An investigation, implementation and comparison of 3 important particle simulation techniques:
PP : Particle-Particle
PM : Particle-Mesh and
TC : Tree-code

Kees Lemmens
Faculty of Applied Physics,
University of Technology,
Delft.

August 22, 1997

Version 1.1

Members of committee :

Prof.dr.ir. L. Dekker.
Prof.dr.ir. S.W. de Leeuw.
Dr.ir. J.J. Derksen.
Dr.ir. E.E.E. Frietman.
Dr.ir. H.X. Lin.

This report was completely realised without using *any* commercial software: all programs were written using the Public Domain JED editor and compiled with one of the best C-compilers available: the GNU-C Compiler.
All drawings were made using the Xfig vector drawing package, the 2D and 3D graphs were generated through GNUplot and the typesetting was done using LaTeX 2E with the LyX "wysiwyg" frontend. As operating system the excellent Public-Domain Unix system called Linux was used, installed on a Pentium 75 with 32 MB internal memory.

Contents

Abstract

In this report 3 basic particle simulation techniques are investigated, implemented and compared with each other: the Particle-Particle method, the Particle-Mesh method and a variant of the Barnes-Hut ([4]) Tree-Code algorithm. All of them were implemented in such a way that it was possible to feed the same datafile with information about particle positions, velocities and masses as well as the number of iterations and the size of the timestep to all 3 different programs. This gives the possibility to compare the 3 methods directly with respect to i.e. speed, speedup, accuracy and/or memory requirements.

Many problems that were encountered during the implementation of the 3 programs and of course possible solutions are discussed as well. One of the intentions of this report is to be able to make someone who is not familiar with particle simulations techniques known with the basics in a relatively short time, without going too deep into the hairy and confusing details of the individual methods.

The Particle-Particle and Particle-Mesh techniques are basically the same as described in i.e. the book "Computer Simulations using particles" by Hockney and Eastwood ([9]) except that in this book they are almost exclusively discussed for 1D, while I used 2D instead. The step from 2D to the 3D world - which is the most interesting from a practical point of view - is relatively easy, while 1D is too far of from reality to my opinion.

The Tree-Code algorithm discussed in this report is a variant on the Barnes-Hut algorithm, with the most important differences the building of the tree (where I used a method I decided to call "sort-and-count") and the definition of the Θ criterion (where I use the diagonal length instead of the length of one rib).

Finally a heuristic is presented, that is supposed to be able to analyse a problem within a reasonable time and then give an indication about which of the 3 discussed methods would be best suited for that problem.

To give this indication the heuristic mainly uses the density auto-correlation and the ratio between the average particle displacement during the simulation (AD) and the average minimum interparticle distance (AMI).

List of Figures

List of Tables

Preface

This report describes the results obtained during a research period from Sept 1996 - August 1997.

The research period was part of the Master Degree program from the Faculty of Applied Physics of the Technical University of Delft, to meet the requirements for the final examination.

I would like to thank the following persons for their support during this research period:

- Dr.ir. A.H.P. van der Burgh for his valuable advice and support during my studies and during the research period.
- Dr.ir. H.X. Lin for his thoroughly proofreading the report, for all the useful discussions I had with him about various topics when I didn't know how to carry on with a certain problem and of course for getting fresh coffee or tea when it seemed to be really necessary.
- Prof.dr.ir. L. Dekker for his warm attention, his practical tips in avoiding the administrative pitfalls of the Applied Physics study and for always suggesting the right persons in the organisation when needed.
- Prof.dr.ir. S.W. de Leeuw, Dr.ir. J.J. Derksen and Dr.ir. E.E.E. Frietman for their proofreading and final comments on this report.
- All other members from the Applied Analysis department of the Faculty of Applied Mathematics for always being available to answer nasty questions about mathematical subjects (not necessarily being a part of this report) or to discover yet another erroneous and overlooked minus sign.
- Ryan and Mark for creating the boundary conditions at home that were essential to be able to finish this report : being (sometimes) quiet, washing up the dishes alone and delivering drinks and hugs at crucial moments.

Chapter 1

Introduction

Many different physical phenomena can be described as interactions between pairs of particles. Examples are the motion of planets, satellites and comets in a solar system, colliding large star clusters, but also detergent (and dirt !) molecules in a washing machine, electrons in conductors and semi-conductors, and even charged dust particles in the air.

In all these cases there exists a pairwise interaction (force) between the particles in these systems. These forces can be either a gravitational, electrical or another fundamental force. Also combinations with both attractive and repulsive components, such as van der Waals intermolecular forces are of course possible.

These pairwise interactions depend in many cases on a constant property of both interacting particles such as their mass or charge and on the distance between them, composing a central or conservative force. In the case of gravitational or electrical forces the interparticle force drops with the square of the distance (for a simulation in 3D space !), in other cases this can be more and also less progressive.

Also the relative angle can be of importance, as is the case in so-called "multipole systems" such as with electrical dipoles. The latter is an example of a non-central force, but note that it is always possible to decompose a non-central force into a combination of central forces. (the influence of an electrical dipole on a charge can also be viewed as the influence of 2 separate charges on the third one !)

If we want to simulate such a system in time it is enough to calculate for every timestep the total force acting on each particle (note that this total force is composed of all pairwise interactions with the other particles !) and then calculate for each particle the change of speed and the displacement within this interval. This is basically what we we do in almost all types of particle simulations.

About two years ago I became interested in this technique and I made a simple straightforward sequential and parallel implementation of a particle simulation program on our Parsytec 16 processor transputer system. This program only worked with gravitational or coulomb forces.

However, this simple implementation showed me that there where many practical problems involved in particle simulations :

- If we have a system of N particles we would have order $N * (N - 1)$ calculations for *each* timestep, which makes it practically impossible to use a large number of particles, as the simulation would then take too much time.
- If particles come very close ($r \ll 1$) to each other the forces ($\sim 1/r^2$) become so strong that the discretization error during this timestep in for example the acceleration would become unacceptable.
- What to do with collisions or almost collisions ? In many problems it is unavoidable that particles come very close, so there should be a mechanism to deal with collisions, but thusfar I found remarkably little comments on this in the literature.
- Often the forces of far-away particles can either be neglected in relation with the nearby particles or be taken as one total force, coming from the centre-of-mass point of a far-away set of particles. In these cases we would obviously do too much work if we would use the method as described above.
- What integration method should be used ? I used Runge-Kutta 4 in this first version, but it is very important to realize that - although the discretization error here would be less than i.e. for Euler - the large roundoff error and it's propagation after many iterations for this method could easily become unacceptable.
- Is there a way to determine how correct a simulation would describe the real world ? I realised that two simple things should always apply: the total energy (sum of kinetic and potential energy) has to be conserved and the total momentum should be conserved. So these criteria can be used to discriminate between totally useless simulations and (possibly) useful ones.
- As it was much easier to visualise, I used a 2 dimensional coordinate system for these first simulations. Are there any fundamental differences between 2D and 3D systems we should be aware of or wouldn't it make a difference if - for reasons of efficiency and visualization - we use a 2D system instead of a 3D system to analyse the subject and test the software ?

I tried to answer the questions above and found that apart from the technique I used in my first implementation, which is widely known as **Particle-Particle** as I found out later, all other commonly used techniques could be either viewed as variations on the so-called **Particle-Mesh** technique, as a variation of the **Tree-Code** principle introduced by J. Barnes and P. Hut or as a combination of one of these three basic methods with some minor improvements.

In many publications very clever extentions of these three methods are presented, but I didn't find any of them that describes and compares them all in relation to each other. Also, many implementation aspects (both software engineering and physical aspects) are not or only very roughly discussed.

As I was educated to be a teacher and used the Particle-Particle method as an exercise for our students for the "Parallel Computation I" practical work for about 2 years, I found that it was difficult to find literature that explains the more basic aspects of Particle Computer Simulations, which makes the "take-off" for a newcomer relatively difficult. For this reason I decided to start this investigation with the following targets in mind :

- Write a general overview of all practical aspects involved in particle simulation on computers, that is readable also for newbies.
- Build readable, decent software that uses modern software techniques to deal with i.e. dynamical memory alllocation for all 3 basic methods, both as a sequential and a parallel (message-passing) program, as a lot of existing software (which is mostly in Fortran) is practically unreadable and - from a software-development point of view - often poorly designed. (Sorry folks !)
- Mention practical problems I ran into during development, (also the more basic ones !) so that a newcomer doesn't have to reinvent the wheel anymore, as there is unfortunately little information on this particular subject.
- Compare the 3 techniques as mentioned before, show their strong and weak points and illustrate them by using the same input for all 3 implementations. Also, give some simple numerical criteria that could be used to decide which algorithm is most suited for a certain type of problem. This was - as far as I know - never done before: there are many articles about one of the three techniques and their derivatives, but I didn't find any that puts them together.
- Discuss their parallelisation aspects, such as speedup and scalability on a message-passing system.

Note that in this report I will only consider gravitational/electrical forces, as other forces can often easily be implemented by only changing a few lines in the code (except for Particle-Mesh). Also only 2D simulation will be investigated (this because 3D is less easily graphically visualised) as the techniques are almost the same for both 2D and 3D, apart from a very few aspects (see section 2.3.1).

1.1 Overview of discussed multibody simulation techniques

Hereafter I'll explain the three basic techniques very briefly. A more profound discussion can be found in later chapters.

1.1.1 Particle-Particle method

The Particle-Particle method is the simplest one of all particle simulation methods. For each timestep it calculates the acceleration for every particle by evaluating the force contribution

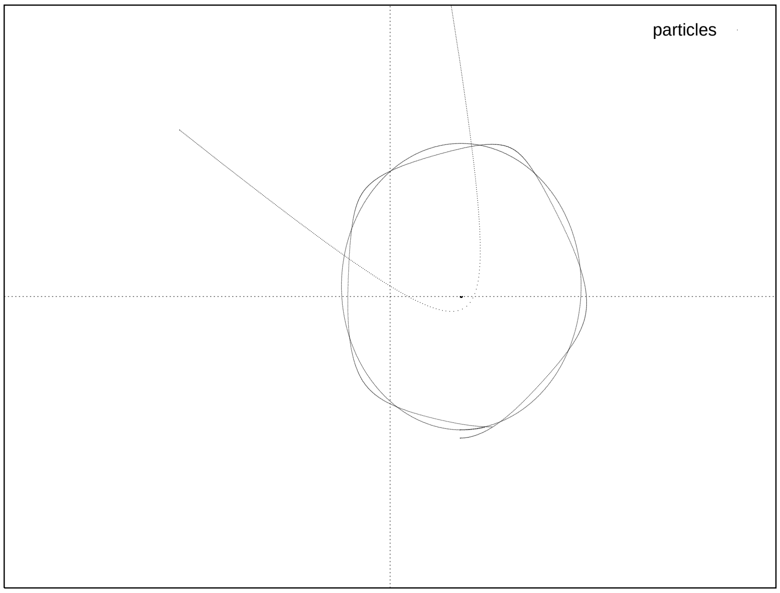


Figure 1.1: Example of Particle-Particle simulation.

at time t of *all* other particles on it, no matter how small this contribution may be. After that it calculates the new speed and displacement at time $t + \Delta t$ using one of the possible integration methods.

As we have to calculate the force contribution of all other particles for each particle in the set, the time necessary to calculate the acceleration for *one* particle increases linearly with the total number of particles. So, the time we need to perform one iteration step (where we calculate the acceleration for *all* particles) increases quadratically with the number of particles. For this reason the Particle-Particle method is very inefficient for large numbers of particles.

This triggered people to start the search for more efficient techniques to simulate large numbers of particles and they even found a couple of them, although these new techniques often tend to be less accurate than Particle-Particle.

A typical example of a 4 body Particle-Particle simulation can be found in figure 1.1. The Particle-Particle method will be discussed more extensively in chapter 3.

1.1.2 Particle-Mesh method

The Particle-Mesh method is one of the oldest improvements over Particle-Particle, introduced somewhere around 1985 by R.W. Hockney & J.W. Eastwood (see [9]). It not only uses a discretization of time, but also a discretization of space. A mesh (or grid if you like) with a constant meshsize is "laid" over the particle space. This could be either a 1D, 2D or 3D mesh,

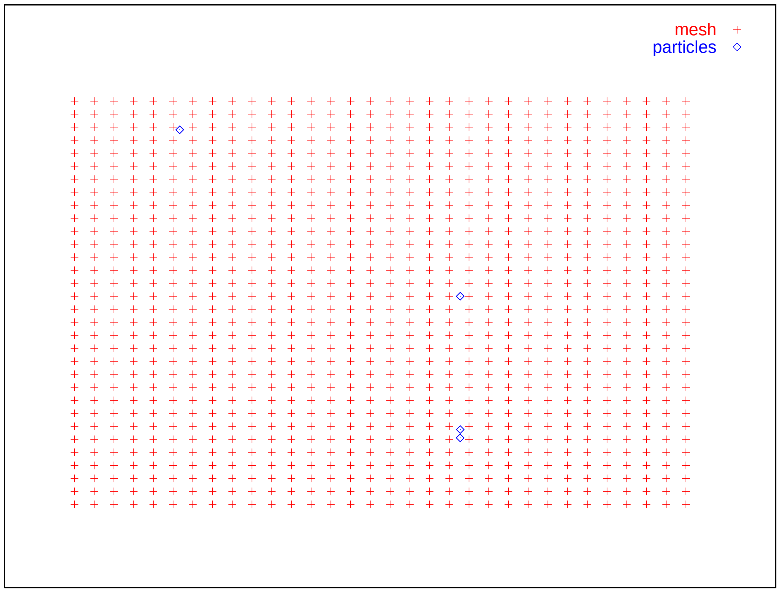


Figure 1.2: Example of a mesh for the problem from the Particle-Particle section.

depending on the type of simulation. Then the particle masses are assigned to one or more of the nearest gridpoints on this mesh. This assignment - which uses an interpolation technique - can be done in many different ways: ranging from very inaccurate to quite accurate.

So, basically mass is not considered to be on other points than on the meshpoints: the mass distribution has been discretized through this mesh. But if we have distributed the masses over the mesh, we do have in fact a discretization of the *density* function $\rho(x, y)$! [1]

The simplest way to calculate the acceleration is by taking all force contributions of all meshpoints masses to our particle (considering the meshpoints to be particles and then using straightforward Particle-Particle). This only leads to a speed improvement if the total number of meshpoints M is (much) less than the total number of particles N. But the total number of calculations then still increases proportionally with the square of the number of particles (or more accurate with $N \cdot M$), so that doesn't bring us much further as the number of meshpoints can't be too small for reasons of accuracy.

