Mon. Not. R. Astron. Soc. **261,** 396–414 (1993)

A new tree code method for simulation of planetesimal dynamics

Derek C. Richardson[★]

University of Cambridge, Institute of Astronomy, Madingley Road, Cambridge CB3 0HA

Accepted 1992 September 15. Received 1992 August 3

ABSTRACT

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 planetesimals at 1 au from the Sun. 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. In typical runs with a few hundred central particles, the tree code is approximately 2–3 times faster than a recent direct summation method and requires about 1 CPU day on a Sparc IPX workstation to simulate 100 yr of evolution. The average relative force error incurred in such runs is less than 0.2 per cent in magnitude. In general, the CPU time as a function of particle number varies in a way consistent with an $O(N \log N)$ algorithm. 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 – including a comparison with previous work – are presented.

Key words: methods: numerical – celestial mechanics, stellar dynamics – Solar system: formation.

1 INTRODUCTION

The evolution of planetesimals is a subject that has been studied widely in recent years, using both analytical and numerical methods (e.g. Safronov 1969; Greenberg et al. 1978; Wetherill 1980; Nakagawa, Hayashi & Nagazawa 1983; Wetherill & Stewart 1989; Emori, Nakazawa & Ida, in preparation). Despite advances in our understanding of the processes governing protoplanet formation, it is acknowledged that analytical treatments still only give a very rough picture of the true nature of planetesimal dynamics. 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, particle spin – can be taken into account. However, in order to address the most central issues in planetesimal dynamics and planetary formation, numerical simulations of systems with large dynamical range and high spatial resolution are required (Palmer, Lin & Aarseth 1993, hereafter PLA; Aarseth, Lin & Palmer 1993, hereafter ALP).

In order to minimize the computational expense arising from the large number of particles needed to generate such realistic simulations, two important techniques have been devised. The first of these methods, hereafter termed 'box code', was introduced by Wisdom & Tremaine (1988, hereafter 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 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. 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 simulator from $O(N^2)$ to $O(N \log N)$.

[★]E-mail: dcr@mail.ast.cam.ac.uk.

In this paper, a new scheme is introduced that exploits the advantages of both methods with an emphasis on the planetesimal problem (particle sizes ~ 10–100 km). In the new method, a tree structure is imposed on a representative patch of a ring of uniformly distributed planetesimals at 1 au from the Sun. Periodic boundary conditions are applied, and eight 'ghost boxes' are placed around the central patch. A standard N-body integrator (Aarseth 1985) is used to advance the system and the resulting data (velocity dispersions, collision frequencies, mergers, spin distributions, etc.) are recorded for later analysis.

The technical details of the new scheme are presented in Section 3. Emphasis is placed on the problems that result from the unique 'box-tree' configuration, and how these problems can best be eliminated or minimized. First, however, more information regarding the fundamentals of box and tree codes in general is given in Section 2 in order to simplify subsequent discussion. The code performance – including timing and accuracy – is analysed in Section 4. The results are presented in both tabular and graphical form, and demonstrate the considerable savings achieved by the new method. A summary and further comments are given in Section 5.

2 FUNDAMENTALS

2.1 Box code

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. 1). Under these conditions, the equations of motion for the ring particles can be linearized in the following form:

$$\begin{aligned}\ddot{x}&=\mathscr{F}_x+3\Omega^2x+2\Omega\dot{y},\\ \ddot{y}&=\mathscr{F}_y-2\Omega\dot{x},\\ \ddot{z}&=\mathscr{F}_z-\Omega^2z,\end{aligned} \tag{1}$$

where $\boldsymbol{\mathscr{F}}=(\mathscr{F}_x,\mathscr{F}_y,\mathscr{F}_z)$ is the sum of gravitational forces per unit mass due to the other ring particles and $\boldsymbol{\Omega}=(0,0,\Omega)$ is the angular velocity of the reference point about the planet. For a ring system at $a=1$ au from the Sun, the linear velocity of the reference point is $\Omega a=(GM_\odot/a)^{1/2}\approx 30$ km s^{-1}. Usually a system of units is chosen such that $\Omega\equiv 1$.

In order to provide a realistic gravitational perturbation on the central box members by the external ring particles and to allow for collisions at the box boundary, it is necessary to include contributions from the eight patches closest to the central box. To achieve this, note that, in the absence of interparticle gravitational forces, equation (1) is invariant under the transformation

$$(x,y,z)\longrightarrow\left(x+i_xs,\,y-\frac{3}{2}\,i_xs\Omega\,t+i_ys,\,z\right), \tag{2}$$

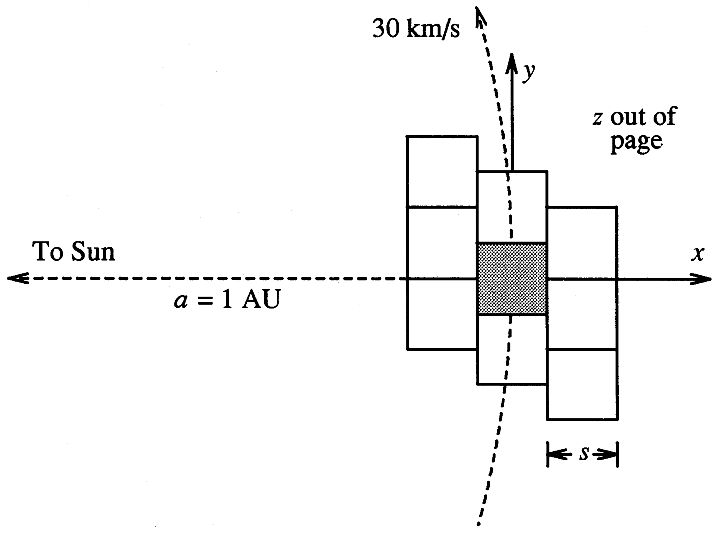

Figure 1. Rotating coordinate system used in the planetesimal simulation. Note that $a\gg s\gg R$, where R is the average particle radius. The central box (shaded region) is surrounded by eight ghost boxes, some of which are shifted in the y-direction to illustrate Keplerian shear.

where i_x and i_y are integers, s is the box size, and t is the time variable. Thus the neighbouring patches or ghost boxes can be chosen to have centres at $(i_x s, -(3/2) i_x s\Omega\, t + i_y s, 0)$, where $i_x, i_y \in \{-1, 0, +1\}$ (not both zero). The particle distribution in each ghost box is identical to the distribution in the central box at all times. However, boxes with $i_x = \pm 1$ experience a shear of magnitude $(3/2) s\Omega\, t$ due to the differential rotation of the ring (Fig. 1). As t increases, multiples of the box size are subtracted from the shearing distance to ensure that the ghost boxes remain close to their original positions with respect to the central box. A constant particle number in each box is achieved (assuming no mergers) by applying boundary conditions in such a way that particles leaving the central box are replaced by their corresponding images entering *from* a ghost box. Thus there are N central particles and $8N$ ghosts in the system at all times.

WT did not include interparticle gravitational forces in their experiments, but were able to show that, for cases where such forces were negligible, their numerical technique agreed well with analytical results. ALP applied the box code to the problem of planetesimal formation around the Sun, in which the computationally expensive interparticle gravitation terms were added. At each integration step for a central box particle, the perturbation was obtained by summing over contributions from all other central box particles and their ghosts. A neighbour scheme and fast square root algorithm were employed to reduce the computation time somewhat. It was found that the numerical results were consistent with analytical predictions made in PLA.

