */
        ReadNInt(),
        ReadNLng(),
        ReadNFlo(),
        ReadNDbl(),
        ReadNStr();
```

B.2.2 rdpar.c

```
/*
```

```
 * rdpar ver 1.1 - DCR 93-03-23
 * ================================
 * A parameter file parsing utility based on the original RDPAR.
 *
 * Modifications by David Earn:
 *  7 May 1993: added __STDIO_H_SEEN and __STDLIB_H_SEEN lines for Convex cc
 *
 * SYNOPSIS:
 *
 * Call OpenPar(char *filename) to prepare a parameter file for reading, and
 * ClosePar(void) when you're done. The parameter file should be organized
 * with parameter "labels" or key words at the start of each line, followed
 * by the data, separated with whitespace (no commas). Comments may be
 * added anywhere by prefacing with "!" or "#" (the remainder of the line
 * is ignored). Currently, "!" and "#" cannot be used inside character
 * strings. Single integers, long integers, floats, doubles, and strings can
 * be read by calling:
 *     void ReadInt(char *label, int *myint);
 *     void ReadLng(char *label, long int *mylng);
 *     void ReadFlo(char *label, float *myflo);
 *     void ReadDbl(char *label, double *mydbl);
 *     void ReadStr(char *label, char *mystr, int maxstrlen);
 * where "label" is the keyword(s) as used in the parameter file. Note that
 * strings must be given a maximum length. For ReadStr(), strings need not
 * be bracketed by quotes (") (which are removed if present); apostrophes
 * (') are left alone. To read more than one item of a given type, use:
 *     int ReadNInt(char *label, int *myintarray, int numelem);
 *     int ReadNLng(char *label, long int *mylngarray, int numelem);
 *     int ReadNFlo(char *label, float *myfloarray, int numelem);
 *     int ReadNDbl(char *label, double *mydblarray, int numelem);
 *     int ReadNStr(char *label, char **mystrarray, int numelem, int maxstrlen);
 * where "numelem" is the number of elements of each type to be read. These
 * should be on the same line as the key words (up to 256 characters),
 * separated by whitespace. Strings MUST be delimited by quotes in this case.
 * The ReadN commands return the constant NEND (-1) if the keyword
 * NEND_LABEL (the string "NULL", without quotes) appears in place of a
 * data item. This is useful for reading multiple data sets with the same
 * keywords, e.g.,
 *     x, y, z positions              1 2 0
 *     x, y, z positions              3 4 0
 *     x, y, z positions              NULL NULL NULL
 * At the moment it is primarily the responsibility of the programmer to
 * ensure that valid arguments are passed to the rdpar routines. Further
 * functionality may be added in future versions. Comments welcome!
 *
 * USAGE:
 *
 * Build the object code by executing the Makefile (type "make").
 * To use the rdpar routines, add the object ("rdpar.o") to the argument
 * list of your program compile command. Be sure to #include "rdpar.h".
 * To compile a test program, use "make test".
 *
 */

/*LINTLIBRARY*/                          /* Disable lint complaints of unused functions */

/* Copyright notice */

#include "COPYRIGHT"
```

```
/* Header files */

#include "rdpar.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifndef __STDLIB_H_SEEN                                          /* Convex */
# include <malloc.h>
#endif

/* Some #define's */

#define VERBOSE 0                                     /* Non-zero to switch on */

#define EOS '\0'                                  /* String termination marker */
#define TAB '\011'                              /* TAB character (whitespace) */
#define SPACE ' '                                     /* Space (whitespace) */
#define COM1 '!'                                          /* Comment marker */
#define COM2 '#'                                          /* Comment marker */
#define QUOTE '\"'                                        /* String delimiter */
#define CR '\n'                                           /* Carriage return */

/* Useful macros */

#define MIN(x,y) ((x) < (y) ? (x) : (y))                     /* Minimum of x & y */
#define MAX(x,y) ((x) > (y) ? (x) : (y))                     /* Maximum of x & y */

/* Local variables */

FILE *fp;                                                /* Pointer to par file */
char *data[MAX_RDPAR_SIZE];                         /* Storage for file in memory */
int line;                                           /* Counter/total no. of lines */

/* Local routines */

static void
        strip(),
        stripSpace(),
        stripLeadingSpace(),
        stripTrailingSpace(),
        cutToSpace(),
        error();

static int
        stripQuotes(),
        my_cindex();

static char
        *findLabel(),
        *my_index(),
        *my_sindex();