There is however a much better (read faster !) way to calculate the accelaration :

According to the famous Poisson's equation it is possible to calculate the potential $\Phi(x, y)$ on every gridpoint only from the density function $\rho(x, y)$. So the next step is to solve this Poisson equation and thus find the potential on every meshpoint. I did that by using FFT (Fast Fourier Transform) techniques, but it is also possible to do this in a much simpler, but unfortunately also more time consuming way.

[1] Note that in this case a 2D system is assumed, resulting in a density function that uses mass/area instead of mass/volume as is the case in a 3D simulation. This results in a force that drops with the inverse of the distance r instead of the inverse of the *square* distance r^2.

If we know the potential $\Phi(x, y)$ (which is a scalar !) at every meshpoint, it is also quite easy to find the gravity (or electric) field $\vec{E}(x, y)$ in every meshpoint by taking the gradient of the potential (which can be different in different directions and thus is a vector !) .

Well, for a gravity field this field is simply the acceleration that works on a particle if it is exactly located on the meshpoint or, in formula : $a = g$.
For an electric field we should multiply with the charge q of the particle (to find the force) and divide by the particle mass m (to find the acceleration), but that is just a matter of scaling as long as the $\frac{q}{m}$ factor is a constant. In formula this looks like : $a = \frac{q}{m} \cdot E$.
For particles not exactly on the meshpoints, the acceleration can be found by using some sort of interpolation technique between the meshpoints. This interpolation technique is for several reasons almost always the same one as used for the mass assignment to the mesh.

The main disadvantage of this method is that the interaction between close particles (where close implies at a shorter distance than the meshspacing) is only a very rough approximation and so it is important that the total force on a particle is mainly determined by the far away particles if we want this method to be accurate. For the same reason collision management is of course not possible here.

An example for a mesh for the problem shown in the previous section is given in figure 1.2. Note that apart from accuracy aspects, the simulation is also very inefficient in this case ! More information about the Particle-Mesh method can be found in chapter 4.

1.1.3 Tree-Code method

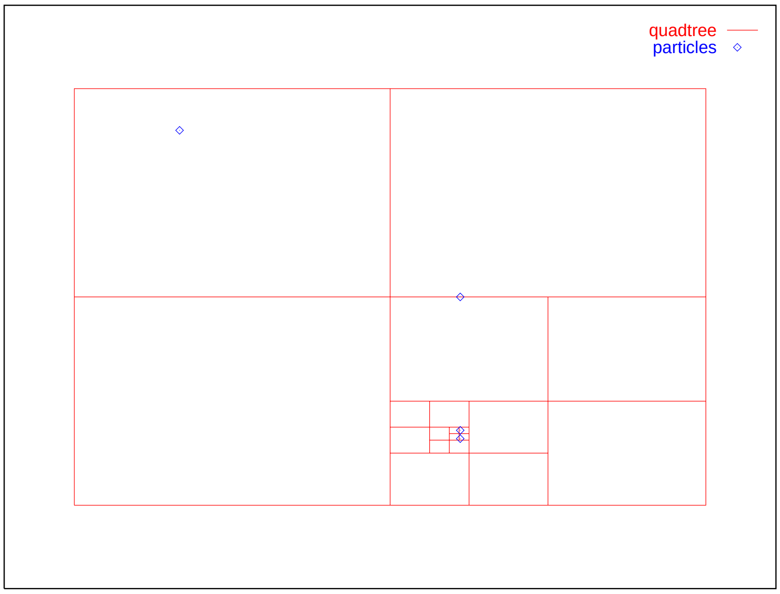


Figure 1.3: Quadtree for the problem from the Particle-Particle section.

Another improvement over Particle-Particle, but one that uses a complete different approach, is the so-called Tree-Code method, introduced in 1985 by J. Barnes and P. Hut (see [4]).

The basic idea is again to reduce the number of calculations, without loosing too much of the accuracy. However, this time we do not use an uniform grid, but a sort of an adaptive grid, that is very dense at places where many particles can be found and very sparse at places where there are little or no particles at all.

The idea is that the force contribution of a large group of far way particles can be accurately approached by substituting this large group with a single virtual particle with the total mass of all particles and with its position in the centre of mass of the far away group.

First we draw a rectangle (the *root cell*) around the particle space, large enough to keep the particles inside during the whole simulation. Then we start splitting up this root tree into two parts for all directions, until we have all particles located in one separate cell.

So, in a 1D simulation there are 2 children cells and each one has half the size of the parent cell (a *binarytree*), in a 2D simulation there are 4 children each a quarter of the parent area (a so called *quadtree*) and in a 3D model there are 8 children each obviously $\frac{1}{8}$ of the parent cell volume (an *octtree*).

It is important to notice that only the smallest cells (often called *leaves*) contain one particle (or even zero particles !) : their parents may of course contain many more particles and the root cell even contains them all.

If we have thus build our tree, we can start computing the total mass and the centre of mass for each cell. This is easy for the leaves: it is just the mass and position of the particle it contains. For the larger ones the total mass is constructed by the total mass of its 2, 4 or 8 children (and not further than that level !) and the centre of the mass is calculated using the positions and masses of its 2,4 or 8 children.

Ok, now the tree is completely setup (for this iteration ...) and it is time to start calculating the particle accelerations. This is done in one of the two following ways :

- By calculating the force contribution of individual particles, provided they are close to the current particle.
- By taking only the force contribution of the centre of mass of a complete tree cell, in case its centre of mass is far enough away.

The choice between these 2 possibilities is made by a special parameter Θ , that should be chosen in such a way that it is never possible that self-interacting forces on a particle may occur.

An example of a quadtree for the problem from the previous sections is given in figure 1.3. The details about the Tree-Code algorithm and the choice of parameter Θ are discussed more extensively in chapter 5.

The Tree-Code method is more accurate than Particle-Mesh, especially for highly non-uniform distributions (provided the parameter Θ is not too small). It generally takes however more calculations per iteration than Particle-Mesh (although this also depends on the number of meshpoints in relation to the number of particles).

Another advantage above Particle-Mesh is that collisions can be much easier dealt with: collision management is in my implementation basically the same as in Particle-Particle.

Chapter 2

General aspects

2.1 Integration methods

To integrate our equations of motion by using discrete timesteps - whatever of the three discussed simulation methods we're going to use - we always need an algorithm to make it possible to *approximate* these equations in our computer program.
There are today many different numerical integration methods, all with their specific advantages and disadvantages, but many of their features can be described in terms of the errors they introduce into the approximation if we compare them with each other, using one fixed timeinterval and one fixed number of integration steps.

With numerical integration techniques we always have to deal with to 2 major sources of error:

- Discretisation errors
- Round-off errors

First I shall describe some numerical integration methods for the equations of motion of our particles and then I'll discuss the 2 major sources of error more lengthy.

2.1.1 Brief discussion of several methods

As said earlier, in my first (somewhat clumsy) multibody simulations I only used Euler and Runge-Kutta 4 for numerical integration. However, they both have some disadvantages : the first has a large discretisation error, so it requires a very small timestep to be accurate enough; the second has smaller discretisation errors, but takes a lot of computations per timestep which is why the round-off error grows much more rapid than with Euler.

A few very important remarks first :

It is important to notice that the acceleration *(*only*)* depends on the relative particle positions (which is always the case for a conservative force), so if we calculate the acceleration *after* an update of the positions,we actually use the approximated acceleration at time $t + \Delta t$ and not at t !

Also, if we update the velocities *before* updating the positions we use the velocity at time $t+\Delta t$ and not at t to find the new positions ! However, updating the positions before updating the velocities is no problem, as the velocities do not directly depend on the positions.

So, in a straight forward scheme the most logical order is:

1. Calculate the new accelerations using the positions
2. Update the positions using the velocities
3. Update the velocities using the accelerations

Note however that this does't apply to the leapfrog and wobble schemes ! In fact these methods make a clever use of these dependencies to reduce the error, by deliberately altering their order.

Euler This method is the simplest one : it only uses the first term of the Taylor expansion (see page 17), so there are only linear terms and only first derivatives in this method. For our type of problems it looks as follows:

$$s(t + \Delta t) = s(t) + v(t) * \Delta t \tag{2.1}$$
$$v(t + \Delta t) = v(t) + a(t) * \Delta t \tag{2.2}$$

Of course the order may swapped, but then we have to realize that we actually use the speed at $t + \Delta t$ for the position calculation.

Heun This method - often called the improved Euler method - just takes the average value at t and $t + \Delta t$ for both the speed and position updates. It is a so-called second order method.

$$s(t + \Delta t) = s(t) + \Delta t * \frac{v(t) + v(t + \Delta t)}{2} \tag{2.3}$$
$$v(t + \Delta t) = v(t) + \Delta t * \frac{a(t) + a(t + \Delta t)}{2} \tag{2.4}$$

Note that we need to store 2 values for both speed and acceleration for each timestep instead of only one as in Euler, so it requires extra memory to keep the old values as well as the new ones.

Runge-Kutta 4 Here we use even more values between t and $t + \Delta t$ to approximate the next step. It is formally a fourth order method. In the formula below we only use 3 points, but in this case the evaluation at $t + \frac{1}{2}\Delta t$ is taken twice,which is just because all functions involved ($a(t)$ and $v(t)$) only take one argument. In other cases however this may not be true and we have to use 4 different points.

$$s(t + \Delta t) = s(t) + \Delta t * \frac{v(t) + 4 * v(t + \frac{1}{2}\Delta t) + v(t + \Delta t)}{6} \tag{2.5}$$

$$v(t + \Delta t) = v(t) + \Delta t * \frac{a(t) + 4 * a(t + \frac{1}{2}\Delta t) + a(t + \Delta t)}{6} \tag{2.6}$$

Note that this not only requires storage for 3 values for both speed and acceleration, but also the extra calculation of the speed and acceleration at half the timestep, and especially the acceleration calculation is a very time consuming step in particle simulations.

Leapfrog This method, which is by far the most popular one for particle simulations, looks like a first order method and also takes the same number of calculations as needed for a first order method, but it can be proven that it is in fact a genuine second order method or even better, at least as good as Heun. Also, under certain conditions the total roundoff error will always be bounded between an upper and a lower limit, so no matter how many iterations we'll carry out, the roundoff error will never exceed a certain extreme, in contrast with i.e. Euler. (see the error propagation discussion in 2.1.4)

The clever thing is that evaluation of speed and position is being done at times that are $\frac{1}{2}\Delta t$ apart from each other, so that they can be viewed as being an average between their value at respectively t and $t + \Delta t$! This trick cancels all the odd terms in the Taylor approximation, only leaving the even terms. For this reason the leapfrog scheme is also said to be "time-reversible". The scheme looks as follows :

$$v(t + \frac{1}{2}\Delta t) = v(t - \frac{1}{2}\Delta t) + a(t) * \Delta t \tag{2.7}$$

$$s(t + \Delta t) = s(t) + v(t + \frac{1}{2}\Delta t) * \Delta t \tag{2.8}$$

In practice this is almost always done by setting the initial velocity to the value $w(0) = v(0) - \frac{1}{2}\Delta t * a(0)$ and then use this new speed instead of the original, so that we can use a scheme very much equal to the Euler scheme above :

$$s(t + \Delta t) = s(t) + w(t) * \Delta t \tag{2.9}$$

$$w(t + \Delta t) = w(t) + a(t + \Delta t) * \Delta t \tag{2.10}$$

Note that the acceleration is now updated *between* calculating the new positions and the new speeds !!

Wobble This is a method I "developed" myself before I knew about the Leapfrog scheme. However, it isn't too bad : little calculations and fairly accurate, so I'd like to mention it also in this report, but unfortunately the Leapfrog scheme seems to be better and also a little faster.
The Wobble scheme doesn't use an offset in speed such as in leapfrog, which could make evaluation of i.e. energies at a certain time a trifle easier (if you don't neglect the offset at all, as I did in most cases ...).
The basic trick is to split the timesteps into two different types: the *even* ones and the *odd* ones. For the even timesteps we use a scheme similar to the Euler scheme above , but for the odd ones we reverse the order of the updates for speed and position.
So, for $t = (2n+1) \cdot \Delta t$ (with n an integer) it looks like :

$$s(t + \Delta t) = s(t) + v(t) * \Delta t \tag{2.11}$$

$$v(t + \Delta t) = v(t) + a(t + \Delta t) * \Delta t \tag{2.12}$$

And for $t = 2n \cdot \Delta t$ (n again integer) :

$$v(t + \Delta t) = v(t) + a(t) * \Delta t \tag{2.13}$$

$$s(t + \Delta t) = s(t) + v(t + \Delta t) * \Delta t \tag{2.14}$$

The accelerations are updated before the scheme itself, so in the odd ones before the position updates and in the even ones before the speed updates.
Note that the Wobble scheme has a better balance and roundoff error than i.e. the Euler scheme, but it still has larger roundoff errors than the Leapfrog scheme.

RKWobble This one uses the same trick as the wobble scheme(s) above, but now with a Runge-Kutta 4 style scheme, instead of a simple first order scheme.

2.1.2 Using only one equation of motion

It is perfectly possible to use only positions and accelerations and completely forget about the speeds, by substituting the speed using the equation for the positions. But in that case we have to remember the previous positions (instead of the speed) to be able to integrate the second-order equation we'll then obtain.

For the Euler scheme in 2.2 this second order scheme looks like :

$$\frac{s(t + 2 * \Delta t) - s(t + \Delta t)}{\Delta t} = \frac{s(t + \Delta t) - s(t)}{\Delta t} + a(t) * \Delta t \tag{2.15}$$

which could be (by recollecting terms and shifting the time over Δt) rewritten as :

$$s(t + \Delta t) = 2 * s(t) - s(t - \Delta t) + a(t - \Delta t) * \Delta t^2 \tag{2.16}$$

And for the Leapfrog scheme in 2.10 this yields in the same way :

$$s(t + \Delta t) = 2 * s(t) - s(t - \Delta t) + a(t) * \Delta t^2 \tag{2.17}$$

The term $s(t) - s(t - \Delta t)$ (which is the difference in position !) in fact represents the velocity term (now multiplied by Δt) that we saw in the original equation schemes.