2.2 Tree code

The CPU time required to run a typical N-body simulation using a standard direct method scales as $O(N^2)$ since, in repeated intervals typically much shorter than the dynamical time, $O(N)$ force calculations must be performed, each involving a sum over contributions from $O(N)$ particles (ignoring ghosts). In the present context, reasonably large N ($\sim 10^2$–10^4) is required to examine the evolution of the planetesimal mass spectrum, making numerical computation with a standard direct method unrealistic. BH put forward a hierarchical algorithm that reduces the expense to $O(N \log N)$ for sufficiently large N, but which introduces a small error in the calculated force, on average ~ 1 per cent (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 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).

Before explaining the details of how the force is calculated, it is helpful to define some of the terms that describe components of the tree. A cell that contains more than one particle (and therefore at last eight sub-boxes in 3D) 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 $2^{n\ell}$.

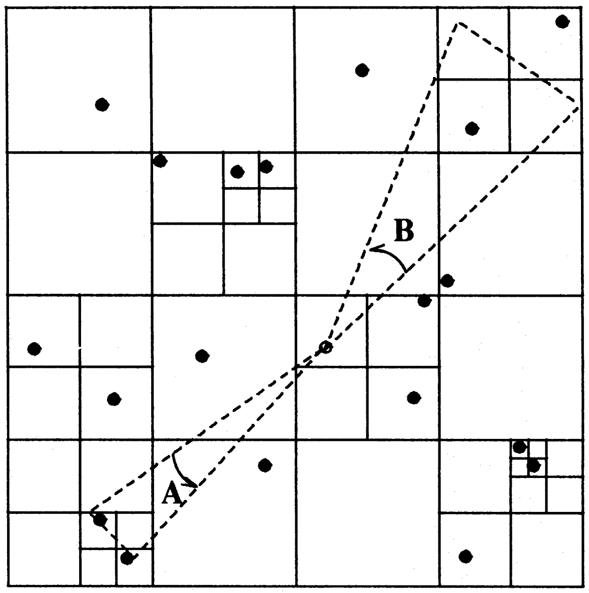

Figure 2. Twenty particles in a 2D tree. Two possible expansions are shown for one particle (open circle): in case (A), the opening angle may be small enough for a multipole expansion, but in case (B) the angle is probably too large so the force contribution of the two particles in the upper right would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

To calculate the force at a point, then, 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 $\theta_C \lesssim 1$ rad), a multipole expansion is performed about the centre of mass of the cell (see Fig. 2). Otherwise, its children are examined: if a leaf is found, its force contribution is calculated directly; but if a child is a branch, then the procedure continues recursively until the angle subtended by the smaller cell is sufficiently small, etc. The number of poles used in the expansion determines the accuracy of the force approximation (see Section 4.4). 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, in preparation, 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. Hernquist 1987)

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

where the gravitational constant G has been defined such that $G\mathrm{M}_\odot \equiv 1$, where $\mathrm{M}_\odot$ is the mass of the Sun. Here M is the total mass of the cell particles (in solar masses), and $\boldsymbol{Q}$ is the quadrupole moment tensor of the cell, given by

$$Q_{jk} = \sum_i m_i(3x_{i,j}x_{i,k} - r_i^2\delta_{jk}), \tag{4}$$

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 $\boldsymbol{Q}$ is symmetric, and, in 3D, traceless, so that only $2n-1$ elements need be stored in memory for each matrix. Hernquist also gives a useful recursion relation for calculating the quadrupole moment of a cell from the quadrupole moments of its children (cf. Shift Theorem);

$$\boldsymbol{Q} = \sum_i^{N_{\rm cells}} \boldsymbol{Q}_i + \sum_i^{N_{\rm cells}} m_i[3\boldsymbol{r}_i'\boldsymbol{r}_i' - (r_i')^2\mathbf{I}], \tag{5}$$

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 parallel architectures (e.g. Hernquist 1990; Makino 1990). 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 (Gammie, private communication) is obtained by constructing a tree for the central box alone, which can then be mapped without modification on to each ghost box since the relative position of any two particles is preserved under equation (2) (recall that it is the sliding motion of the boxes which gives rise to the shear). The total force 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 (Section 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 ghost 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 Section 3, along with a general description of the more basic components of the code.

3 CODE DETAILS

The technical details of box_tree will be presented in this section. First, an overview of the logical structure of the program will be given to provide a framework for the more detailed discussion. Next, a short description of the layout and use of the major data structures will be presented. Following this, the special problems encountered during development of the code – and their solutions – will be described. Fig. 3 shows a schematic outline of the main integration loop for reference.

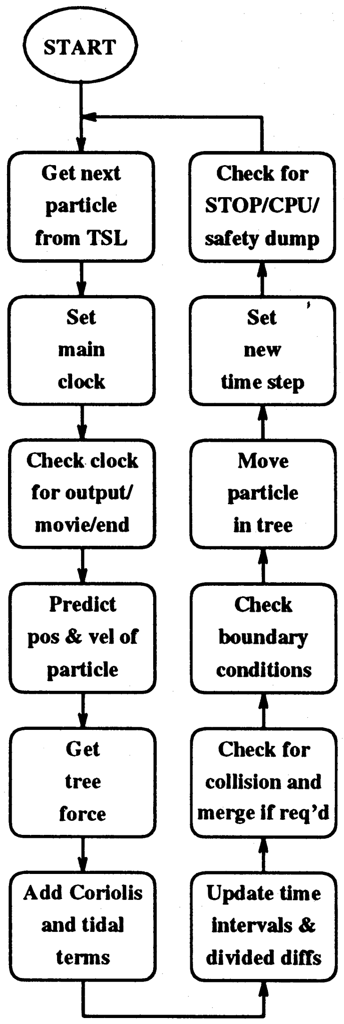
START
Get next particle from TSL
Set main clock
Check clock for output/ movie/end
Predict pos & vel of particle
Get tree force
Add Coriolis and tidal terms
Update time intervals & divided diffs
Check for collision and merge if req'd
Check boundary conditions
Move particle in tree
Set new time step
Check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

Figure 3. A schematic of the program flow in the main box_tree integration routine, described in Section 3.

3.1 Overview

The box_tree integrator is based on a standard fourth-order polynomial N-body code with individual time-steps (Aarseth 1985), much the same as that used in ALP. Particles are given initial positions and velocities at time $t=0$ and the force and first three derivatives acting on the particles are calculated analytically. Each particle is assigned a time-step δt_i proportional to $r/\dot{r}$, where r is the relative position of the particle's nearest neighbour (which may be a ghost). Divided differences are introduced by converting from the Taylor series derivatives. The times $t_i+\delta t_i$ (initially $t_i=0$) 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 ($\mathscr{F}^{(3)}$) 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 ($\mathscr{F}^{(1)}$) and summing the contributions to the force (including ghosts). Then new derivatives/differences are calculated and a fourth-order semi-iteration is performed to improve the accuracy further. 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 and tree codes. 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, performance of multipole expansions over nodes that are sufficiently small or far away, or addition of contributions from individual particles; (3) addition of Coriolis and tidal terms to the force (see equation 1); (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 Section 4.4). 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, pre-processor 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 roughly 350 bytes of information per particle. Memory for these structures is allocated dynamically at run time, and de-allocated 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 (approximately 485 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 Get_num_leaves(branch, num_leaves)
BRANCH_T *branch;
int *num_leaves;
{
    register int i;
    for (i=0; i<MAX_NUM_CHILDREN; i++)
        switch (branch->child_type[i]) {
            LEAF:
                ++(*num_leaves);
                break;
            BRANCH:
                Get_num_leaves(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 that 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, pre-processor 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, and logical flags 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 155 routines (of which 74 are global), grouped into 16 files. There are two 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 just under 300 kB in size (less than 10 000 lines). The size of the executable is just under 250 kB when compiled and optimized using cc -O on a Sparc IPX running SunOS 4.1.1.