/* Non-ANSI C may not declare these... */

#ifndef _STDIO_H                                                    /* Sun */
#ifndef __STDIO_H__                                     /* Silicon Graphics */
#ifndef _H_STDIO                                          /* IBM RISC 6000 */
#ifndef __STDIO_H_SEEN                                             /* Convex */

extern int fclose(), fprintf(), printf();
```

```
#endif
#endif
#endif
#endif

/* Include main() for testing if desired */

#ifdef TEST
int main()
{
        /* The following bit of code provides a test facility */

        int dumint, iarray[2];
        long int dumlng, larray[1];
        float dumflo, farray[5];
        double dumdbl, darray[3];
        char dumstr[MAX_STR_LEN], *sarray[2];

        /* Open par file */

        OpenPar("rptest.par");                    /* rptest.par is supplied with the code */

        /* Test single reads */

        ReadStr("Comment line", dumstr, MAX_STR_LEN);
        (void) printf("ReadStr() returned \"%s\".\n", dumstr);

        ReadInt("Random number seed", &dumint);
        (void) printf("ReadInt() returned %i\n", dumint);

        ReadLng("Long int", &dumlng);
        (void) printf("ReadLng() returned %li\n", dumlng);

        ReadFlo("Box size", &dumflo);
        (void) printf("ReadFlo() returned %f\n", dumflo);

        /* Note that the second value can be obtained by a repeat read */

        ReadFlo("Box size", &dumflo);
        (void) printf("ReadFlo() returned %f (second read)\n", dumflo);

        ReadDbl("Pi", &dumdbl);
        (void) printf("ReadDbl() returned %.9f\n", dumdbl);

        /* Test multiple reads */

        while (ReadNInt("Track particle", iarray, 2) ≠ NEND)
            (void) printf("ReadNInt() returned %i %i\n", iarray[0], iarray[1]);

        while (ReadNLng("Just one long", larray, 1) ≠ NEND)
            (void) printf("ReadNLng() returned %li\n", larray[0]);

        while (ReadNFlo("Many floats", farray, 5) ≠ NEND)
            (void) printf("ReadNFlo() returned %.1f %.1f %.1f %.1f %.1f\n",
                        farray[0], farray[1], farray[2], farray[3], farray[4]);

        while (ReadNDbl("Double test", darray, 3) ≠ NEND)
            (void) printf("ReadNDbl() returned %g %g %g\n", darray[0],
                darray[1], darray[2]);
        sarray[0] = (char *) malloc(MAX_STR_LEN);
        sarray[1] = (char *) malloc(MAX_STR_LEN);

        while (ReadNStr("String test", sarray, 2, 256) ≠ NEND)
            (void) printf("ReadNStr() returned \"%s\" \"%s\"\n", sarray[0],
                sarray[1]);

        /* Close par file */

        ClosePar();

        /* Normal termination */

        return 0;
}
#endif

void OpenPar(fpstr)
char *fpstr;
{
        /* Opens par file and reads lines into memory */

        static int first_call = 1;
        char buf[MAX_STR_LEN], *bp;

        /* Make sure only one file is open at a time */

        if (first_call) {
                fp = NULL;
                first_call = 0;
        }

        if (fp ≠ NULL)
                error("OpenPar():  File already open...close it first.");

        /* Attempt to assign file pointer */

        if ((fp = fopen(fpstr, "r")) == NULL)
                error("OpenPar():  Unable to open parameter file.");

        /* Read in data lines, stripping comments */

        line = 0;

        while (fgets(buf, MAX_STR_LEN, fp) ≠ NULL) {
                bp = buf;
                strip(&bp);
                if (strlen(bp) > 0) {
                        if (line == MAX_RDPAR_SIZE)
                                error("OpenPar():  Max file size exceeded.");
                        data[line] = (char *) malloc((unsigned) strlen(bp) + 1);
                        (void) strcpy(data[line++], bp);
                }
        }

        /* Close par file */

        (void) fclose(fp);
```

```
#if (VERBOSE ≠ 0)
        (void) printf("OpenPar():  %i line(s) read from \"%s\"\n",
                        line, fpstr);
#endif
}

void ClosePar()
{
        /* Resets file pointer and deallocates storage */

        fp = NULL;

        for (--line; line ≥ 0; line--)
                free(data[line]);
}

void ReadInt(label, x)
char *label;
int *x;
{
        /* Reads integer x associated with keyword(s) label */

        *x = atoi(findLabel(label));
}

void ReadLng(label, x)
char *label;
long int *x;
{
        /* Reads long integer x associated with keyword(s) label */