2.1.3 Discretisation errors

Discretisation errors are caused by the fact that by using an approximation function to predict the new situation after one integration step, we will always loose information if compared with the real situation. The size of the error of course depends on the stepsize : the larger the step, the bigger the error.
But it also depends on the exact form of this approximation function : this is simply a straight line in the case of the Euler method (which is why we call it a first order method), but it is a parabola in the case of the Heun method (called a second-order method) and a 4th-order polynomial in the case of Runge-Kutta 4 (obviously called a fourth order method). In general we can say that the higher the order, the smaller the (discretisation) error.

What makes things worse is that the discretisation error is being propagated: the error made in step 1 will be amplified in step 2 and so on.

The *local* discretisation error (thus the error for one timestep) is given by the following general formula :

$$\epsilon_{n+1} = \frac{1}{(r+1)!}\Phi^{(r+1)}(n) \cdot (\Delta t)^{r+1} \tag{2.18}$$

where r is the order of the integration method, $\Phi()$ is a function that gives the analytical solution of the differential equation and thus $\Phi^{(r)}(x_n)$ the value of the rth derivative of function Φ for step n, and ϵ is the difference between the value we'll find at step $n+1$ using the numerical method and the exact functionvalue of x at step $n+1$.

This looks rather horrible at first glance, but it is more easy to see this if we realize that basically *every* numerical approximation uses only the first few terms of the infinite Taylor series for $\Phi()$ to achieve the approximation :

$$\Phi(n+\Delta t) = \Phi(n) + \Phi^{(1)}(n) * (\Delta t) + \frac{1}{2!}\Phi^{(2)}(n) * (\Delta t)^2 + \frac{1}{3!}\Phi^{(3)}(n) * (\Delta t)^3 + ... \tag{2.19}$$

A first order method only uses the zero and first term of the Taylor series, a second order the zero, first and second term and so on. So, it must be obvious that the difference between the approximation and the exact value is dominated by the *first* term of the *neglected* part of the full Taylor series, i.e. the second term for a first order system, the third for a second order system and so on !!

Well, now we know the error made in one step, but of course we are more interested in the *total* discretisation error (this is the *accumulated* error after a number of steps) for a certain number of timesteps. Fortunately that looks a little easier:

For $p = \frac{t_{n+p} - t_n}{\Delta t}$ timesteps the total discretisation error is roughly given by the formula :

$$\epsilon_{n+p} \leq p \cdot \epsilon_{n+1} = \frac{t_{n+p} + t_n}{\Delta t} \cdot \epsilon_{n+1} = c * (\Delta t)^r \tag{2.20}$$

with c being some unimportant constant and r the *order* of the integration method. The latter part of the equation was found by substituting the expression 2.18 for ϵ_{n+1}.

Note that the order of the global discretisation error is one lower than the order of the local discretisation error. Also, in particle simulation we may say in general that the global discretisation error can never be bounded for an unbounded number of timesteps, due to the nature of the problem.

More information about this subject can be found in many books about differential equations and numerical methods, such as in Boyce & Diprima ([6]) : "Numerical methods", and Strauss ([16]) : "Computation of Solutions".

2.1.4 Round-off errors

Apart from discretisation errors, we also see a very different source of errors, introduced by the computer itself. Computers can almost never calculate exact answers: they are only accurate up to a limited number of digits, so every calculation introduces a round-off error. It is obvious that the more calculations we have to make in *one* timestep, the larger the maximum roundoff error for one timestep will be.

In some cases it is possible that the total round-off error will *never* exceed a certain maximum: in that case we will call the integration method **stable.**

In the case of our particle simulations several methods can be stable, provided the timestep is small enough. Stability often depends on the size of the timestep: if the timestep gets too large possibly stable systems will also become unstable.

Let's view the error-propagation for Euler and Leapfrog a little closer :

2.1.4.1 Euler error-propagation

We can rearrange the Euler scheme to obtain a second-order difference equation, instead of two first order equations, simply by combining the two first order equations from the scheme into one equation, as already shown in 2.16. This equation could be rewritten to :

$$s(t + \Delta t) - 2 * s(t) + s(t - \Delta t) = a(t - \Delta t) * \Delta t^2 \tag{2.21}$$

The round-off error at step n (called ϵ_n) is given by subtracting the approximation $s_n^{'}$ we found on the computer (using a limited number of digits) from the exact approximation s_n (using an infinite number of digits) :

$$\epsilon_n = s_n^{'} - s_n$$

If we substitute this into 2.21 and bear in mind that the acceleration only depends on the positions we'll get :

$$\epsilon_{n+1} - 2 * \epsilon_n + \epsilon_{n-1} = (a(s_{n-1} + \epsilon_{n-1}) - a(s_{n-1})) * \Delta t^2$$

Which is more or less equal to (approximating the second part using the standard definition for the derivative) :

$$\epsilon_{n+1} - 2 * \epsilon_n + \epsilon_{n-1} = \frac{\delta\, a(s_{n-1})}{\delta\, s_{n-1}} \epsilon_{n-1} * \Delta t^2$$

Well, this is a standard second order difference equation. Assuming solutions of the form $E = a\lambda_1^n + b\lambda_2^n$ and substituting the term $\frac{\delta\, a(s_{n-1})}{\delta\, s_{n-1}}$ with it's maximum amplitude Ω_E^2, this yields us :

$$\lambda * \epsilon_n - 2 * \epsilon_n + \frac{1}{\lambda}\epsilon_n = \frac{1}{\lambda}\epsilon_n * (\Omega_E \Delta t)^2$$

or (dividing by $\epsilon_n$and multiplying with λ) :

$$\lambda^2 - 2 * \lambda + 1 - (\Omega_E \Delta t)^2 = 0$$

This quadratic equation only has complex roots for $4 - 4 * (1 - (\Omega_E \Delta t)^2) < 0$ or $(\Omega_E \Delta t)^2 < 0$, which of course has no solutions. This implies that the error in the Euler scheme will at any rate be *always* **nonperiodic**, so unless both roots are smaller than 1 (in which case the error is being damped out) the Euler scheme will be always unstable.

Well, as both roots are only between ± 1 for $1 \pm \frac{1}{2}\sqrt{(\Omega_E \Delta t)^2} < 1$ and because this equation obviously has no valid solutions for both roots at the same time, this must lead to the conclusion that Euler is in our case *always* **unstable**.

2.1.4.2 Leapfrog error-propagation

If we take the Leapfrog scheme from equation 2.17 we'll find in the same way as above :

$$s(t + \Delta t) - 2 * s(t) + s(t - \Delta t) = a(t) * \Delta t^2 \tag{2.22}$$

and thus :

$$\epsilon_{n+1} - 2 * \epsilon_n + \epsilon_{n-1} = (a(s_n + \epsilon_n) - a(s_n)) * \Delta t^2$$

Which is more or less equal to (approximating the second part using the definition for the derivative) :

$$\epsilon_{n+1} - 2 * \epsilon_n + \epsilon_{n-1} = \frac{\delta\, a(s_n)}{\delta\, s_n}\epsilon_n * \Delta t^2$$

This is again a standard second order difference equation. Let's assume again solutions of the form $E = a\lambda_1^n + b\lambda_2^n$ and also substitute the term $\frac{\delta\, a(s_n)}{\delta\, s_n}$ with it's maximum amplitude $-\Omega_L^2$, as we are only interested in this extreme. This yields us in the same way as with Euler :

$$\lambda * \epsilon_n - 2 * \epsilon_n + \frac{1}{\lambda}\epsilon_n = \epsilon_n * -(\Omega_L \Delta t)^2$$

or (dividing by $\epsilon_n$and multiplying with λ) :

$$\lambda^2 + ((\Omega_L \Delta t)^2 - 2) * \lambda + 1 = 0$$

This quadratic equation has complex roots for $((\Omega_L\Delta t)^2 - 2)^2 - 4 < 0$ or $(\Omega_L\Delta t)^4 - 4(\Omega_L\Delta t)^2 < 0$ which equals to $(\Omega_L\Delta t)^2((\Omega_L\Delta t)^2 - 4) < 0$. The first solution is trivial, the second yields to $|\Omega_L\Delta t| < 2$. As long as this is true the Leapfrog scheme will be **stable**, because the magnitude of the roots are in that case always on the unit-circle. Note that $\Omega_L\Delta t$ can't be negative, so there is no reason to investigate this any further.

For $|\Omega_L\Delta t| > 2$ both roots are real and one of them is moving from -1 to $-\infty$, while the other moves from -1 to 0, so in this case we always have one root with a magnitude larger than 1. Thus the Leapfrog scheme will be unstable for $|\Omega_L\Delta t| > 2$.

Of course this leaves us with the problem of finding Ω_L, but for the moment it is enough that we know that under certain conditions this scheme is always stable with respect to the round-off error. Thus, how much it may look the same as Euler above it is definitely different at this point !!

It is important to realize that short simulations may be better carried out with an unstable method that has a small roundoff error for the given interval, while for long simulations stability may be a far more important criterion than a small roundoff error over a certain interval. (see picture 2.1)

2.2 How accurate is our simulation ?

It is of course very important to be able to say something about how accurate a certain simulation run resembles the true situation. The best of course would be to have a system that could be solved analytically, so we could compare every value at every timestep with the

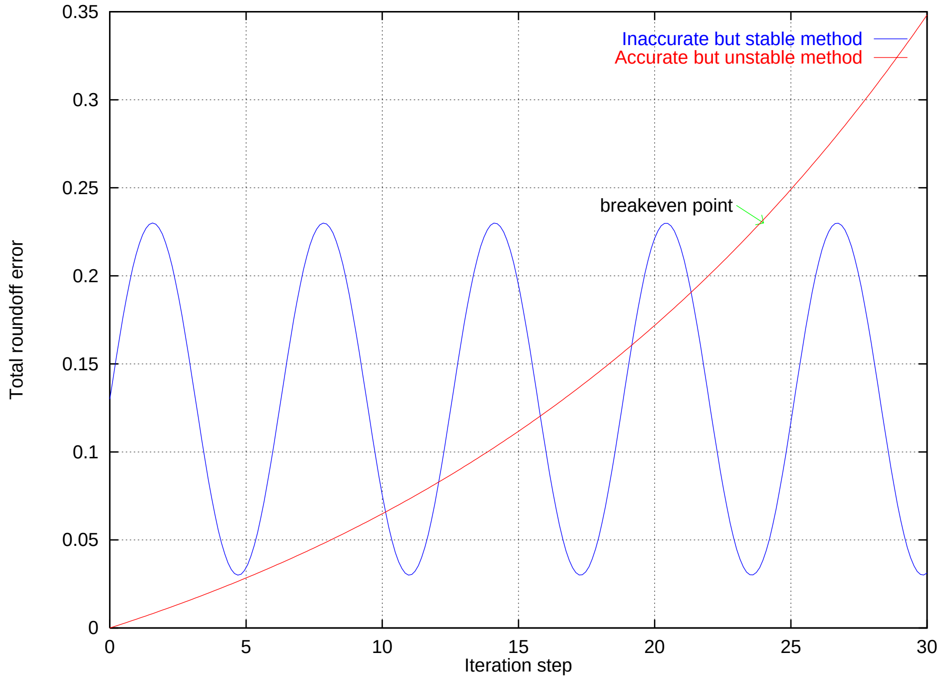


Figure 2.1: Comparison between a stable method with a large roundoff error and unstable method with a small roundoff error. The tradeoff is at about 24 iterations in this example.

correct value if we wish, but of course it would be pretty useless to carry out a simulation if we already knew the answer !

However, these kind of problems can be used as a kind of testcase for a certain program, before trying it on less trivial problems. What I often used was a system where a very light particle was running in a circle-shaped orbit around a very heavy particle at rest. Every deviation from the circle would be an indication of certain errors in the program.

What could be very useful too is to apply basic physical conservation laws on the simulation : at every time both the energy and the momentum in all directions should be conserved, so calculating these at a certain point in the simulation and compare them with their initial values may give a lot of information about the accuracy.

2.2.1 Momentum conservation

At every moment the total momentum of the system will be given by:

$$\vec{p}_{tot} = \sum_{i=1}^{N} \vec{p}_i \tag{2.23}$$

Simple, linear or first order integration schemes, combined with an exact algorithm as Particle-Particle, will in practice always conserve momentum: if particles interact, the force from particle a on particle b will be - according to Newton's 3th law - always equal but opposite to the force from particle b on particle a. This implies that both also have the same magnitude of error. Well, as these errors are always opposite to each other, they will cancel each other if we sum all momentums to find the total momentum !!

It is possible to only compare the amplitude of the total momentum as I did initially, but it is generally better to compare the momentum in every direction separately as this doesn't take any more time and memory. Due to a bug in the code I swapped the x and y components of the velocity in one case but wasn't able to notice it, as it had no influence on the amplitude of the total momentum ! It was only after this that I decided to compare the momentum in every direction separately ...

Higher order schemes have more problems in conserving momentum, but the nice thing about leapfrog and also wobble is that - although they are mathematically second order - they have the momentum conservation behaviour of a first order method !

But be careful: it is important to realize that - even if the total momentum is conserved - the individual momenta may still contain a large error ! However, fortunately enough this will often result in a large error in the total energy.

2.2.2 Energy conservation

At every moment the total energy of the system will be given by the sum of the kinetic and potential energies. The potential energy is assumed to be zero at infinity and thus always negative, yielding us for a normal 3D system :

$$E_{tot} = \sum_{i=1}^{N} \frac{1}{2} m_i \cdot v_i^2 - \sum_{i=1}^{N} \sum_{j \neq i}^{N} \gamma \cdot \frac{m_i \cdot m_j}{|r_{i \to j}|} \tag{2.24}$$

For a 2D system with genuine 2D forces such as we find for example in a 2D Particle-Mesh simulation (where the force drops $\sim \frac{1}{r}$ instead of $\sim \frac{1}{r^2}$) the pairwise potential energy is proportional to $\log |r_{i \to j}|$, resulting in a slightly diffferent formula :

$$E_{tot} = \sum_{i=1}^{N} \frac{1}{2} m_i \cdot v_i^2 - \sum_{i=1}^{N} \sum_{j \neq i}^{N} \gamma' \cdot m_i \cdot m_j \cdot \log |r_{i \to j}| \tag{2.25}$$

In contrast with the momentum above the energy is much less likely being conserved :

Problems arise with the kinetic energy (the first part in both formulas) as it squares the speed, thus dropping the sign in the error, so that the cancellation that worked so nicely with the momentum doesn't work here anymore. Also the relative error is doubled by taking the square of the speed, which makes things even worse.