3.3 Special 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 below.

3.3.1 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 time-scales, such as the minimum cell crossing time. However, these methods are unsuitable for the current project: rebuilding of the tree every time-step is far too expensive, since a high collision frequency leads to very short time-steps, but it is difficult to assign an effective tree-step 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 Sections 3.3.2 and 3.3.5, however).

Fig. 4 is a schematic representation of the tree repair algorithm designed for the planetesimal problem. 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 repair is performed since the tree structure will not change. Otherwise, there are three possibilities: (1) the

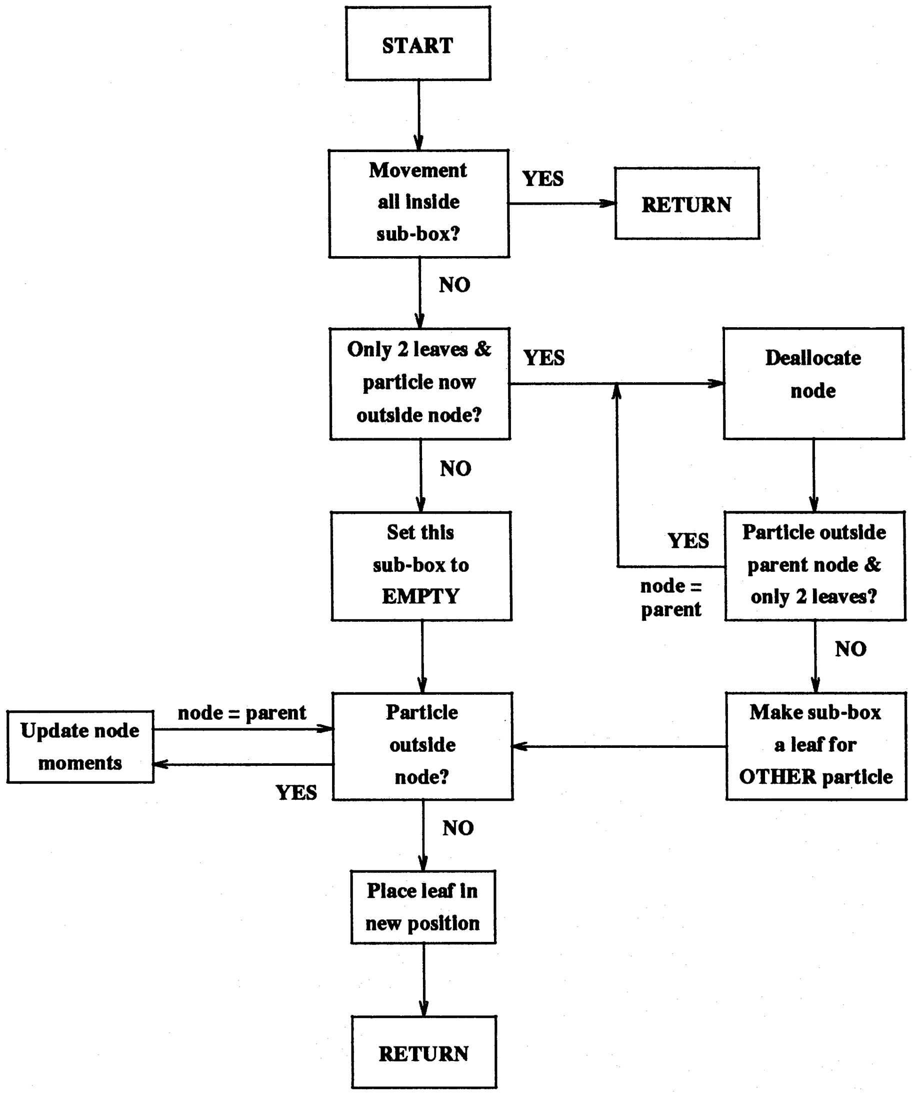
START
Movement all inside sub-box?
YES
RETURN
NO
Only 2 leaves & particle now outside node?
YES
Deallocate node
NO
Set this sub-box to EMPTY
YES
node = parent
Particle outside parent node & only 2 leaves?
NO
Update node moments
node = parent
YES
Particle outside node?
Make sub-box a leaf for OTHER particle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

Figure 4. Schematic representation of the tree repair algorithm, described in Section 3.3.1.

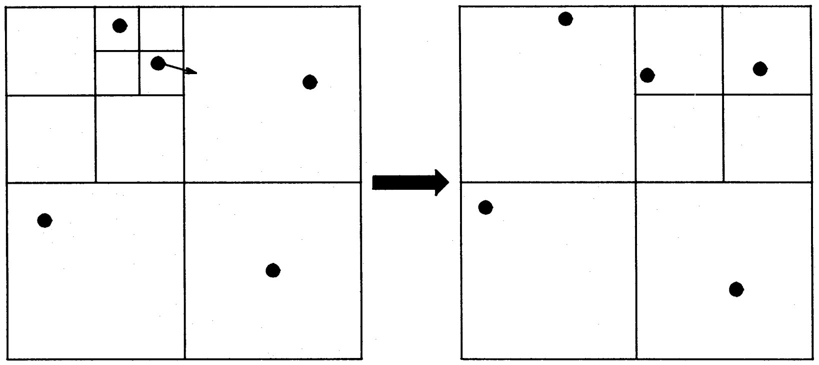

Figure 5. An example of tree repair. Here two nodes are destroyed and one is created.

particle is moving between sub-boxes of its parent; (2) the particle is leaving the current node and there are two or more particles remaining in the node; or (3) the particle is leaving its node and there is only *one* particle remaining. In cases (1) and (2), the sub-box previously occupied by the particle is set empty; no further structural changes need be made since the original node remains intact. In the third case, however, there is only one particle left behind (called the 'other particle') so the node and its parents must be destroyed until either (a) the new position is contained within an ancestral node, or (b) there are at least three particles in an ancestral node (namely the first particle *before* it moves, the other particle, and at least one more), whichever comes first. When such a node is found, the other particle is made into a leaf in the appropriate sub-box of the ancestral node. At this point in all cases, the current node and its ancestors are updated (Section 3.3.2) 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. Fig. 5 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 per cent of the total computation time during a typical run (see Section 4.3).