        *x = atol(findLabel(label));
}

void ReadFlo(label, x)
char *label;
float *x;
{
        /* Reads float x associated with keyword(s) label */

        *x = (float) atof(findLabel(label));
}

void ReadDbl(label, x)
char *label;
double *x;
{
        /* Reads double x associated with keyword(s) label */

        *x = atof(findLabel(label));
}

void ReadStr(label, str, len)
char *label, *str;
int len;
{
        /* Reads string str (max length len) associated with keyword(s) label */

        int l;
        char *substr;
        substr = findLabel(label);
        l = stripQuotes(&substr);                    /* Remove quotation marks, if any */
        (void) strncpy(str, substr, MIN(len, l));
}

int ReadNInt(label, x, n)
char *label;
int *x, n;
{
        /* Reads n integers associated with label into array x */

        int i;
        char *substr;

        substr = findLabel(label);

        /* Loop over integers (assume they're all there!) */

        for (i = 0; i < n; i++) {

                /* Check for end of list read */

                if (my_cindex(NEND_LABEL, substr) == 0)
                        return NEND;

                /* Assign value */

                x[i] = atoi(substr);

                /* Skip to next value in string */

                if (i < n - 1) {
                        cutToSpace(&substr);
                        stripLeadingSpace(&substr);
                }
        }

        return 0;
}

int ReadNLng(label, x, n)
char *label;
long int *x;
int n;
{
        /* Reads n long integers associated with label into array x */

        int i;
        char *substr;

        substr = findLabel(label);

        for (i = 0; i < n; i++) {
                if (my_cindex(NEND_LABEL, substr) == 0)
                        return NEND;
                x[i] = atol(substr);
                if (i < n - 1) {
                        cutToSpace(&substr);
                        stripLeadingSpace(&substr);
                }
        }

        return 0;
}

int ReadNFlo(label, x, n)
char *label;
float *x;
int n;
{
        /* Reads n floats associated with label into array x */

        int i;
        char *substr;

        substr = findLabel(label);

        for (i = 0; i < n; i++) {
                if (my_cindex(NEND_LABEL, substr) == 0)
                        return NEND;
                x[i] = (float) atof(substr);
                if (i < n - 1) {
                        cutToSpace(&substr);
                        stripLeadingSpace(&substr);
                }
        }

        return 0;
}

int ReadNDbl(label, x, n)
char *label;
double *x;
int n;
{
        /* Reads n doubles associated with label into array x */

        int i;
        char *substr;

        substr = findLabel(label);

        for (i = 0; i < n; i++) {
                if (my_cindex(NEND_LABEL, substr) == 0)
                        return NEND;
                x[i] = atof(substr);
                if (i < n - 1) {
                        cutToSpace(&substr);
                        stripLeadingSpace(&substr);
                }
        }

        return 0;
}

int ReadNStr(label, str, n, len)
char *label, *str[];
int n, len;
{
	/* Reads n strings (max length len) assoc'd with label into array str */

	int i, l;
	char *substr;

	substr = findLabel(label);

	for (i = 0; i < n; i++) {
		if (my_cindex(NEND_LABEL, substr) == 0)
			return NEND;

		/* Strip off quotation marks around current string */

		l = stripQuotes(&substr);

		/* Get length of string and copy to array */

		l = MIN(len, l);
		(void) strncpy(str[i], substr, l);

		/* Add EOS marker */

		str[i][l - (l == len ? 1 : 0)] = EOS;

		/* Skip to next string */

		if (i < n - 1) {
			substr += l;				/*DEBUG what if l = MAX?*/
			stripLeadingSpace(&substr);
		}
	}

	return 0;
}

static char *findLabel(label)
char *label;
{
	/*
	 * Returns pointer to first data item after label. Note that
	 * the label is then "deleted" from memory, so subsequent searches
	 * for the same label will find the next occurence in the par file.
	 *
	 */

	int l;
	char *substr;

	/* Strip the label, just in case */

	strip(&label);

	/* Scan through the stored file a line at a time */

	for (l = 0; l < line; l++)

		/* Look for a match */

		if ((substr = my_sindex(label, data[l])) ≠ NULL) {
```

```
                        /* Strip space up to the first data item */

                        stripLeadingSpace(&substr);

                        /* Remove the label from mem with the help of an EOS */

                        data[l][0] = EOS;

                        /* Return pointer to data */

                        return substr;
                }

        /* Fatal error if the label is not found */

        {
                char errstr[MAX_STR_LEN];

                (void) sprintf(errstr, "findLabel():  Label \"%s\" not found.",
                                label);
                error(errstr);
        }