For the potential energy term (the second part in the formulas) there is also a problem : as it use the absolute value for the distance, the sign in the error is dropped here too, causing the same effect as with the kinetic energy above.

Note that the computation of the potential energy is an $O(N^2)$ operation. This implies that with large problems ($N > 10^4$) this may take a relatively large amount of time, which is why I used an approximation for this value in the case of the analyser program described in chapter 6.

Furthermore the discretisation error often causes the system to slowly heat up, for reasons not shown in this report. This also contributes to an increasing deviation from the initial energy. Some algorithms solve this heating problem by decreasing the per-particle energy after a certain number of timesteps with a small factor, but as I consider this a very inelegant solution I did not implement this.

So, in general we may say that energy is not conserved, whatever scheme we use. On the other hand, if the deviation is not to large, the simulation may still be quite accurate. Deviations as large as 1-5 % are no problem in many cases.

2.3 Long range versus short range forces

It is obvious that a force that drops with the seventh power of the distance will be far less important for far away particles, than a force that drops only with the square of the distance, like i.e. a gravitational force. In fact the influence of the first one may be completely ignored for all particles except for the "nearest neighbours", while the second may still be felt at a considerably longer distance.

So, we see that the distance over which the force can influence other particles can differ very widely. A force that is only felt at close locations could make it possible to use a much faster algorithm by totally neglecting the force at distances higher than a certain threshold. Such a force will be called a *short range force* from now on. On the other hand a force could work over a very long distance and this one is obviously called *long range force.*

Special attention should be paid to the density: even if a force drops sharp with the distance it couldn't be neglected if the number of far away particles is much higher than the number of close particles, although it can be possible in some cases to only take into account the total influence of many far away particles, as we do in Tree-code simulations.

Special problems concerning a (more or less) uniform density in 2D and 3D space will be discussed hereafter :

2.3.1 2D and 3D aspects of uniform density

Consider a system of particles with a uniform density for the moment. This means that the concentration of particles is more or less the same over the whole working area.

Note: In a purely uniform distribution all contributions will cancel out each other, as for

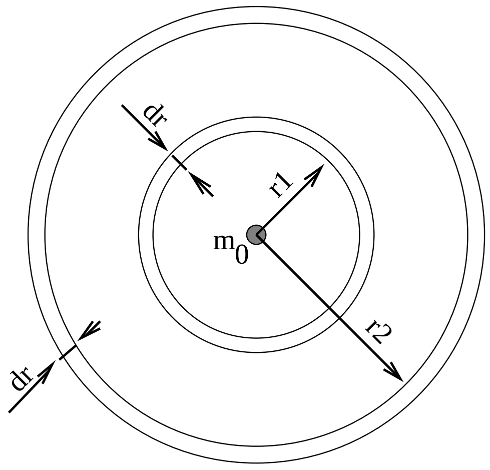


Figure 2.2: Influence of particles at certain distance $r \rightarrow (r + dr)$ on the total force.

every force contribution from a certain distance there is another contribution from exact the opposite side, effectively cancelling the first one. So only for places where the distribution is *not* purely random the discussion hereafter will hold completely (i.e. near the edges), although what it says about each *individual* force contribution will always be true, although not very useful.

Now let's see what happens to the total force from all particles at a given distance on a certain test particle, if we assume that the force drops with the square of the distance (as is the case with the 3D gravitational force) both for a 2D system and a 3D system.

2D system Consider the density to be given by $\rho = \frac{N}{A}$, with N the total number of particles and A the total area of our 2D space. Then we may write for the number of particles in a small band from circle r to circle $r + dr$:

$$N_{r \rightarrow r+dr} = \rho \cdot dA = \rho \cdot d(\pi \cdot r^2) = \rho \cdot 2\pi \cdot r \cdot dr \tag{2.26}$$

And for the (3D) force of one particle in such a band on the particle in the centre, assuming all other masses are equal :

$$F_{r \rightarrow r+dr} = -\gamma \frac{m_0 \cdot m}{r^2} \tag{2.27}$$

Which gives us for the total force from all particles in a small *ring* $r \rightarrow r + dr$:

$$F_{tot_{dr}} = N \cdot F_{r \rightarrow r+dr} = -\frac{\gamma \cdot \rho \cdot 2\pi \cdot m \cdot m_0}{r} \cdot dr \tag{2.28}$$

This is for the bands r_1 and r_2 in figure 2.2 respectively :

$$\begin{aligned} N_{r_1} &= \rho \cdot 2\pi \cdot r_1 \cdot dr \\ N_{r_2} &= \rho \cdot 2\pi \cdot r_2 \cdot dr \end{aligned}$$

So the total forces become respectively (assuming $m = m_0$) :

$$\begin{aligned} F_{tot_{dr_1}} &= N_{r_1} \cdot F_{r_1} = -\gamma \cdot \rho \cdot 2\pi \cdot m^2 \cdot \frac{1}{r_1} \cdot dr \\ F_{tot_{dr_2}} &= N_{r_2} \cdot F_{r_2} = -\gamma \cdot \rho \cdot 2\pi \cdot m^2 \cdot \frac{1}{r_2} \cdot dr \end{aligned}$$

If we divide the second equation by the first one, we find an expression for the change of the total band force as a function of the distance of the band :

$$\frac{F_{tot_{dr_1}}}{F_{tot_{dr_2}}} = \frac{r_2}{r_1} \tag{2.29}$$

or, in words: for a 2D system the influence of a band at distance r is proportional to the inverse of the distance r to the centre of the band. This implies that the force of far away particles is less important than from nearby particles.

3D system Now consider the density to be given by $\rho = \frac{N}{V}$, with N the total number of particles and V the total content of our 3D space. Then we may write for the number of particles in a small volume from the sphere at r to the sphere at $r + dr$:

$$N_{r \to r+dr} = \rho \cdot dV = \rho \cdot d(\frac{4}{3} \cdot \pi \cdot r^3) = \rho \cdot 4\pi \cdot r^2 \cdot dr \tag{2.30}$$

Which gives us for the total force of all particles in a small *volume* $r \to r + dr$ (assuming again a gravitational force) :

$$F_{tot_{dr}} = N \cdot F_{r \to r+dr} = -\gamma \cdot \rho \cdot 4\pi \cdot m \cdot m_0 \cdot dr \tag{2.31}$$

And this leads of course to :

$$\frac{F_{tot_{dr_1}}}{F_{tot_{dr_2}}} = 1 \tag{2.32}$$

or, in words: for a 3D system the influence of a band at distance r is *not proportional to the distance* in the gravitational/coulomb case !! This implies that for a uniform density the force contribution of far away particles is exactly as large as the contribution from nearby particles !!

So, a short-range force in a 2D simulation could turn into a long-range force in a 3D system without changing anything else ! This is important to bear in mind if - as I did in this report - only 2D simulations are being considered: in 3D systems with gravitational and electrical forces and a more or less uniform density, the influence of far away particles can *never* be neglected !

It is obvious that with forces that drop more progressively than $\frac{1}{r^2}$ forces, the faraway particles can be again safely neglected and it is equally obvious that in simulations with a higher dimension than 3 (whatever it may stand for !) also these more progressive forces sometimes can't be ignored anymore.

2.4 How to choose the timestep

The exact size of the timestep determines the amount of "linearisation" introduced by the numerical integration if compared with the real problem. Of course this shouldn't be too far off from reality. On the other hand: the larger the timestep, the less calculations we need and thus the faster our simulation will be. What we should ask ourselves is how large can we make it without causing serious problems ?

It is obvious that for a particle with a constant speed and no forces working on it we can make our timestep as large as we like. So, the amount of *change* in speed within one timestep determines how accurate our approach will be : i.e. if the speed doubles within one timestep the results will be very unreliable.

From this follows that we shouldn't allow the speed to change more than about 5-10 % in one timestep. But, this is impossible for a particle initially at rest ! To avoid this problem we'd better use an *absolute* maximum than a relative maximum. We could i.e. look at the average initial speeds and choose the maximum allowable change according to this average.

Also important is the change in the angle or direction: if the magnitude of the speed hasn't changed it is still possible that the direction has changed considerably. Of course we have the same problem here as above: for a particle initially at rest it is impossible to say something about the relative change in direction.

The easiest thing is to look at the absolute change of speed in each separate direction: in that case we will cover both the magnitude and the direction criteria.

Another aspect in choosing the timestep is the closest distance between 2 particles that we allow (the `COLLISIONDETECT` macro in the code). If the speed is so large that in one timestep a particle has moved more than the collision detect distance, there could have been a collision between 2 particles without any detection, causing the new positions to be meaningless ! This is even a bigger problem in the case of our gravitational or Coulomb forces: the interparticle force goes to ∞ if the particle distance goes to 0 !

So we never should allow the position to change with more than about 10-25 % of the collision detect distance.

It may be a good idea to relate the collision detect distance to the maximum allowable acceleration: i.e. 2-5 times as large, rather than make it a fixed value that could be only changed at compile time.

Another important restriction for the choice of the timestep was already given earlier in this chapter: it should be small enough to avoid round-off errors to become unbounded.

2.4.1 Particle-Mesh considerations

If we use an algorithm where the interparticle distance doesn't play a direct role in the computation of the acceleration -such as is the case with the Particle-Mesh method - a minimum collision distance will be meaningless.

However, the Particle-Mesh method has another limit instead: it should never be possible that a particle moves with more than 10-25 % of the mesh distance. This immediately brings us to another problem: if we halve the mesh distance, we should also halve the timestep !

2.4.2 Variable timesteps

In the discussion above we simply stop the simulation if one of the limits is being exceeded. Another possibility could be to make the timestep larger or smaller according to the force at a certain moment and thus keep the change in speed per timestep more or less constant. We could do this for the whole system of particles, but this could easily result in an unacceptable increase of the total running time. However, it is also possible to assign each particle it's own individual timestep, which is of course much more efficient, although this results in many practical problems, especially with the Tree-Code and Particle-Mesh codes:

- At what timestep should we update the Tree or Mesh ? If we update at the smallest individual timestep the Tree or Mesh will always be up to date, but this would result in an enormous increase of the total runtime, as the Tree/Mesh updates are the most time consuming steps.
- At every moment some parts of the Tree or Mesh are out of date. What is the exact impact of this fact on the total accuracy of the simulation ?
- The Leapfrog and Wobble schemes are based on having fixed timesteps. Many of their advantages such as the second order discretisation error and possible stability for round-off error propagation, will vanish if the timesteps aren't fixed anymore. (This also applies to system-wide variable timesteps)

For these reasons I decided to refrain from using a variable timestep. After all one of the goals was to build source code that could be easily understood and adapted by newbies and the implementation of this - although not impossible - would make it far less readable.

2.5 Collision management

If particles come very close to each other, their interaction may show a more complex behaviour than the simple gravitational or Coulomb interaction. For example extra, more progressive repulsive components may occur, as is the case with interacting atomic particles.

It is also possible that they collide using the simple mechanical collision laws for either central elastic or central non-elastic collisions as is the case with particles that interact through the gravitational interaction. Partly non-elastic collisions and non-central collisions are very complicated to deal with and are considered to be beyond the scope of this report.

Finally we could ignore all collisional effects if they don't play a significant role, as we do in Particle-Mesh.

Some of the techniques will be discussed in short hereafter :

2.5.1 Collisionless systems: Particle-Mesh

As the Particle-Mesh method ignores pairwise interaction (it can only be applied to systems of more or less uniform distributed particles), it is not possible to deal with collisions here.

In fact we should stop the particle-mesh simulation if to many particles come to close to each other : it is supposed to be used only for (almost) **collisionless** systems.

Unfortunately, checking the relative positions to determine if we have to stop, can only be done if we check the pairwise interparticle distance and that was exactly what we wanted to avoid as it introduced the $O(N^2)$ scalability of the Particle-Particle method. But, this can be improved if we only check for particles in the current cell and its neighbours, but even this causes a lot of extra calculations.

This is being done in a more enhanced version of Particle-Mesh, that uses the Particle-Particle approach for particles close to each other : **Particle-Particle-Particle-Mesh** or $\mathrm{P^3M}$, so here it would be possible to add collision management, but also the $\mathrm{P^3M}$ method is beyond the scope of this report.

2.5.2 Elastic collisions

Elastic collisions occur if colliding particles do not change their individual particle mass and shape during the collision : only kinetic energy and momentum are being exchanged and nothing else. Note that elastic collisions can only be calculated for *central* collisions: for non-central forces we need more information about the change in direction for at least one of the particles.[1]

However, this is in general no problem for 2 different reasons :

- If particles come very close they will interact very strong through either the gravitational or electrical forces, thus changing each other's direction in such a way that they move almost in opposite directions at the time of the collision and thus the collision will be almost a frontal one, provided the distance at the time of collision is small enough. (see picture 2.3)

- In discrete particle simulations the particles are often regarded as being dimensionless: i.e. their radius is zero. In that case the only possible collision is a central one or there wouldn't be any collision at all !

[1] This extra information could be supplied by using a random generator.

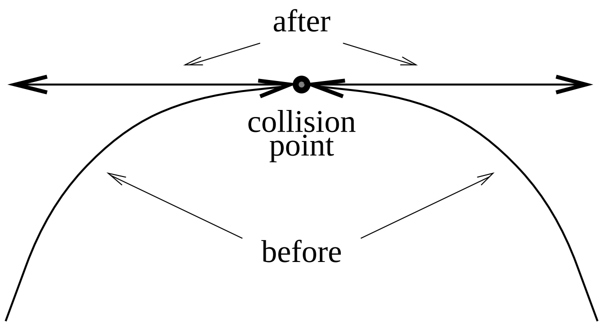


Figure 2.3: Elastic collisions between attracting particles are often almost central.

Larger problems arise through the necessity of small collision distances : this requires the timestep to be small as well. But this is already discussed in section 2.4 at page 26.

How should we calculate the new speeds after a central elastic collision occurred ?