3.3.2 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 Section 3.3.1. 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. Since updates are performed only during tree repair, however, 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, as done by 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 very easy 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 (5). For completeness, the quadrupole tensor derivatives are given by

$$\begin{aligned}
\dot{Q}_{jk} &= 3(\dot{x}_j x_k + x_j \dot{x}_k) - 2\delta_{jk} \sum_{k'} x_{k'} \dot{x}_{k'}, \\
\ddot{Q}_{jk} &= 3(\ddot{x}_j x_k + \dot{x}_j \dot{x}_k + x_j \ddot{x}_k) - \delta_{jk}(\dot{r}^2 + 2\sum_{k'} x_{k'} \ddot{x}_{k'}), \qquad (6) \\
\dddot{Q}_{jk} &= 3(\dddot{x}_j x_k + \ddot{x}_j \dot{x}_k + \dot{x}_j \ddot{x}_k + x_j \dddot{x}_k) - 2\delta_{jk} \sum_{k'} (x_{k'} \dddot{x}_{k'} + \dot{x}_{k'} \ddot{x}_{k'}),
\end{aligned}$$

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 introduction of the octupole moment (see Section 4.4).

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/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 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 purposes 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 (Section 3.3.3).

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,

$$\Delta t_{\mathrm{M}}=\epsilon_{\mathrm{M}}\left(\frac{s_{\mathrm{cell}}\sum_k|\mathscr{F}_k|+|\dot{\boldsymbol{x}}|^2}{\sum_k|\dot{x}_k|\sum_k|\dot{\mathscr{F}}_k|+|\mathscr{F}|^2}\right), \tag{7}$$

and, for the quadrupole,

$$\Delta t_{\mathrm{Q}}=\epsilon_{\mathrm{Q}}\left(\frac{\sum_i|Q_i|\sum_i|\ddot{Q}_i|+|\dot{\boldsymbol{Q}}|^2}{\sum_i|\dot{Q}_i|\sum_i|\dddot{Q}_i|+|\ddot{\boldsymbol{Q}}|^2}\right), \tag{8}$$

where ϵ_{M} and ϵ_{Q} are constants. A check is made to ensure that 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 choices of ϵ_{M} and ϵ_{Q} are discussed in Section 4.4.

There are further complications that result from node prediction, but discussion of these will be deferred to Section 3.3.5.

3.3.3 Boundary conditions

Both particles and ghost boxes are subject to periodic boundary conditions in box_tree. The y-positions of the sliding boxes are checked each integration step and are offset by one box size if $(3/2)\Omega t-N_{\mathrm{X}}>1/2$, where N_{X} is the number of times such a crossing has already occurred (initially zero). This ensures that the ghost boxes remain within half a box size of the central box at all times, preventing the symmetry of the box system from becoming overly skewed.

The situation for particles in the central box is somewhat more complicated. Once an updated particle position has been assigned during an integration step, a boundary check is performed. If the particle is found to be leaving the central box, the particle is replaced by its ghost image which is simultaneously entering the central box from the opposite side. In the case of crossings in the x-direction, the central box undergoes a net change in its z angular momentum (L_z) because of the uniform shear across the box. This discontinuity in L_z is required to keep the positions and velocities of the particles continuous across the boundaries. However, L_z is otherwise a conserved quantity (see WT), and is therefore a useful check on the stability of the N-body method. The z angular momentum per unit mass, with respect to the inertial frame, of a particle with coordinates (x, y, z) in the rotating frame is given by

$$\ell_z=X\dot{Y}-\dot{X}Y, \tag{9}$$

where to first order $X = a + x$, $Y = y$, $\dot{X} = \dot{x}$, and $\dot{Y} = \Omega(a + x) + \dot{y}$. If $a \gg x$, y and $\dot{x}$ is small, then

$$\ell_z = (\dot{y} + 2\Omega x)a + \Omega a^2. \tag{10}$$

Since $a(=1$ au) and Ω are constants, it is sufficient to call $L_z^i = m_i(\dot{y} + 2\Omega x)$ the z angular momentum of particle i. Hence, to keep $L_z = \sum_i^N L_z^i$ constant, the following correction must be made after each boundary crossing:

$$L_z = L_z^{\text{old}} - \frac{1}{2}\Omega m_i i_x s, \tag{11}$$

where $i_x s$ is the x-offset of the ghost box that the particle is entering (cf. equation 2).

The technical problems caused by particles undergoing boundary crossings in the tree are conveniently handled by the tree repair routines (Section 3.3.1). However, the repair algorithm will actually miss updating the moments of the root node following a boundary crossing, since the particle never leaves the root but is instead displaced within it by a large distance. This is simply due to the fact that the root node and the central box overlap perfectly (i.e. $s \equiv s_{\text{root}}$). The solution is to pass a flag from the main integration routine to the tree repair algorithm, indicating whether or not a particle boundary crossing occurred and the root node should be updated.

3.3.4 *Two-dimensional trees*

Since planetesimals are more or less confined to a plane, it seems inefficient to use a 3D tree to represent the system. Nevertheless, 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 disc. The additional degree of freedom also prolongs the evolution process (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 on to 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 Section 3.3.5 below.

A less crucial but none the less 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, resulting in an infinite number of subdivisions. In practice, this situation has never arisen, but particles 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

$$i_{\text{node}} = (i_{\text{node->parent}})2^n + j + 1, \tag{12}$$

where $i_{\text{Root}} \equiv 0$ 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 used primarily for bookkeeping purposes and has proved useful when debugging. Unfortunately, on a Sparc IPX, i_{node} must be less than 2^{31} for signed integers ($i_{\text{node}} = -1$ is used for initialization), which for $n = 2$ means that 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, 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). 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 Section 3.3.1 or Fig. 4). 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 need be made. 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 $s_{\text{root}}/2^{\ell_{\max}}$), but for expansion purposes it is highly probable that the effective size (see Section 3.3.5 below) will always be greater, so expansions will not be affected at all.

3.3.5 *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 (Section 4.4). 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 $\theta = s_{\rm eff}/r$. Naturally, $s_{\rm eff}$ can be made arbitrarily accurate by reducing the node update time-step coefficients $\epsilon_{\rm M}$ and $\epsilon_{\rm Q}$ (cf. equations 7 and 8), but it has been found (Section 4.4) that the coefficients can retain reasonably large values and still give good results.

Particles can *still* wander outside their nodes, 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. To do this in all cases results in a very noticeable increase in CPU time, but it has been found that this check need only be performed when a particle is within $\delta t_{\rm M} \Omega s$ of the node, where $\delta t_{\rm M}$ is the maximum allowed time-step (typically 0.005 yr) and s is the box (patch) size, as before. Note that this method is reliable only if $\theta_{\rm C} \lesssim n^{-1/2}$, which can easily be shown to guarantee elimination of the self-gravity problem in the static case.

As mentioned in Section 3.3.4, imposition of a 2D tree on a 3D system gives rise to some troublesome projection effects. Fortunately, a simple generalization of the definition of $s_{\rm eff}$ 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 are 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 (Sections 3.3.1 and 3.3.2), 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.

3.3.6 Collisions

The collision routine in box_tree is fairly sophisticated, employing a full 3D dissipative collision model with particle spin (to be described in detail elsewhere). A crucial aspect of box_tree is the reliable detection of collisions, which are the only means of energy dissipation in the planetesimal system. Since particle time-steps are proportional to the ratio of the relative position and velocity of the nearest neighbour, only very unusual situations involving more than two bodies simultaneously would ever be a problem. However, collisions are invariably detected *after* they have actually occurred, since currently only a simple comparison of distance versus sum of radii is performed (the particles are assumed to be spherical). Consequently, two particles may penetrate each other briefly before being allowed to merge or bounce. Typically the penetration distance is $\lesssim 2$ per cent of the collision distance. Such distance discrepancies can lead to errors in angular momentum conservation of the order of 0.1 per cent during the collision, which can then affect the conservation of L_z (Section 3.3.3). The only practical solution to this problem at present is to make the time-steps even smaller when particles come within a certain distance of each other, say 10 particle radii. The following formula has been found to reduce the error in angular momentum conservation following a collision by two orders of magnitude in typical runs:

$$\Delta t = \begin{cases} \epsilon(r/\dot{r}), & r > 10R, \\ \epsilon(r/\dot{r})2^{-[(\mathrm{int})(10R/r)]}, & r < 10R, \end{cases} \tag{13}$$

where R is the average particle radius and ϵ is a constant, usually 0.02. Since a smooth variation is not required, the integer function is used in equation (13) so that the factor involving the power of 2 can be calculated with fast bit-shifting operations.