        /* The following is to keep lint happy */

        return NULL;
}

static char *my_index(tgt, c)
char c, *tgt;
{
        /*
         * Returns a pointer to the first occurence of character c in string
         * tgt, or NULL if c does not occur in the string. Note that my_index()
         * is not an ANSI C function which is why it is given here explicitly.
         *
         */

        int len, cc;

        /* Automatic failure if tgt has zero length */

        if ((len = strlen(tgt)) == 0)
                return NULL;

        /* Search for the first occurence */

        for (cc = 0; cc < len; cc++)
                if (c == tgt[cc])
                        return (tgt + cc);

        /* No match */

        return NULL;
}

static int my_cindex(src, tgt)
char *src, *tgt;
{
        /*
```

```
         * Returns the numerical position in string tgt where the
         * entire string src first occurs. The code -1 is returned
         * if src does not occur anywhere in tgt.
         *
         */

        int slen, tlen, c, cc;

        /* Automatic failure if src cannot fit inside tgt */

        if ((slen = strlen(src)) > (tlen = strlen(tgt)) || slen == 0)
                return -1;

        /* Do a character-by-character comparison until a match is found */

        for (c = 0; c < tlen - slen + 1; c++) {
                cc = 0;
                while (src[cc] == tgt[c + cc])
                        if (++cc == slen)
                                return c;
        }

        /* No match */

        return -1;
}

static char *my_sindex(src, tgt)
char *src, *tgt;
{
        /*
         * Returns a pointer to the position in tgt at which string src
         * first differs from string tgt.
         *
         */

        int len, c;

        /* Automatic failure if src is longer than tgt */

        if ((len = strlen(src)) > strlen(tgt))
                return NULL;

        /* Search for the first difference */

        for (c = 0; c < len; c++)
                if (src[c] ≠ tgt[c])
                        return NULL;

        /* Return pointer to first differing character */

        return (tgt + c);
}

static void strip(str)
char **str;
{
        /*
         * Removes comments and carriage returns as well as leading
         * and trailing whitespace from string str.
         *
         */

        /*DEBUG can't handle ! or # or \n inside quotes...*/

        char *ptr;

        /* Return if string already empty */

        if (strlen(*str) == 0)
                return;

        /* Stick EOS's in place of comment markers or CR */

        if ((ptr = my_index(*str, COM1)) ≠ NULL)
                *ptr = EOS;

        if ((ptr = my_index(*str, COM2)) ≠ NULL)
                *ptr = EOS;

        if ((ptr = my_index(*str, CR)) ≠ NULL)
                *ptr = EOS;

        /* Remove whitespace */

        stripSpace(str);
}

static void stripSpace(str)
char **str;
{
        /* Removes leading and trailing whitespace */

        if (strlen(*str) == 0)
                return;

        stripLeadingSpace(str);

        stripTrailingSpace(str);
}

static void stripLeadingSpace(str)
char **str;
{
        /* Removes leading whitespace, incrementing str pointer as required */

        if (strlen(*str) == 0)
                return;

        while ((*str)[0] == SPACE || (*str)[0] == TAB)
                ++(*str);
}

static void stripTrailingSpace(str)
char **str;
{
        /* Removes trailing whitespace */

        int l;
        if (strlen(*str) == 0)
                return;

        while ((*str)[l = strlen(*str) - 1] == SPACE || (*str)[l] == TAB)
                (*str)[l] = EOS;
}

static int stripQuotes(str)
char **str;
{
        /*
         * Strips 2 quotes from str, incrementing pointer str if necessary.
         * The length of the delimited string (or portion thereof) is returned.
         *
         */

        int l;
        char *ptr;

        /* Leading quote assumed to be in position 0 */

        if ((*str)[0] == QUOTE)
                ++(*str);

        /* Find next occurence of a quote, and shift string down to cover it */

        l = strlen(*str);

        if ((ptr = my_index(*str, QUOTE)) ≠ NULL) {
                l -= strlen(ptr);
                for (; strlen(ptr) > 0; ++ptr)
                        *ptr = *(ptr + 1);
        }

        /* Return the length of the string that was delimited by quotes */

        return l;
}

static void cutToSpace(str)
char **str;
{
        /* Increments pointer str until the first character is not whitespace */

        if (strlen(*str) == 0)
                return;

        while ((*str)[0] ≠ SPACE && (*str)[0] ≠ TAB)
                ++(*str);
}

static void error(str)
char *str;
{
        /* Fatal error routine; will dump core if possible */

        (void) fprintf(stderr, "\007Rdpar error in %s\n", str);

        abort();
}
```