Although this is plain classical mechanics I'll discuss this briefly hereafter, to save other people the time of either searching for this in other places or to derive this themselves :

From the laws of energy and momentum conservation it follows that at every time the total momentum in all directions and the total energy should be all constant. So, for the momentum and energy *before* and *after* the time of impact we may write :

$$\begin{aligned}
m_1 \cdot w_{x_1} + m_2 \cdot w_{x_2} &= m_1 \cdot v_{x_1} + m_2 \cdot v_{x_2} \\
m_1 \cdot w_{y_1} + m_2 \cdot w_{y_2} &= m_1 \cdot v_{y_1} + m_2 \cdot v_{y_2} \\
m_1 \cdot w_1^2 + m_2 \cdot w_2^2 &= m_1 . v_1^2 + m_2 \cdot v_2^2
\end{aligned}$$

Here 'v' denotes the speed before collision, 'w' the speed after collision.

Also, if we transfer this system to the centre-of-mass system (with speed v_c), the total momentum within this centre-of-mass system must be necessarily zero. This makes the computation quite easy as from energy conservation follows that - although the direction of the momentum may change - the magnitude of the momentum stays the same for all individual particles :

$$\begin{aligned}
w_{x_1} - v_{c_x} &= -(v_{x_1} - v_{c_x}) \\
w_{y_1} - v_{c_y} &= -(v_{y_1} - v_{c_y}) \\
w_{x_2} - v_{c_x} &= -(v_{x_2} - v_{c_x}) \\
w_{y_2} - v_{c_y} &= -(v_{y_2} - v_{c_y})
\end{aligned}$$

Note that in the centre-of-mass system the collision is always a frontal collision ! The centre-of-mass speed $\overrightarrow{v_c}$ used in the equations above is given by :

$$v_{c_x} = \frac{m_1 \cdot v_{x_1} + m_2 \cdot v_{x_2}}{m_1 + m_2} \tag{2.33}$$

$$v_{c_y} = \frac{m_1 \cdot v_{y_1} + m_2 \cdot v_{y_2}}{m_1 + m_2} \tag{2.34}$$

And this altogether yields us the equations that enable us to calculate the new speed components after a central elastic collision between two particles :

$$w_{x_1} = \frac{(m_1 - m_2) \cdot v_{x_1} + 2 \cdot m_2 \cdot v_{x_2}}{m_1 + m_2} \tag{2.35}$$

$$w_{x_2} = \frac{(m_1 - m_2) \cdot v_{x_2} + 2 \cdot m_1 \cdot v_{x_1}}{m_1 + m_2} \tag{2.36}$$

$$w_{y_1} = \frac{(m_1 - m_2) \cdot v_{y_1} + 2 \cdot m_2 \cdot v_{y_2}}{m_1 + m_2} \tag{2.37}$$

$$w_{y_2} = \frac{(m_1 - m_2) \cdot v_{y_2} + 2 \cdot m_1 \cdot v_{y_1}}{m_1 + m_2} \tag{2.38}$$

2.5.3 Non-Elastic collisions

Another possible interaction could be that the two colliding particles become “clogged” and continue from now on as if they were one new particle with the total mass, speed and energy of the old ones. There are 2 ways to achieve this without reducing the number of particles (which is sometimes difficult from a software point of view, especially on a distributed memory machine) :

1. We can give the new particle information to one of the old particles and set the mass and speed of the other one to zero, effectively flagging the last one as being unused.
2. We can equally divide the mass between the two old particles and give them both the same new speed and position. This insures they will always stay together as if they were one new particle.

However, I found that from a practical point of view the second method is less attractive, because of the fact that their relative distance is zero, which introduces a singularity for the pairwise potential energy between these two particles that can easily confuse our computer simulation. For this reason I only used method 1 in practice.

Also note that all non-elastic collisions have another impact on the pairwise potential energy: non-elastic collisions do *not* conserve energy ! This implies that checking the total energy before and after the simulation can be pretty useless, unless the internal energy stored in the joined particles is also taken into account.

As for the equations for the non-elastic collision: they are even more simple than those for the elastic one.

The new mass for both particles becomes :

$$\begin{aligned} m_1 &= m_1 + m_2 \\ m_2 &= 0 \end{aligned}$$

The new speed can be found by assignment of the total momentum to one particle and then divide this by the total mass and give the other one a zero speed :

$$w_{x_1} = \frac{m_1 \cdot v_{x_1} + m_2 \cdot v_{x_2}}{m_1 + m_2} \tag{2.39}$$

$$w_{x_2} = 0 \tag{2.40}$$

$$w_{y_1} = \frac{m_1 \cdot v_{y_1} + m_2 \cdot v_{y_2}}{m_1 + m_2} \tag{2.41}$$

$$w_{y_2} = 0 \tag{2.42}$$

And the new position is simply that of the centre-of-mass for one and an arbitrary value (0 !) for the other :

$$u_{x_1} = \frac{m_1 \cdot s_{x_1} + m_2 \cdot s_{x_2}}{m_1 + m_2} \tag{2.43}$$

$$u_{x_2} = 0 \tag{2.44}$$

$$u_{y_1} = \frac{m_1 \cdot s_{y_1} + m_2 \cdot s_{y_2}}{m_1 + m_2} \tag{2.45}$$

$$u_{y_2} = 0 \tag{2.46}$$

Note that the change of mass needs some special arrangements on a distributed memory system : normally only the particle positions need to be updated from the other slaves, but if we allow non-elastic particles we should also update the particles masses at every iteration step !

2.5.4 Extra repulsive components

As said already: it is also possible to add an extra repulsive component to the force. This extra component should only become important at short distances and so it should drop faster with the distance than the $\frac{1}{r^2}$force. To avoid this component becomes significant at distances of about 1 , it also needs a small constant. The total force then becomes something like :

$$F_{tot_{1\to 2}} = -\gamma \cdot \frac{m_1 \cdot m_2}{r_{1\to 2}^2} + \frac{\gamma}{c_1} \frac{m_1 \cdot m_2}{r_{1\to 2}^{c_2}} \tag{2.47}$$

With c_1 quite large (i.e. > 1000) and c_2 larger than 2 (i.e. 4).

The calculation of the extra force component takes some more time if we do it for all particle interactions, but it is perfectly possible to only calculate this for particles that are close to each other, provided the component is small enough. We have to check for this distance anyway, so it virtually doesn't add any extra calculations to our simulation.

Note that this extra component should also be taken into account for the total energy computation (see equation 2.24) or else problems may arise with energy conservation, although the contribution on larger distances is often neglectable.

In practice it seems that this method is probably the most widely used type of collision management, although I choose the elastic collision to be the default for my simulations.

Chapter 3

Particle-Particle method

3.1 Introduction

The Particle-Particle method is the oldest and most basic particle simulation technique. It is a very straight forward way to describe the behaviour of a system of particles that attract or repulse each other through pairwise interactions.

The pairwise force from particle i on particle j is given by :

$$\vec{F}_{i\to j} = -\vec{F}_{j\to i} = c \cdot \frac{q_1 \cdot q_2}{|r_{i\to j}|^3} \cdot \vec{r}_{i\to j} \tag{3.1}$$

In this formula c is a constant that determines the direction and scale of the force : negative values result in attracting forces, positive values in repulsive forces. The factor q is some property of the interacting particle (i.e. mass or charge) and $|r_{i\to j}|$ the length of the vector connecting both particles. The last term calculates the different components of the force in different directions (X, Y and possibly Z).

The total force on one particle in a system of N particles is the vector sum of all separate contributions as given by 3.1 , resulting in :

$$\vec{F}_{i_{tot}} = \sum_{i\neq j}^{N} F_{i\to j} \tag{3.2}$$

If we know the total force acting on a particle at a certain time, we can use this to calculate the acceleration working on that particle :

$$\vec{a}_{i_{tot}} = \frac{\vec{F}_{i_{tot}}}{m_i} \tag{3.3}$$

Note that this m is always the mass of the particle, no matter if we calculate gravitational or Coulomb forces ! It may sound trivial, but I saw people using the charge instead : this happens easily if we calculate the acceleration directly, without using the force as an intermediate. This is correct for gravitational forces, but not for Coulomb forces !!

After that we can determine the new speed and position using one of the integration schemes as discussed in 2.1 . Roughly the complete algorithm looks as in table 3.1:

```
- calculate the total energy and momentum
- set the timecounter to 0
- while ( timecounter < endtime )
  - for all particles do
    - set running total for acceleration a to 0
    - for all particles except the current one do
      - calculate the distances in all directions (r_x,r_y and r_z)
      - calculate the length of the vector r = sqrt(r_x^2 + r_y^2 + (r_z^2))
      - calculate the acceleration contribution
        and add to the running total for a
    - endfor
    - calculate new speed and position using a suitable scheme
  - endfor
  - write or plot the new particle positions
  - increase timecounter with the timestep
- endwhile
- check the energy and momentum for conservation
```

Table 3.1: Flowchart Particle-Particle code

3.2 Implementation aspects

During development several minor improvements were added :

- The code uses a signal handler to force the calculation of the end momentum and energy and shows this together with the starting energy and momentum and some other information as soon as the simulation is ended prematurely (i.e. by pressing Ctrl-C).
- To be able to use arbitrary testfiles, all necessary arrays are being dynamically allocated. This required a few extra lines, but it made the code much more flexible.
- Particles with zero mass are skipped in the total force calculation and in the energy and momentum calculation. This is done to provide a mechanism through which we may

perform a non-elastic collision, without the need to remove particles physically from our particle list.

3.2.1 Tunable parameters

There are many variables that can be used to alter the specific behaviour of the simulation. In the datafile the stepsize, the number of iterations and the number of iterations between a screen update can be specified. For the exact meaning of the datafile layout, see the appropriate appendix in A.

Commandline options :

-i <nr> it is possible to ask for a specific integration method with the `-i` option. This needs an additional number, that is currently given by the following enumerator list : **DUMMY (0), EULER (1), LEAPFROG (2), WOBBLE (3), RUNGA1 (4), RUNGA2 (5), RKWOBBLE (6)**
It is by default set to the Leapfrog integration scheme (which is number 2).

-a This causes the graphics display to be cleared before every new update, so the result looks like an animation instead of a cumulative position plot.

-c <nr> The type of collisionmanagement to be used. Can be either an elastic collision (1), a non-elastic collision (2) or the use of an extra repulsive component (3). Makes it possible to choose another collision management algorithm for particles that come within the COLLISIONDISTANCE. (See section 2.5) The default is currently to perform a complete elastic collision.

-x This switch suppresses all graphics output. It is useful for i.e. batchprocessing, if only the total running time and total error in energy and momentum are of interest. Without this switch the program can only run if an X-windows capable terminal is used and the DISPLAY variable is set.

-f Skip the energy and momentum computation at the start and the end. Omitting the energy computation isn't of much use with Particle-Particle, but it can save an enormous amount of time in case of the Particle-Mesh and Tree-Code programs for the larger data sets, as the computation of the energy is of order $O(N^2)$!

-o <file> Write all particle coordinates to the file specified at every plotstep. Note that this not necessarily has to be at every iteration !

At compile time there are several other parameters that can be set :

MAGX and MAGY These determine the scaling factor that is being used to plot the particle orbits on the screen. They are for historical reasons both set to 150. It is possible to change the aspect ratio by these changing parameters if necessary. All testfiles are chosen in such a way to fit exactly on the plot window if the default scaling factor is used. (See also the appendix about scaling of parameterfiles)

USE3DFORCES With normal gravitational or electrical problems we use a 3D force, dropping with the square of the distance ($F \sim \frac{1}{r^2}$). But in a 2D simulation using the Particle-Mesh method, the force only drops with the absolute value of the distance ($F \sim \frac{1}{|r|}$). To be able to compare the results of PP and PM, this macro should be undefined. In that case also PP will use forces proportional to $\frac{1}{|r|}$.

COLLISIONDISTANCE This is the minimum distance allowed between two particles, to avoid the acceleration gets to high. One of the collision mechanism routines as described in section 2.5 is called as soon as this distance is reached.

LAMBDA This is the constant c as used in the interparticle force calculation (see equation 3.1). It is currently set to the Cavendish constant, that determines the size of a gravitational force. Note that the code assumes attractive forces by default, so the minus sign is embedded in the code itself.

DEBUGLEVEL A value of 0 is the default: it shows no debugging information. A value of 1 shows some more information about the simulation, but still shows the graphical output as well. A value of 2 shows even more debug information and also disables the graphical output.

Although it is possible to use a variable timestep in the Particle-Particle simulations (small timesteps for the fast particles and large ones for the slower ones), this is not (currently) implemented as it would be useless for the other two simulation methods. (See discussion of variable timesteps in section 2.4.2).

3.3 Visualisation techniques

Just a few remarks about the visualisation:

- To show the particle orbits or the new particle positions a simple graphic interface - written by myself using X-windows in combination with Athena widgets - is used. This graphic visualisation process forks from the simulation process and runs as a separate thread that communicates with the simulation program through a Unix pipe.
  This construction ensures the simulation program has little overhead for graphics: it only should write the new coordinates to the pipe and then the bulk of the work is done in the graphic process. This is a pleasant feature for timing purposes as the processtime spent in the graphic process doesn't influence the processtime in the simulation process.
- Animations are achieved by clearing the whole display before every new screen update. Cumulative plots just leave the old positions on the screen as well as the new ones (which is the default).

- Plotting just uses a dot or a small circle to visualise the particle positions: it doesn't connect the previous position with the new one as would be done if we'd draw a straight line between them instead. This makes it possible to get some idea of the speed as well: the larger the distance between 2 positions, the higher the speed of that particle. Drawing a straight line would hide this extra information.
- On the parallel (Parsytec) computer *only* the simulation process is run. The graphic output still uses the same X-windows interface as described earlier, but this time it runs separately on the (Unix) host computer. The communication between them is established by connecting the standard output of the simulation process (through a server process on the host computer called the **iserver**) to the standard input of the visualisation process.
  This is done because the only connection between the Parsytec and the host computer is through the **stdin**, **stdout** and **stderr** channels from the server process, so we *must* use either **stdout** or **stderr** and we cannot open a separate pipe.
- Scaling and shifting the picture can be done through some extra buttons in the visualisation window, so it is not strictly necessary to scale the problem until it fits the graphic window or to adapt the scaling parameters in the source code.