Since collisions and mergers create discontinuities, the particles involved get special treatment. In particular, after each collision, the force polynomials of the two particles (or the remaining particle after a merger) must be reinitialized. Further, the particle(s) must be completely removed from the tree and replaced so that the derivatives of the moments of the ancestors are changed to reflect the new velocity and force terms of the colliders. A special streamlined 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. However, collisions and mergers generally happen so infrequently that all this extra work is negligible over the course of a run. 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.

4 TEST RESULTS

In order to assess the reliability and efficiency of box_tree, a number of test runs were performed. First, a comparison was made with a simulation carried out by ALP that used the box method alone. Then several timing and accuracy tests were performed by varying input parameters in a controlled fashion. The results of these tests will be presented in this section.

4.1 Comparison with ALP

Model Z2 of ALP was chosen for comparison with a box_tree run. In this simulation, $N = 100$ (central particles), $m_i = 8 \times 10^{-11}$ $M_\odot$ and $\rho_i = 1.4$ g cm^{-3} (so $R_i \simeq 2 \times 10^{-6}$ au). The box size s was 66 Roche radii [where a Roche radius is defined as $(m/3)^{1/3}$ in the current system of units] and the initial velocity dispersions were $\sigma_x = 3 \times 10^{-4}$ Ωa, $\sigma_y = \sigma_z = \sigma_x/2$, appropriate for a 'warm start' simulation. Particle positions in x and y were determined randomly and particle velocities were set to the appropriate dispersion multiplied by a Gaussian deviate. Positions and velocities in z were obtained by randomly choosing a maximum distance above the mid-plane equal to σ_z times a Gaussian deviate and randomly choosing a phase between 0 and 2π for the subsequent harmonic motion (see equation 1). Initial bound pairs were rejected if the semimajor axis was smaller than one Roche radius. Fig. 4 of ALP shows the velocity dispersion evolution of the planetesimals over 10 000 yr: 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 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.

Fig. 6 shows the first 4000 yr of a similar run performed using box_tree. Initial conditions were chosen in the same way, although a different random number generator seed was used. The opening angle θ_C was 0.6, the monopole and quadrupole time-step coefficients were 0.001 and 0.01 respectively (see Section 4.4 below), and the radial coefficient of restitution was set to 0.5. The evolution in velocity dispersion follows the same general trend as in ALP: after 4000 yr, σ_x has risen to 0.004 (over 10 times its starting value), and $\sigma_y \sim \sigma_z \sim \sigma_x/2$. Equilibrium has not been attained, but the dispersions are visibly beginning to level out (the equilibrium value of σ_x is expected to be ~ 0.0045 and should be attained in $\sim 10\,000$ yr according to ALP). Fluctuations in σ_x and σ_y are at the $\sim 100 N^{-1/2}$ per cent 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 ~ 50–100 yr) in σ_z present in both graphs, but the higher sampling rate makes the effect much more visible in Fig. 6. This

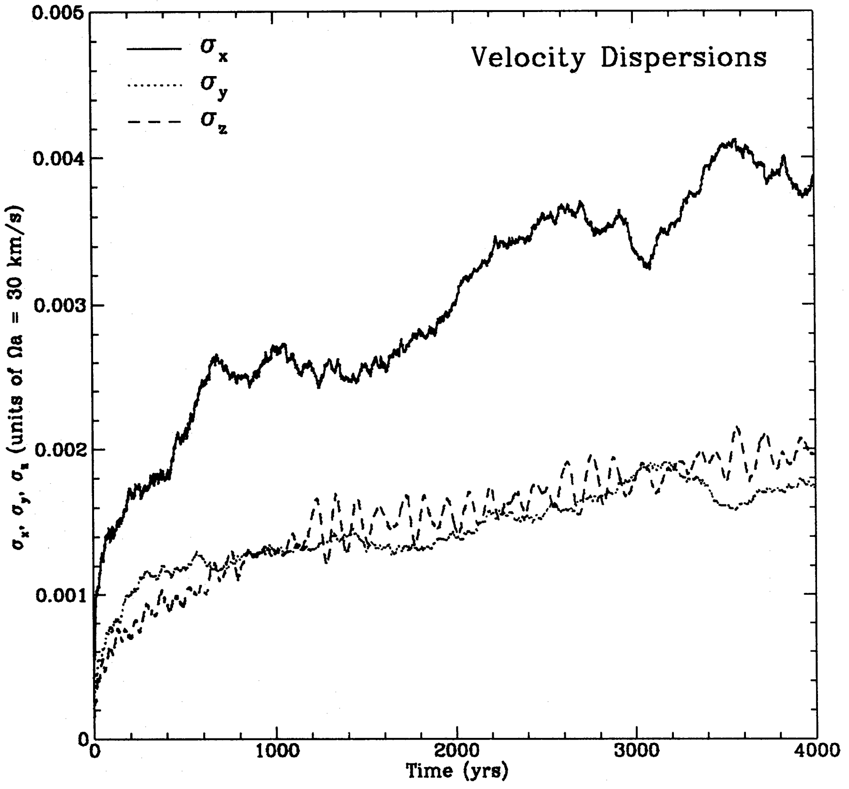

Figure 6. Velocity distribution evolution of a 4000-yr box_tree run with $N = 100$. The system undergoes rapid heating in the first few hundred years, then slowly climbs towards equilibrium (at roughly 10 000 yr, beyond the range of the graph shown here). The periodic oscillation in σ_z is attributed to systematic self-gravity effects. Otherwise fluctuations are mostly random in nature.

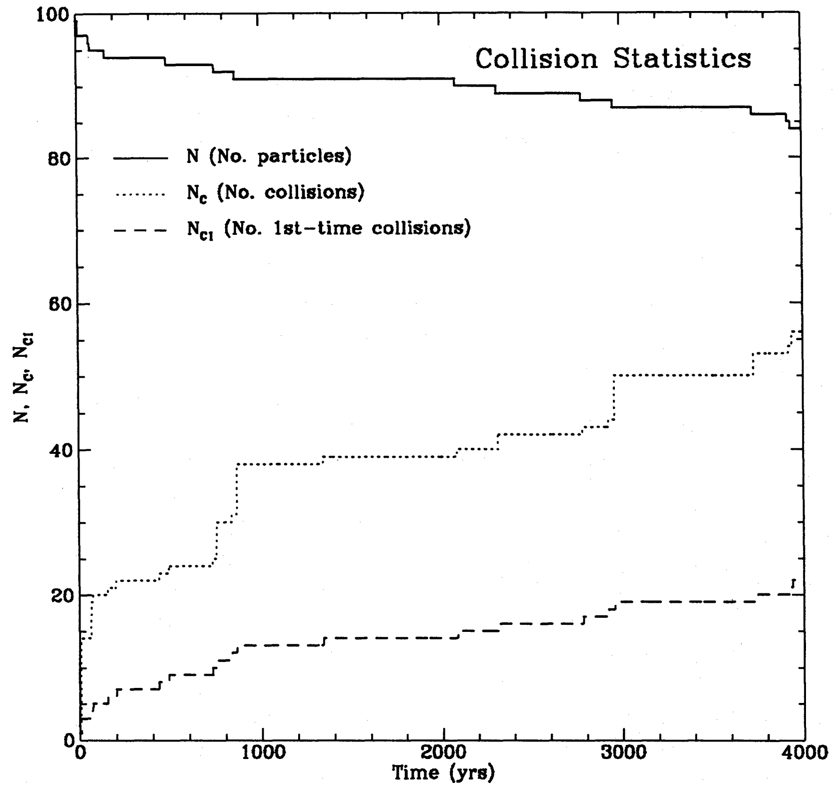

Figure 7. Collision statistics for the system described in Fig. 6. The graph shows that most encounters lead to a merger after one or two collisions.

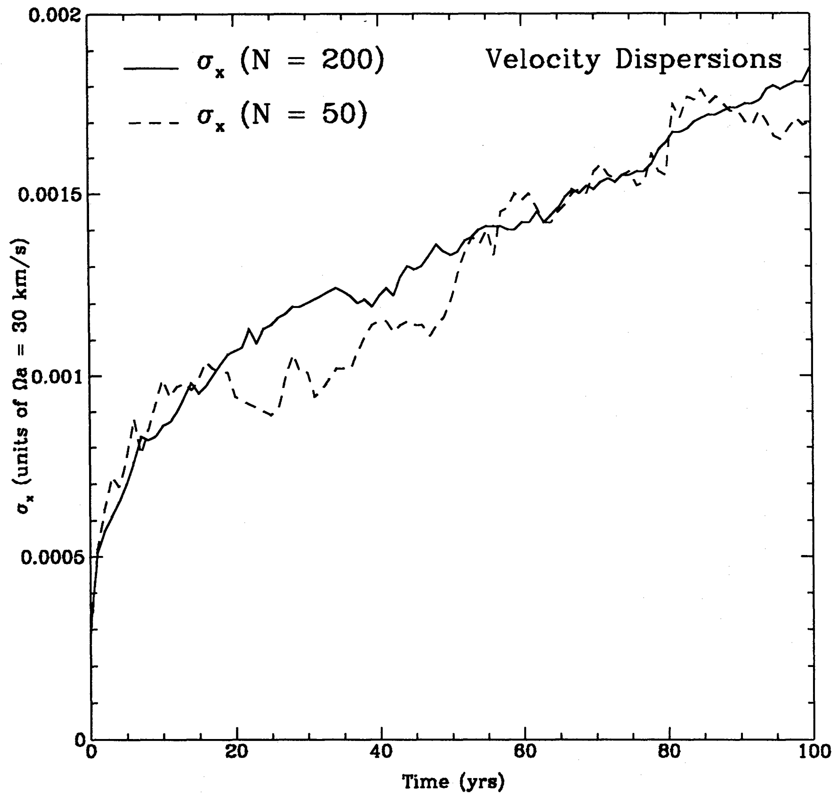

Figure 8. Velocity dispersion evolution in x over 100 yr for two systems of particles in 3D: (1) 200 particles in a box of width 0.04 au (solid line); and (2) 50 particles in a box of width 0.02 au (dashed line). The systems should behave similarly, since they have identical surface densities. This is 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.

interesting oscillation appears to be a consequence of self-gravity in a thin disc and merits further study. Collision statistics for the box_tree run are shown in Fig. 7 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. Fig. 8 shows the growth in x-velocity dispersions over 100 yr for: (1) $N=200$, $s=0.04$ au (solid line); and (2) $N=50$, $s=0.02$ au (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 criterion, at least within the current regime of interest.

4.2 Timing tests

As discussed in the Introduction, the main goal of the tree code is to reduce the cost of the force calculation from $O(N^2)$ to $O(N \log N)$. Fig. 9 is a graph of CPU time versus particle number for θ_C between 0 and 1 (inclusive) in increments of 0.1 rad. Also shown (dashed line) is the time taken for the direct force when used in place of the tree force. Note that this is *not* the same as $\theta_C=0$, since a lengthy search by node (tree walk) is performed even in the zero-angle case. The initial conditions for the tests were the same as for the run discussed in Section 4.1, except that 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. Fig. 10 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 $O(N^2)$ 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 in the direct-force case, showing that the tree method becomes more and more advantageous as the particle number increases, consistent with an $O(N \log N)$ algorithm. For the $\theta_C=0.6$ case with $N=250$, the tree method is approximately seven times more

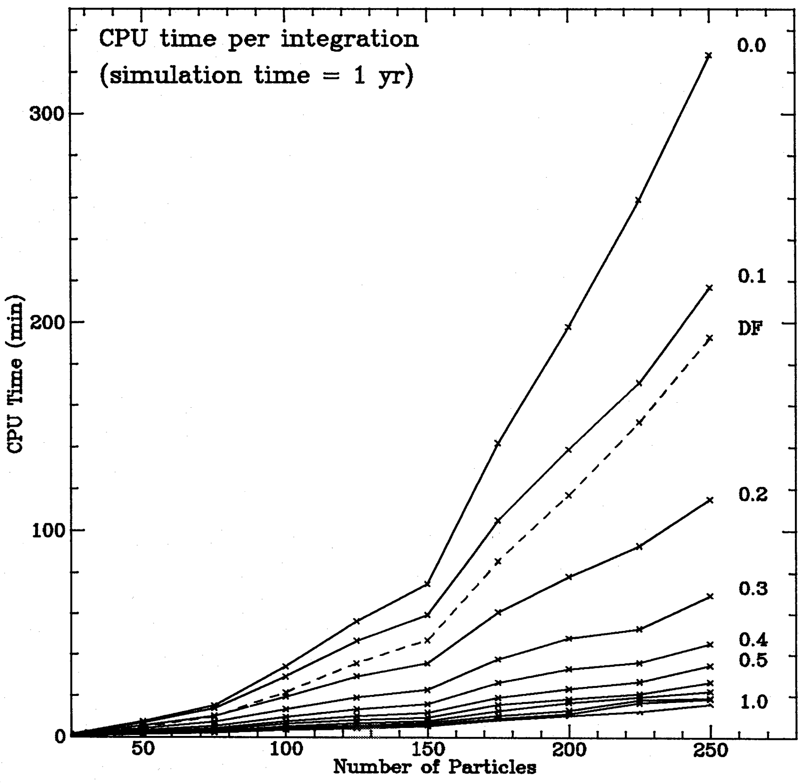

Figure 9. CPU time per run on a Sparc IPX for 10 values of θ_C ($0 \leq \theta_C \leq 1$) and for 10 values of N ($25 \leq N \leq 250$). Also shown is the direct-force case (dashed line). Initial conditions were identical for each N. Runs were performed on a Sparc IPX using box_tree code compiled with the −O optimization flag. 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.