3.4 Restrictions

From all simulation methods discussed in this article, Particle-Particle has the least accuracy restrictions of all. However, things may easily go wrong if particles come to close to each other, so we should be certain that our timestep is small enough to have a reliable simulation also for the shortest possible (or allowable !) distances.

But another much more important restriction rises with Particle-Particle simulation: the acceleration calculation for a system of n particles requires roughly $n \cdot (n-1)$ operations (this could be halved on a sequential system if we assume the force from particle i on j would be minus the force from particle j on i).

This implies for a system of 10 particles about 50 operations per timestep, but for a system of 100000 particles about 5000000000 operations per timestep ! Of course this makes Particle-Particle simulation completely useless for the simulation of large systems.

The two other methods described in this report both scale with $n \cdot \log_2 n$, which is for a system of 10 particles still about 30 operations per timestep, but for a system of 100000 particles now only 1600000 operations per timestep (which is more than 3000 times less as PP) ! Note however that one operation in PM or TC may take up to 10-50 times as much time as one operation under PP.

Chapter 4

Particle-Mesh method

4.1 Introduction

The Particle-Mesh method is one of the oldest improvements over Particle-Particle, introduced somewhere around 1985 by R.W. Hockney & J.W. Eastwood (see [9]). In the PP method we only discretize the time-step, but in the PM method also the spatial coordinates are being discretized by setting up a rectangular grid over the particle space (either 2D or 3D).[1]

After we choose the appropriate grid (which should be large enough to avoid any particles moving outside the grid during the simulation !), the particle masses are assigned to the nearest grid points. This can be done very simplistic such as is the case with "Nearest Grid Point" or in a more advanced way like i.e. "Cloud In Cell" (explained later).

After that we could simply use the PP method to calculate the total force on each particle by summing all contributions of all meshpoints, just as if they were normal particles. However, this would only give us an advantage if the number of meshpoints (M) is much less than the number of particles (N), which could lead to intolerable discretisation errors. It also introduces many practical problems, such as what would happen if a particle is exactly located on a meshpoint (which results in a singularity !). For these reasons this method - although it is the first that comes in mind if you're totally new in these matters :-) - is almost never used.

But there is a much more clever technique that results in a dramatic decrease of the number of computations: As soon as we have assigned the particle masses to the gridpoints, we have in fact created a "density distribution" and according to "Gauss law" there is a relation between the density $\rho(x, y)$ and the divergence of the field (which is $\nabla \cdot E(x, y)$ in case of a Coulomb force and $\nabla \cdot a(x, y)$ in case of a gravitational force) :

[1] Hockney and Eastwood only describe a 1 dimensional model in their book and also the nice article by Peter MacNeice [11] only discusses 1D, so the extension to 2D was mostly derived by myself. Problem with 2D is that the density is an area density, changing the nature of the force. Only 3D simulations are comparable with the standard PP and TC methods.

$$\nabla \cdot E(x,y) = -c.\rho(x,y) \tag{4.1}$$

But, as the gradient of the potential ($\nabla\Phi$) - which is in itself is a scalar field - at a certain location is by definition always equal to the vector field E,

$$E(x,y) = -\nabla\Phi(x,y) \tag{4.2}$$

we can easily derive an important relation between the potential and the density, often called the "Poisson equation" :

$$\nabla^2\Phi(x,y) = c.\rho(x,y) \tag{4.3}$$

The constant c is determined by the nature of the force : it equals $-\frac{1}{\epsilon}$ for the Coulomb force and $4\pi\gamma$ for the gravitational force with 3D problems.

An important problem arises with the definition of the density: in 2D simulations as we use here the density is not mass or charge per unit of *volume*, but mass or charge per unit of *area* ! This fundamentally changes the nature of the force if the Poisson equation is used (see table 4.1). This has to be kept in mind if we want to be able to compare the simulation output for PP and for PM.

	E	ρ	Φ	F	Example in 3D
1D	$\frac{c \cdot Q}{2}$	$\frac{Q}{l}$	$\sim \|r\|$	C	2 charged plates
2D	$\frac{c \cdot Q}{2\pi r}$	$\frac{Q}{A}$	$\sim \log \|r\|$	$\sim \frac{1}{\|r\|}$	2 charged wires
3D	$\frac{c \cdot Q}{4\pi r^2}$	$\frac{Q}{V}$	$\sim \frac{1}{\|r\|}$	$\sim \frac{1}{r^2}$	2 charged particles

Table 4.1: The relation between the dimension and the potential, field and force for an electrical problem.

The Poisson equation from above would be an easy one to solve if we'd know the potential and we wanted to find the density. However, life isn't always that easy: in the case of a known density and an unknown potential we have to solve an integral equation instead.

But fortunately a man called Joseph Fourier invented an amazing mathematical trick to solve this problem : "Fourier Transforms" or "spectral analysis". If the Poisson equation from above is transformed into the so called "frequency domain" using a (spatial) Fourier Transformation, the differential equation changes into an algebraic equation that can be easily solved :

$$(i\vec{K})^2\widetilde{\Phi}(k,l) = c.\widetilde{\rho}(k,l)$$

The Fourier coefficient $\vec{K}$ is just the vector of elements (k,l) , but as it is squared this simply results in its inner product $k^2 + l^2$. The term i^2 only gives us an extra minus sign and so we find for the transformed potential :

$$\widetilde{\Phi}(k,l) = -\frac{c}{k^2+l^2}\widetilde{\rho}(k,l) \tag{4.4}$$

Transforming back to the spatial domain using "Inverse Fourier Transforms" gives us then the potential $\Phi(x,y)$, which can be used to calculate the field $E(x,y)$ in every meshpoint (which is either the electric field or the acceleration).

With this information we can find the "exact" field at the particle location by using some interpolation technique and then we can directly calculate the new position and speed of the particle using one of the integration methods as described in section 2.1.

How the mass-assignment, the Fourier Transforms (that should be discrete ones because of the spatial discretisation) and the field interpolation are done, is something I'll discuss in more detail hereafter.

Roughly the complete algorithm looks as in table 4.1 :

```
- calculate the total energy and momentum
- initialise the mesh (dimensions, memory)
- set the timecounter to 0
- while ( timecounter < endtime )
  - for all particles do
    - compute density ρ by assigning mass to nearby meshpoint(s)
      (using either NGP or CIC)
  - endfor
  - calculate Fourier Transformed density ρ̃ on the mesh
  - calculate (FFTed) potential Φ̃ on the mesh using Poisson's
    equation (∇²Φ = c.ρ)
  - find potential φ using an Inverse Fourier Transform
  - for all particles do
    - compute the field (acceleration) in nearby meshpoints (E⃗ = −∇φ)
    - interpolate the field at the particle position
    - calculate new speed and position using a suitable scheme
  - endfor
  - write or plot the new particle positions
  - increase timecounter with the timestep
- endwhile
- check the energy and momentum for conservation
```

Table 4.2: Flowchart Particle-Mesh code

A word about the boundary conditions for particle problems: in most simulations there is an infinite particle space, that is simulated by using so-called "periodic boundary conditions".

The mesh should be sufficiently large to insure that no particle would move outside the particle space during the intended time of the simulation.

However, particles moving in for example a box with solid walls could be simulated as well by applying different boundary conditions: in this case “Neumann boundary conditions” have to be set, resulting in the speed being always 0 at the boundaries. For more information about boundary conditions the reader is referred to [6] or [16].

4.2 Mass (Charge) to Mesh assignment

To find the density in the gridpoints, we need to use a technique to assign the particle masses to the gridpoints. Two of the most well-known methods will be briefly discussed: a zero-order and a first-order one. (See for a discussion of the 1D version [9] and [11])

To find the density (which is mass per unit of the dimension, i.e. length for 1D, area for 2D and volume for 3D) we have to scale the mass with the size of one cell. It may be a good idea to scale the masses with the size of one cell (2D: cellsize.x * cellsize.y) at initialisation time and thus use a “scaled” mass, to avoid having to do this every time the mass assignment is carried out.

Note that the gridpoints are in my implementation chosen to be in the *centre* of each mesh cell and not on the edges.

4.2.1 Nearest-Grid-Point

This method is very simple and is the same for 1D, 2D or 3D: For each particle just look for the gridpoint that is closest and then assign (add) all the mass of the particle to the mass concentrated in that gridpoint (the cell centre). This is in fact a zero-order interpolation.

4.2.2 Cloud-In-Cell

The most popular method is a technique called “Cloud-In-Cell”. It looks for the 2 nearest gridpoints in a 1D simulation, the 4 nearest ones for a 2D simulation etc. Then it calculates the distances from the particle to these points and assigns each of the 2,4 or 8 nearest gridpoints a weighted part of the total particle mass.

In figure 4.1 my interpretation of the CIC method for 2D is shown graphically : The particle is surrounded by a “cloud” exactly the same size as one individual cell. This cloud overlaps at most 4 cells in the 2D case. The relative size of the overlapping areas for each cell is the scale factor that is used to assign the particle mass to the corresponding cell.

As shown in the right part of the figure, the cell that contains the particle can be used to calculate all necessary information for the other 3 cells as well. This approach saves some calculation time if compared with the left part but otherwise yields exactly the same results.

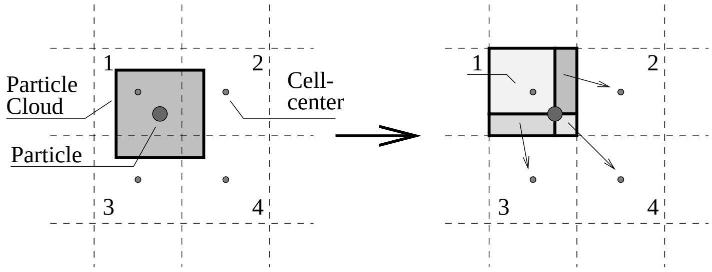


Figure 4.1: Here the 2D CIC mass assignment is shown in a graphic way.

4.3 Calculate Potentials on Mesh

After assignment of the particle masses to the mesh, it is time to convert this density distribution into a potential distribution on the mesh. In the introduction the general method, based on Fourier transforms was already discussed. However, we do not have a density *function*, but only a density *distribution*. As a result, we will only be able to find a potential *distribution*, not a potential *function*. So, instead of differentiating (taking the gradient) we can only use finite differences between the gridpoints.

Equation 4.3 can be written in the form of a finite difference equation (in this case it is a "2nd order centred finite difference equation") :

$$\frac{\Phi(x+1) - 2\Phi(x) + \Phi(x-1)}{(\Delta x)^2} = c \cdot \rho(x) \tag{4.5}$$

or for the 2D case :

$$\frac{\Phi(x+1,y) - 2\Phi(x,y) + \Phi(x-1,y)}{(\Delta x)^2} + \frac{\Phi(x,y+1) - 2\Phi(x,y) + \Phi(x,y-1)}{(\Delta y)^2} = c \cdot \rho(x,y) \tag{4.6}$$

As with the continuous case above, this is a so called implicit problem, that could not be solved in a straightforward manner. One popular technique is Gauss elimination, applied on a tridiagonal (1D) or pentadiagonal (2D) matrix. Another one, which is the one I choose to use here, is through a Discrete Fourier Transform, which is in many ways analog to the discussion for the continuous part in the introduction :

$$\frac{(e^{ik\Delta x} - 2 + e^{-ik\Delta x})\widetilde{\Phi}(k)}{(\Delta x)^2} = c \cdot \widetilde{\rho}(k) \tag{4.7}$$

and again for the 2D situation :

$$\frac{(e^{ik\Delta x} - 2 + e^{-ik\Delta x})\widetilde{\Phi}(k,l)}{(\Delta x)^2} + \frac{(e^{il\Delta y} - 2 + e^{-il\Delta y})\widetilde{\Phi}(k,l)}{(\Delta y)^2} = c \cdot \widetilde{\rho}(k,l) \tag{4.8}$$

Using a little math (Euler's formula) we can rewrite this into :

$$\frac{(2\cos k\Delta x - 2)\widetilde{\Phi}(k,l)}{(\Delta x)^2} + \frac{(2\cos k\Delta x - 2)\widetilde{\Phi}(k,l)}{(\Delta y)^2} = c \cdot \widetilde{\rho}(k,l) \tag{4.9}$$

And this results (using the relation $\cos x - 1 = -2\sin^2(\frac{1}{2}x)$) in :

$$-\left(\frac{(2 \cdot \sin\frac{1}{2}k\Delta x)^2}{(\Delta x)^2} + \frac{(2 \cdot \sin\frac{1}{2}k\Delta y)^2}{(\Delta y)^2}\right) \cdot \widetilde{\Phi}(k,l) = c \cdot \widetilde{\rho}(k,l) \tag{4.10}$$

Now let's define the following intermediate functions K and L to make it all look a little less heavy :

$$K = k\frac{\sin(\frac{1}{2}k\Delta x)}{\frac{1}{2}k\Delta x} = k \cdot sinc(\frac{1}{2}k\Delta x)\ ,\ L = l\frac{\sin(\frac{1}{2}l\Delta y)}{\frac{1}{2}l\Delta y} = l \cdot sinc(\frac{1}{2}l\Delta y)$$

Substitution then yields an equation that closely resembles the continuous Fourier transformed Poisson equation 4.4 we saw before in the introduction :

$$\widetilde{\Phi}(k,l) = -\frac{c}{K^2 + L^2}\widetilde{\rho}(k,l) \tag{4.11}$$

As could be easily seen $K$and L could just be replaced with k and l if the mesh spacing would be infinitely small: since quite some time (:-)) the $\lim_{\Delta x \to 0} \frac{sin(\Delta x)}{\Delta x} = 1$. But unfortunately the essence of finite difference is that the spacing is not infinitely small .....

4.4 Fourier Transforms

Ok, above we have discussed in detail how to obtain the Fourier Transformed potential from the Fourier Transformed density distribution. Now it is time to take a closer look at the Fourier Transform routine for a discrete set of numbers itself. It is crucial that this transformation goes fast enough or else we could as well use the PP method instead and forget about all the nasty stuff above.

First we'll take a look at the standard routine, the Discrete Fourier Transform (DFT), just to be able to understand how it works. Unfortunately for the DFT the number of computations to transform a set of N numbers scales with N^2 and as such it isn't of much practical use. But, I needed a simple algorithm that worked flawlessly, to be able to check the results of the other, superior routine I want to discuss: the Fast Fourier Transform (FFT).