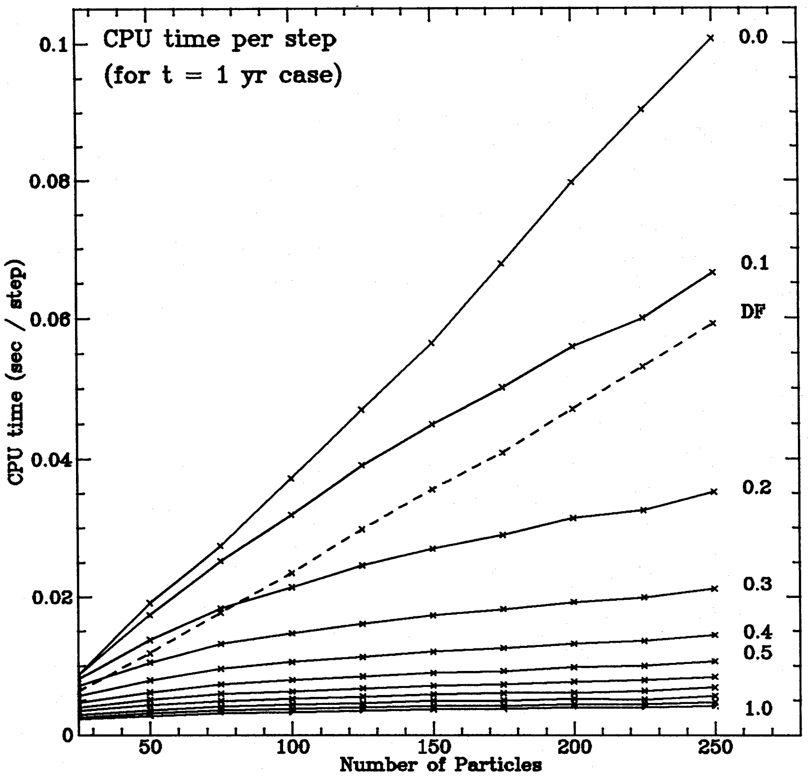

Figure 10. CPU time per integration step for the runs shown in Fig. 9. Note that the direct-force case gives a straight line, as expected for an $O(N^2)$ algorithm. For reasonable values of θ_C, the lines become virtually flat, indicating that the CPU time per step is almost independent of N. This is consistent with an $O(N \log N)$ algorithm. Recall that all overheads are included in the CPU time, so the CPU per step values shown here are slightly larger than the true values.

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, box_tree is still approximately 50 per cent faster (in CPU per step) than the ALP method for $N \sim 100$ and $\theta_C = 0.6$, and 2–3 times faster for $N = 250$.

A timing test with fewer particle cases (N between 20 and 100 in increments of 20) 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 time-scale for the system is much larger than 10 yr (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.

4.3 Performance profile

A performance profile of box_tree was obtained using the gprof utility available with SunOS 4.1.1. The profiler gives a detailed analysis of program flow, and, most important, a breakdown of CPU usage by function. The 14 most expensive functions (those that required more than 1 per cent of the total CPU time) are shown in Table 1 for a run lasting 25 yr (over 1.2×10^6 time-steps in ~ 340 CPU min on a Sparc IPC). 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. Table 1 shows that the most time was spent calculating the multipole expansion during the profile run, but note that there were more than 2.5 times as many expansions as direct-force calculations. The next most expensive box_tree routine was calc_r2_data(), which simply calculates the relative position and square distance between two positions. The reason the routine took up so much time is simply that it was called so often (more than 2×10^6 times) and had to perform n multiplications each call. The system square root function was the fourth most expensive routine, suggesting that a fast square root algorithm would benefit the code. Many of the top 14 functions involve predictions, and several are related to the boundary conditions. Notice that only just over 1 per cent of the CPU time was spent inside the main integration routine itself between calls to the other functions. Note also that functions involving tree repair and extended node checks do not appear on the list, as they took less than 1 per cent of the total CPU time.

One way to speed up the code further would be to replace portions of functions or entire functions themselves with pre-processor macros, i.e. in-line code. The reason for this is that box_tree was written to be as general as possible, so, for instance,

Table 1. Profile of top 14 CPU-intensive box_tree functions.

Function	% CPU	Time (sec)	No. Calls
`add_node_multipole_force()`	19.7	4049.27	79432098
`mcount()`	16.3	3350.86	not avail
`calc_r2_data()`	12.2	2504.80	226631174
`sqrt()`	10.9	2236.87	not avail
`Predict_q_mom()`	6.1	1258.01	78546453
`Predict_com_pos()`	6.0	1229.03	118311049
`add_tree_force()`	5.8	1190.49	11354139
`Copy_vec()`	4.1	833.91	226631184
`Wrap()`	2.6	525.82	145153615
`add_direct_force()`	2.3	469.45	28888027
`Ghost_pos()`	2.2	452.66	145153620
`add_to_quadrupole()`	1.7	359.19	3046490
`Predict_pos_lo()`	1.5	303.45	28888027
`Integrate()`	1.3	264.30	1

both 2D and 3D physical *and* tree models are supported. This means that loops are used to sum over coordinate indices, each requiring the use of a register variable, several summations (both integer and real in most cases), and comparisons. For example, calc_r2_data() consists of only a few lines:

```
static void calc_r2_data(pos1, pos2)
double *pos1, *pos2;
{
        register int k;

        for (r2 = 0.0, k = 0; k < NUM_PHYS_DIM; k++)
             r2 += SQ(rel_pos[k] = pos1[k] - pos2[k]);
}
```

(here r2 and rel_pos[] are global to the force routines, SQ() is a simple macro for computing the square of its argument, and NUM_PHYS_DIM is the number of 'physical' dimensions in the simulation, usually 3). This entire function can be replaced with the following macro:

```
# if (NUM_PHYS_DIM == 3)
#         define calc_r2_data(pos1, pos2)\
                  r2 = SQ(rel_pos[0] = pos1[0] - pos2[0]) + \
                       SQ(rel_pos[1] = pos1[1] - pos2[1]) + \
                       SQ(rel_pos[2] = pos1[2] - pos2[2]);
# else
#        define calc_r2_data(pos1, pos2)\
                  r2 = SQ(rel_pos[0] = pos1[0] - pos2[0]) + \
                       SQ(rel_pos[1] = pos1[1] - pos2[1]);
# endif
```

where it is assumed that NUM_PHYS_DIM is allowed to be 2 or 3 only. These lines of code (depending on NUM_PHYS_DIM) will actually *replace* the line calc_r2_data(arg1, arg2) wherever it appears in the force routines. This eliminates a function call, a loop variable, extra summations and a comparison. A test run with this change was performed under conditions identical to the first profile run, and it was found that a saving of ~6 per cent in CPU was achieved, amounting to 20 min overall. Portions of add_node_multipole_force() could be replaced in this way, as could portions of the prediction and boundary condition routines. Thus a considerable saving could be achieved by extensive use of macros, at the price of code readability. Currently it has been decided *not* to incorporate these macros, to allow for simpler program development. However, large production versions would benefit from these changes.

4.4 Accuracy

There are a number of factors that contribute to differences between the force calculated using the tree method and that calculated using the direct method. First and foremost is the intrinsic error in the multipole approximation, which is a function of the expansion order and the choice of θ_C. 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

$$\varepsilon = \frac{|\mathscr{F}_M - \mathscr{F}_D|}{\mathscr{F}_D}. \tag{14}$$

In the case of a $\theta_C = 0.6$ expansion to quadrupole order for two equal-mass particles with perfectly determined positions, the largest possible error is ~ 24 per cent (~ 58 per cent 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 per cent. Fig. 11 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 ($\theta_C = 1.0$, 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 $\theta_C = 0.2$ and 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 artefact 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 ~ 0.5 per cent (maximum error of order 10–20 per cent) for reasonable values of θ_C ($0.375 \lesssim \theta_C \lesssim 0.625$) and for systems that have a fairly uniform particle distribution. For a more detailed assessment of expansion errors in tree codes, see Barnes & Hut (1989).

It should be noted that, because boundary conditions are applied to predicted node positions (Section 3.3.2), 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

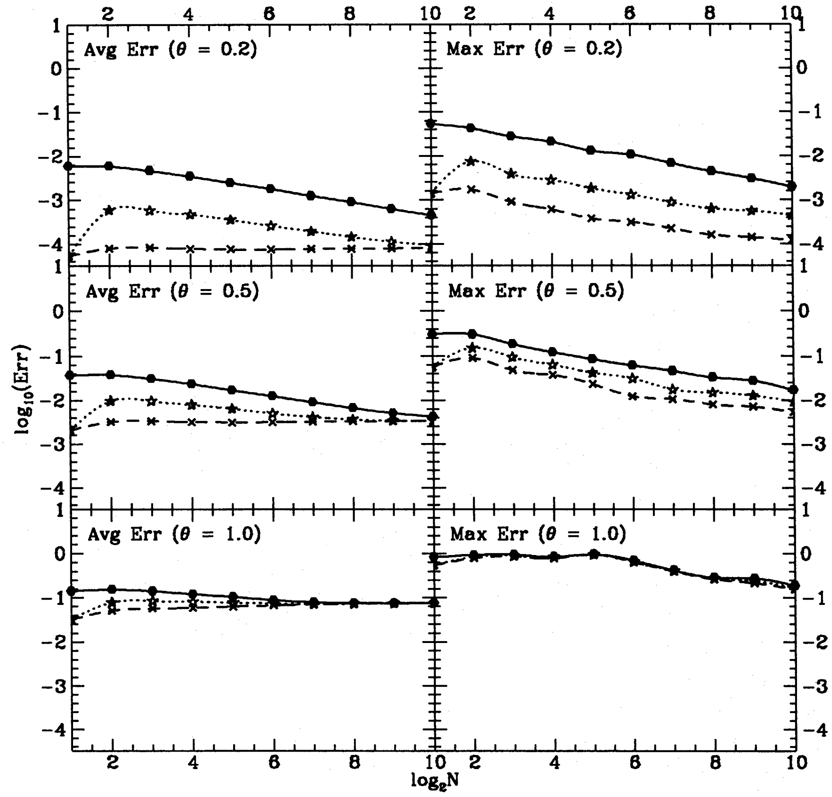

Figure 11. Average and maximum force errors (cf. equation 14) that result from termination of multipole expansions at the monopole (solid line), quadrupole (dotted line), and octupole (dashed line) terms. Graphs are shown for three values of θ_C (0.2, 0.5 and 1.0) taken along the diagonal of a box packed 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. These differences are also more marked for the smaller values of θ_C.

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 (Section 4.2), 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 (ϵ_M and ϵ_Q in equations 7 and 8) that minimize both force errors and CPU expense. The values of ϵ_M and ϵ_Q were independently varied in powers of 10 between 10^0 and 10^{-7} for two cases, one with $N=50$ for 5 yr, and the other with $N=100$ for 1 yr. From these tests and earlier runs, ϵ_M and ϵ_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 initial conditions 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. ε averaged over every expansion for each integration step of the entire run) was 0.176 per cent and the maximum error was 18.0 per cent. In only 268 instances did force errors occur above the 10 per cent 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. Section 3.3.3) – 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 eight resulted in mergers. Both the average and maximum absolute errors agree well with Fig. 11, 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].

5 CONCLUSIONS

A new tree code for fast simulation of planetesimal dynamics in a thin disc 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 expanding recursive 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; and (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. With all these elements in place, a considerable gain in efficiency has been achieved. Though still under development, box_tree is roughly 2–3 times faster than the recent N-body box code of APL for moderate values of N (a few hundred), while maintaining force errors close to the theoretical limit for quadrupole expansions. It is believed that, with the incorporation of a fast square root algorithm and optimized in-line code, an even greater performance gain could be attained. The code in any case becomes increasingly advantageous for larger values of N, in a manner consistent with an $O(N \log N)$ algorithm.

During the development of box_tree, an effort has been made to keep the code as general as possible. This means that it would be relatively straightforward to implement such refinements as gas drag and fragmentation, both important dynamical effects in planetesimal evolution. With a bit more work, the coordinate system could be referred back to a fixed inertial system to allow study of more conventional problems, such as 3D spherical systems (of fixed size) or even a complete planetesimal ring system during a late stage of planetary formation. Although the final production version of box_tree has yet to be written, the author will be pleased to make the code available to researchers when it is ready. The code includes a package for generating and viewing movies of the planetesimal and tree dynamics, which has not only proved invaluable as a diagnostic tool but has turned out to be a very instructive way of examining the non-intuitive particle motions that occur in a rotating frame.

Currently box_tree is being used to study the evolution of planetesimal spin in the early Solar system to determine the dynamical effect of spin transfer of angular momentum. The most valuable aspect of box_tree, however, is the ability to study large-N systems in a reasonable amount of time using a fairly direct (non-statistical) method. This will allow accurate simulations of planetary formation to be performed for earlier stages than have been examined previously. In particular, 2D systems with N as large as 10 000 – representing over 5×10^9 planetesimals of 10-km size – are being investigated, with the aim of obtaining a realistic mass spectrum for simulations of later stages.

It is hoped that box_tree will prove to be a useful tool for numerical simulations in astronomical fields that involve flattened disc systems but that lie outside the regime of planetesimal dynamics. Such areas might include the study of close-packed planetary rings or even molecular cloud collisions in active galactic nuclei.

ACKNOWLEDGMENTS

The author thanks S. J. Aarseth and D. N. C. Lin for their assistance and encouragement while box_tree was under development, as well as for helpful suggestions regarding this paper. Technical advice from S. L. W. McMillan and L. Hernquist was also much appreciated. The author is indebted to the Institute of Astronomy, Cambridge for providing the computing facilities for this project. This work was supported by a Commonwealth Scholarship Award administered through the British Council.

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., Lin D. N. C., Palmer P. L., 1993, ApJ, in press (ALP)
Barnes J., Hut P., 1986, Nat, 324, 446 (BH)
Barnes J., Hut P., 1989, ApJS, 70, 389
Binney J., Tremaine S., 1987, Galactic Dynamics. Princeton Univ. Press, Princeton, NJ
Greenberg R., Wacker J. F., Hartmann W. K., Chapman C. R., 1978, Icarus, 35, 1
Hernquist L., 1987, ApJS, 64, 715
Hernquist L., 1990, J. Comp. Phys., 87, 137
Makino J., 1990, J. Comp. Phys., 87, 148
Nakagawa Y., Hayashi C., Nagazawa K., 1983, Icarus, 54, 361
Palmer P. L., Lin D. N. C., Aarseth S. J., 1993, ApJ, in press (PLA)
Press W. H., Spergel D. N., 1988, ApJ, 325, 715
Safronov V. S., 1969, Evolution of the Protoplanetary Cloud and the Formation of the Earth and Planets. Nauka Press, Moscow
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)