Of course there are many versions for the DFT and FFT available on the net (i.e. NETLIB or NAG) , but most of them are in Fortran and if there is one thing I want to avoid as long as possible it is Fortran (:-). Also there were much less implementations for 2D Fourier Transforms, even in Fortran , and the C versions I saw were so clumsily written (mostly directly transferred from Fortran to C) that I decided to setup my own C library (and distribute it on the net for other people who are interested).

4.4.1 Discrete Fourier Transform (DFT)

The exact so called "Discrete-time (!) Fourier Transform" (do not confuse this with the DFT !!) and it's inverse look as follows:

$$
\begin{aligned}
\tilde{f}(\Omega) &= \sum_{n=-\infty}^{\infty} f(n) \cdot e^{-i\Omega n} \\
f(n) &= \frac{1}{2\pi} \int_0^{2\pi} \tilde{f}(\Omega) \cdot e^{i\Omega n} d\Omega
\end{aligned}
$$

Of course the summation from $-\infty$ to ∞ is impossible to carry out on a computer, where we only have a limited number of samples. We also only want to store a limited number of frequencies in the Fourier domain, which makes the integral also problematic.

For this reason the "Discrete Fourier Transform" was invented, which is devised in such a way that we have *exactly as many frequencies* in the frequency domain *as we have samples* in the spatial (or time) domain. This turns both formules into a summation over a limited number of either N samples or N frequencies. Also in this case we do not store the frequency itself, but the wavenumber k, where the frequency is given by $\Omega_k = k \cdot \frac{2\pi}{N}$. This results in the following set of equations for the DFT :

$$
\tilde{f}(k) = \frac{1}{N} \sum_{n=0}^{N-1} f(n) \cdot e^{-ik\frac{2\pi}{N}n} \tag{4.12}
$$

$$
f(n) = \sum_{n=0}^{N-1} \tilde{f}(k) \cdot e^{ik\frac{2\pi}{N}n} \tag{4.13}
$$

The 2 equations 4.12 and 4.13 are the fundamental rules for using Fourier Transforms on computers. (see for more information about the DFT i.e. [12] , chapter 5)

When using a computer it is often easier to rewrite the complex terms into their separate real and imaginary parts (except if you use Fortran, but as said before : that's out of the question in my case, although I must admit the complex type in Fortran isn't a bad thing at all :-).

If we do so, the first equation (the "analysis", 4.12) expands to :

$$
\tilde{f}(k)_{re} = \frac{1}{N} \sum_{n=0}^{N-1} \{+f(n)_{re} \cdot \cos(k\frac{2\pi}{N}n) + f(n)_{im} \cdot \sin(k\frac{2\pi}{N}n)\} \tag{4.14}
$$

$$
\tilde{f}(k)_{im} = \frac{1}{N} \sum_{n=0}^{N-1} \{-f(n)_{re} \cdot \sin(k\frac{2\pi}{N}n) + f(n)_{im} \cdot \cos(k\frac{2\pi}{N}n)\} \tag{4.15}
$$

And the second equation (the 'synthesis', 4.13) will look like :

$$f(n)_{re} = \sum_{n=0}^{N-1}\{+\tilde{f}(k)_{re}\cdot\cos(k\frac{2\pi}{N}n) - \tilde{f}(k)_{im}\cdot\sin(k\frac{2\pi}{N}n)\} \tag{4.16}$$

$$f(n)_{im} = \sum_{n=0}^{N-1}\{+\tilde{f}(k)_{re}\cdot\sin(k\frac{2\pi}{N}n) + \tilde{f}(k)_{im}\cdot\cos(k\frac{2\pi}{N}n)\} \tag{4.17}$$

As you can see these two routines are almost identical apart from 2 '$-$' signs and a scale factor. For this reason often only one routine is used for both transformations. In that case a small flag is used to put the right signs at the right place.

A few remarks:

- Although the DFT starts at $n = 0$, we should bear in mind that a standard Fourier Transform always has a leftside band and a rightside band with the zero frequency (often called the DC component) in the middle, so in real it should start at $-\frac{N}{2}$ and it should end at $\frac{N}{2} - 1$.
  But as the DFT is periodical, starting at 0 is no problem, as long as we realize that the DC component is at $n = 0$, the *right* side from $n = 0$ to $n = \frac{N}{2}$ and the *left* side from $n = \frac{N}{2}$ to $n = N - 1$. (Note the overlap at $n = \frac{N}{2}$!) An example is given in figure 4.2 .
- The DFT summation starts with $n = 0$ and ends at $n = N - 1$. However, as long as only *real* data is being processed we know that the Fourier Transform only contains *real values* that are *purely even* and *imaginary values* that are *purely odd.* So, in fact it is enough to process only half the data ! The rest could be easily computed by using the following rules :

$$\begin{aligned}\tilde{f}(-k)_{re} &= \tilde{f}(k)_{re}\\ -\tilde{f}(-k)_{im} &= \tilde{f}(k)_{im}\end{aligned}$$

  Where the righthand side is the right band and the lefthand side is the left band of the Fourier transform.

4.4.2 Fast Fourier Transform (FFT)

The Fast Fourier Transform, discovered somewhere about 1965, is an algorithm that computes exactly the same result as the DFT described in the previous part, it only does this tremendously faster, especially for high N, as the number of computations for the DFT scales with $O(N^2)$, while the FFT scales with $O(N \cdot \log N)$.

I am not going to discuss the FFT here, but only mention a few things to bear in mind when using the rather simple library I build for it.

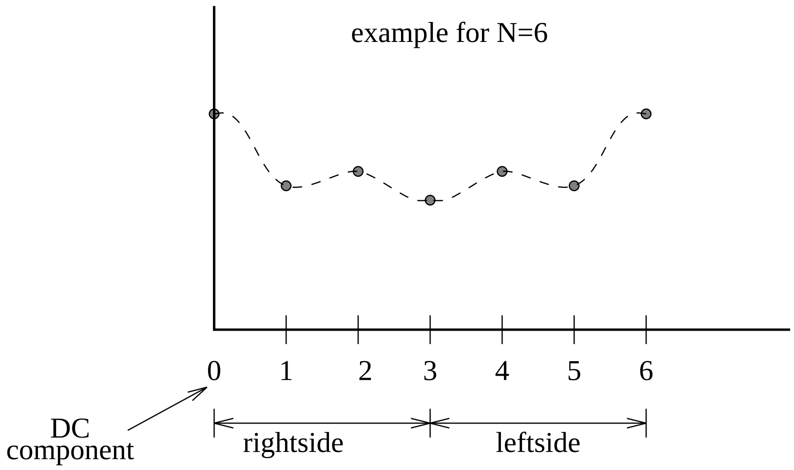


Figure 4.2: DC part, right- and leftband in the asymmetrical DFT (n=0 ... N-1)

A fine (and highly readable !) description of the FFT is given in [13], although the provided code is somewhat clumsy as it was directly derived from Fortran. Nevertheless, my own code was loosely based on these examples as they are very simple and straightforward.

A few remarks :

- FFT's split the input data stream into odd and even parts, that are again splitted into odd and even parts and so forth, until finally we have subvectors small enough to process with normal DFT's. In my code I just continue splitting up until having subvectors with the rather trivial length of 1. This is not the most efficient, but certainly the most simple method and the other methods only give an improvement of 20-30 %.
- Because of the splitting up in odd and even parts, FFT's can only work efficiently with N being a power of 2. If N is not a power of 2, a much more complicated algorithm is needed, unless we pad it up to the next power of 2. To keep it simple, the library just checks if N is a power of 2 and refuses to continue if this is not the case.

4.4.3 2D Fourier Transforms

Everything discussed so far only concerns the 1D Fourier Transform. However, we have to transform over 2 dimensions in our case. This is done by first transforming all the rows separately and after that all the columns. So, for a $N \cdot M$ mesh we have to do N transforms of vectors with length M and M transforms of vectors with length N. This results in a total number of FFT's (and also inverse FFT's) of $N + M$ (for each timestep !).

A practical problem is the memory layout in a computer: this is still only one dimensional, so all rows and columns of the mesh lay behind each other. This implies that we can *either* pass the address of a row to the FFT - assumed that all the row elements are in one contiguous memory area -*or* the address of a column - in case all column elements are in one contiguous area - but certainly not both.

So if we want to use the same FFT routine for both the rows and the columns, we have to swap all elements in the rows and columns halfway the transformation !

As said in the previous section, both N and M should be a power of 2, although it is not necessary to give them both the same value, although this causes extra problems in transposing the matrix halfway (see algorithm) and it requires 2 different cosine lookup tables. Note that the order is unimportant, as long as we do not execute both at the same time ....

The 2D algorithm roughly looks as in table 4.4.3.

```
- if it's the first call:  generate a sine/cosine lookup table for
  the given dimension to speed up calculations
- else if given dimension is different from lookuptable size:
  generate an error
- set all elements in target matrix to zero
- for (row=0; row<dim; row++)
  - fft1d(source[row],target[row],dim,signflag)
- endfor
- transpose the target matrix, now filled with transformed rows
- for(col=0; col<dim; col++)
  - fft1d(target[col],source[col],dim,signflag)
- endfor
- ready
```

Table 4.3: How to FFT a 2-dimensional mesh using only one 1-dimensional FFT routine

4.5 Calculate the gravity (or electric) field

After that we've found the potential on the mesh, we actually have had the worst part (sigh). Now comes the calculation of the gravity field (which is simply the acceleration !) or the electrical field on the mesh and thereafter the interpolation of the field for all particle locations. Using formula 4.2 we can relatively easy find the field if we know the potential. The known and unknown part are interchanged if compared with the Poisson equation, which leads to a simple, straightforward problem.

But - as always - there are a few quirks to be counted with. In the discretised version we (or at any rate I) would tend to write equation 4.2 as follows :

$$E_x = \frac{-(\Phi_{x+1} - \Phi_x)}{\Delta x}$$

But consider what would happen if a particle is located at *exactly* meshpoint x ? In that case we would find a field in meshpoint x that virtually only contains the contribution of that particular particle. So the field at the particle location (which is also x) will be exacly that same field !

In other words : the particle only "feels" it's own influence, and a so-called "selfacting force" works on the particle, which of course leads to nonsense results. (Consider for example the earth: it will never start moving only because of it's own gravity field !!) But, selfacting forces can be easily avoided by applying "central differencing" instead :

$$E_x = \frac{-(\Phi_{x+1} - \Phi_{x-1})}{2\Delta x} \tag{4.18}$$

In that case the contribution of the particle in meshpoint x will be exactly cancelled out, as its influence on Φ_{x+1} is precisely the same as on Φ_{x-1}.

4.5.1 Field interpolation

Now that we know the field in the meshpoints, we can interpolate between the meshpoints to find the field at the particle location. This is in fact a similar (but reversed) problem as the mass-assignment to the mesh. For reasons I won't discuss here in detail, it is important to use *exactly the same interpolation method as in the mass-assignment*, to avoid loss of momentum and the introduction of large numerical errors. Main problem is the introduction of "selfacting forces" in the simulation if this rule is neglected.

So, if we did the mass-assignment using NGP, we should also use an (inverse) NGP for the field interpolation and if we used CIC we should use an inverted CIC to find the field. This ensures, together with the central differencing from above, that the force of particle A on B will be exactly the opposite of the force of B on A.

For further information the nice article by Peter MacNeice ([11]) and of course the "Holy Bible" for Particle Mesh methods by Eastwood and Hockney ([9]) are strongly recommended.

4.6 Implementation and practical aspects

While building the program a lot of little and bigger practical problems had to be solved. I just want to mention a few points hereafter :

- For practical reasons the dimension in both the X and Y direction should be the same, or else we would have problems in transposing the 2D matrix. This also avoids the use of two sets of lookup tables in the DFT.
- A lookup table for cosine and sine values in the Fourier Transform routines can dramatically speedup the calculation time. In my program a vector with `complex` type is

used: the cosine value goes in the real part and the sine value in the imaginary part.

- The FFT'ed discrete Poisson equation (4.10) needs another lookup table for the *sinc* values to avoid numerous useless floating points operations. Because of the symmetry found in the FFT with real inputs, a vector with only half the length of the dimension was sufficient.

- The value of the DC component in the FFT is of no use : not the *absolute* potential but the *relative* potential if compared with an arbitrary chosen reference point is important. For that reason the DC component is set to be zero, to avoid it had impractical (i.e. very large) values.

- The mesh dimensions should be sufficiently large to ensure that no particles would leave the mesh during the whole simulation. If one particle leaves the mesh within a simulation, the program will terminate, as continuing could lead to nonsense results. Currently the dimensions are set to 2 times the largest distance that was found between 2 particles (for both X and Y), unless the dimensions were explicitly specified on the commandline.

- The number of cells in one direction is automatically set to be of the same order as the average minimum interparticle distance (which should be more or less equal to the smallest minimum interparticle distance). It currently has a minimum of 32x32 and a maximum of 512x512. This value can also be overridden on the commandline.

- Large fluctuations in the density distribution results in high frequencies in the Fourier transform. If the mesh is not large enough, this could lead to these high frequencies erroneously increasing the *lower* frequencies: **the aliasing effect**. There should be a function in the program to detect this effect and issue a warning if it gets to large.

- During one iteration a particle may never move more than the magnitude of the particle cell. So, if the cellsize is halved, also the timestep should be halved (provided the initial choice was correct !) , to avoid problems here. As my programs for PP, PM and TC share one set of integration routines, this isn't yet detected, but it should be added to make the program more reliable. (Probably next year ...)

- If a collision occurs in a Particle-Mesh simulation it can for obvious reasons never be properly detected, as the interparticle distance is then much smaller than the cellsize. For this reason, collisions are not being reported as PM shouldn't be used on problems where many collisions are likely to occur.

4.6.1 Tunable parameters

As with PP, there are many variables that can be used to alter the specific behaviour of the simulation. In the datafile the stepsize, the number of iterations and the number of iterations between a screen update can be specified. For the exact meaning of the datafile layout, see the appropriate appendix in A.

Commandline options :

-i,a,x,f,o	Same function as with PP (integration method, animation, no graphics, no energy error computation and output the coordinates to a file).
-w <nr>	Width of the mesh, expressed in the same units as the particle positions. Overrides any default value.
-h <nr>	Height of the mesh, expressed in the same units as the particle positions. Overrides any default value.
-n <nr>	Number of mesh cells in one direction (This is used for both the X and Y direction). Specification overrides the automatic computation.
-d <nr>	Mass distribution and Field interpolation algorithm: 1=NGP and 2=CIC.
-m <file>	Write the initial mesh to a file and exit. Only useful if combined with -o option: The two files can be used to build a picture of the initial situation at the start of the simulation. (i.e. to use it in this report :-)

At compile time there are several other parameters that can be set, but these are the same ones as for PP and are not discussed here. Note that the COLLISIONDISTANCE and COLLISIONTYPE would be useless here !

4.7 Visualisation techniques and problems

The same graphical interface as used for PP is also used for the PM program, so the remarks in the PP chapter also apply here. There is however one thing I'd like to add here:

To check the density and potential distribution I used a circle with a radius proportional to the meshvalue, to give some graphical indication about these distributions. This of course made development significantly easier. Because of the large differences between small and large values I decided upon a logarithmic relation between radius and meshvalue. But, unfortunately some graphic libraries (at any rate the one that was developed by me) , only use the *magnitude* of the supplied radius to draw a circle !

That is no problem in most cases, but in the case of a logarithm this could be disastrous: an ascending series like 10^{-2}, 10^{-1}, 1, 10, 10^2 would be represented on the screen by circles with radius 2, 1, 0, 1, 2 : a symmetric one, so 10^{-2} would be equal to 10^2 !

I did spend 2 full days debugging to find out why my DFT and FFT wouldn't work until finally I found they worked perfectly well : only the visualisation failed because of the problem above. If this warning could save only ONE person from making the same mistake it is already worth the trouble mentioning it here, isn't it ?

4.8 Restrictions

From all 3 methods described in this report, the Particle-Mesh method is the fastest, provided we're talking about a large number of particles. However it has some restrictions that are possibly already clear to the reader, but I like to mention them hereafter for completeness :

- PM only makes sense for simulations with N larger than about 100 particles. For small numbers it is much slower than PP.
- The 2D approach from above can not simulate 3D problems, even if all particles move in one and the same plane. The only way to simulate real 3D problems is to use a 3D mesh instead of a 2D one. This introduces a density function that uses mass per unit of volume and makes it necessary to use 3D Fourier transforms to find the potential on the mesh, resulting in a sharp increase of the number of computations. However, the technique is practically the same.
- Because variations on a smaller scale than the meshspacing are washed out by the algorithm, PM only works for "collisionless" systems, with a more or less uniform distribution.
- The force contribution of close neighbours should be small compared with the total contribution of far away particles. The greater the distance in relation to Δx (and Δy), the more accurate the field approximation on the grid will be.
- The output is almost never reliable enough to predict the exact path of *one* particle: this can only be a rough indication. On the other hand it can show tendencies (like i.e. clustering) quite accurate.

Chapter 5

Tree-Code method

5.1 Introduction

In the previous chapter we discussed the Particle-Mesh method. This method is relatively simple with respect to software engineering, but mathematically a little more complicated (at least in my perspective !).

With the Tree-Code method we have in fact quite the opposite problem : from a mathematical point of view this is quite a straightforward method, so there won't be many nasty formulas in this part, but the necessary dynamic memory management and recursion makes software development not completely trivial.

Tree-code methods are a completely different way to achieve the same result as with Particle-Mesh : downscale the computation time for a simulation to be of order $N \cdot \log N$ instead of N^2.

The basic idea is to replace a faraway group of particles by only *one* virtual particle, exactly located in the centre of mass of this group and with a mass that equals the reduced mass of this group of particles.

This virtual particle is a particle that obviously attracts with the same force, no matter in which direction we are looking. Such a center of force is often called a **monopole.** The first implementation was published about 11 years ago in Nature in an article by Joseph Barnes and Piet Hut (see [4]).

Under some circumstances (for example with a group of particles that is very long in one direction but short in the other direction) the approximation with only one particle could not be good enough. In this case a so called **multipole** would be used, which sounds heavy but in fact is nothing else than using 2, 3, 4 or even more virtual particles instead of only one. (for the example in this paragraph a dipole would be a good approximation).

The extension from Monopole to Multipole is known as "**Fast Multipole Method**" and it's basics were first published by L. Greengard and V. Rokhlin about 10 years ago(see [7]).

Main difference is that the FMM method doesn't store the mass but the potential for a certain cell and as such it may be viewed as a sort of "cross-breading" between PM and pure Barnes-Hut (BH). Because of this potential it is possible to use FFT techniques just as with PM and this can make the FMM even faster than BH. Most people state that the FMM can scale with order N.

As the BH algorithm is relatively simple to understand and because FMM may be viewed as just one of the many possible optimisations of BH, I choose to forget about FMM for the time being and just investigate and discuss BH. If you understand Barnes-Hut and saw Particle-Mesh it is not so difficult anymore to get the essence of FMM, while understanding FMM from scratch is a very hard thing to do.

So let's see in a little more detail how Barnes-Hut works and what kinds of practical problems I ran into when trying to implement it using the article from Nature ([4]) as a starting point.

5.2 Algorithm based on Barnes-Hut method

1. As said before : the essence of the Barnes-Hut algorithm is to discriminate between faraway and nearby particles and to do the timeconsuming Particle-Particle computation only for nearby particles. To achieve that we first choose an initial areasize, large enough too ensure no particle will leave the area during the whole simulation. This initial area is our "rootcell".

2. Now we start splitting up this rootcell into 2^{dim} daughtercells (dim : dimension: either 1, 2 or 3D, resulting in either a binary-, a quad- or an octtree) until we have captured every particle into a separate cell. Obviously for dense areas it will take a lot of splitting up until everything is separated, while in sparse areas there almost won't be any splitting at all : the tree adapts itself to the local complexity !
   Sometimes this is called an adaptive grid, although it is important to realise that here we do *not* use a spatial discretisation, so the term grid is a little misleading in this case. [1] A tree example for the 4-body problem from chapter 1 is shown in figure 5.1.
   Im my implementation I had to count the number of particles in each daughtercell after every split to find out if it was necessary to split that cell any further. Initially this took a lot of the total computation time, but after some experiments with sorting techniques I was able to decrease the time spent for counting considerably (see 5.4.1).
   Later I found out that most of the "guru's" use another technique : insertion of the particles in the tree and split up the cell as soon as it already contained at least one other particle (in fact I found no one who used my sorting method ....), but as my implementation worked reasonably well after adding the sorting optimisations I decided to be stubborn and to continue using my own method, following my favourite maxim

[1] There are also methods that use a mesh (which implies spatial discretization) and that also use a local grid refinement. Only these methods are true adaptive grid methods.

```
- calculate the total energy and momentum
- initialise the tree (rootcell dimensions, initial memory)
- while ( timecounter < endtime )
  - sort all particles on either the X or Y coordinates
  - recursively split the roottree into 4 daughter cells until every
    particle fits into a separate cell (top → bottom)
  - recursively compute the total mass and centre of mass for
    all cells in the tree (bottom → top)
  - count all particles in the rootcell (using the sortlist) to be
    sure no particle moved outside the tree
  - for all particles do
    - recursively compute force contribution of all cells starting
      at the rootcell using the following rules :
    - count the number of particles (n) in the current cell
    - if n = 0 do nothing (empty cell)
    - if n = 1 compute force contribution of this single particle
      (unless it is the currently processed particle !)
    - if n > 1
      - compute the distance r to the centre of mass and the cellarea A
      - if (r/A > θ ):  cell is faraway so compute force contribution
        using total mass and centre of mass and go to next particle
      - else:  recursively process the subcells down the tree until
        1 particle in cell or far enough away
    - endif
    - end recursion
    - calculate new speed and position using a suitable scheme
  - endfor
  - write or plot the new particle positions
  - increase timecounter with the timestep
- endwhile
- check the energy and momentum for conservation
```

Table 5.1: Flowchart Tree-Code for a 2D system

that : “it is better to have your *own* algorithm than a *good* algorithm !”

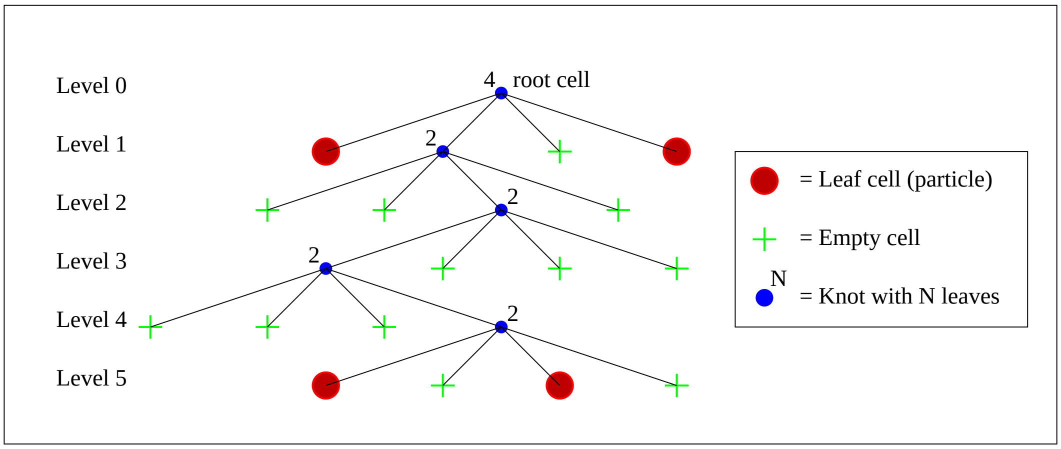
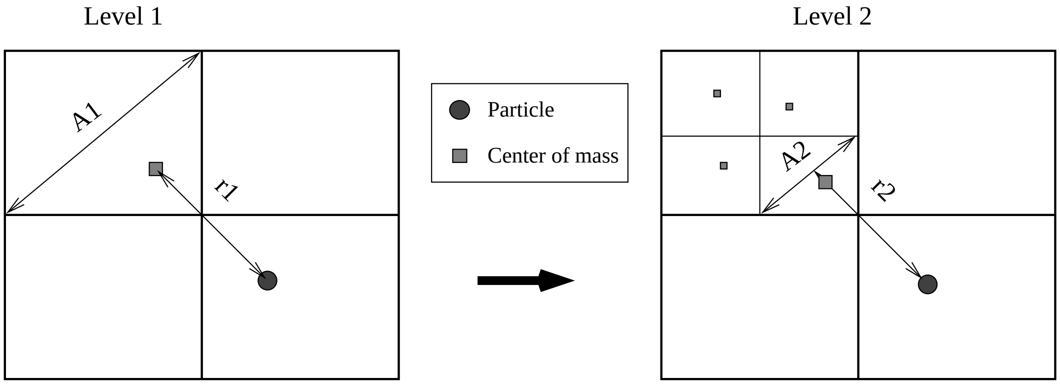


Figure 5.1: Tree example for the 4 body problem from chapter 1.3.

3. Note that building the tree was started at the top and ended at the bottom. Now it is time to compute for each cell the **reduced mass** and the **centre of mass**, and this is much better done starting at the bottom and ending at the top: doing that we could use the information of the lower level in the tree to compute the information for the level just above that.
   This trick ensures that we only have to compute the contribution of at most 2^{dim} (virtual) particles as that is the maximum number of daughtercells each cell can have : in for example a 2D problem the centre of mass of a parent cell can be computed by *only* taking the (centre of) mass of it's 4 daughtercells, without knowing anything about the level below the daughters (or sons if you like) !
   At the lowest level of the tree (in the so-called leaf cells) things are even more simple: there we only have to use the mass and position of the single particle in it and thus it is not necessary to compute or even assign **reduced mass** and the **centre of mass** in this case: just a reference to the particle information will be sufficient.

4. Now the tree is completely finished. But as we have to do the steps above for each iteration it may be a good idea to count all particles in the root cell, to be sure we really have all particles included. This is because it is always possible that a particle moves outside the rootcell dimensions, which could have nasty results for the simulation !

5. If the tree is ready we can turn to the most important part : determine how large the total force is that works on *each* particle. This is again done in a top-bottom approach: for every particle we start at the rootcell and then go down the treebranch until we have reached the lowest point. Every cell we pass contains either 0, 1 or more than 1 particles, each of which asks for a different treatment :



$\frac{r1}{A1} < \Theta$: too close $\Rightarrow$go one level deeper $\qquad$ $\frac{r2}{A2} > \Theta$: this is fine

Figure 5.2: Discrimination between faraway and nearby cells using $\Theta = 1$ (note that $r2 \approx r1$)

(a) $n = 0$: The cell didn't contain any particles and can just be ignored.

(b) $n = 1$: The cell contains one particle : in that case we can just calculate the exact contribution of that single particle to the total force on the particle currently being examined, just as in PP.

(c) $n > 1$: The cell contains more than one particle. Now comes the most difficult part, but also the part that can make TC so much faster than PP : we have to find out if this cell is far enough away to approximate the force by the reduced mass and centre of mass of this cell: the cell is being used as a virtual particle.
This was very cleverly devised by Barnes and Hut : they introduced a special parameter Θ, that should be compared with $\Theta_p = \frac{r}{A}$, where r is the distance from the *particle* being examined to the centre of mass of the *cell* currently being examined (**not** the cell-centre !) and A is a measure for the size of the cell at the current level. The exact value for Θ is the discriminating factor that decides who is far enough away and who is too close.
If the cells are all square or cubic as in BH, then A seems to be equal to the size of only *one* edge, although unfortunately the article by Barnes-Hut isn't very clear at this point. But because I used tree cells that could have *different* dimensions in each direction (which can be dangerous, but also gives the user some extra freedom in case of special problems) I decided to replace this to use the diagonal $A = \sqrt{w^2 + h^2}$ instead.
This yields us $\Theta_p^2 = \frac{r^2}{w^2+h^2}$, [2] where w and h are the cell dimensions.
Note that this results in my value for Θ_p for a square (or cubic) cell being $\sqrt{2}$ (or $\sqrt{dim}$ to be accurate) smaller than in the case of BH.
There are obviously 2 possibilities :

$$\Theta_p > \Theta \tag{5.1}$$
$$\Theta_p \leq \Theta \tag{5.2}$$

[2] I left the squares in the parameter only to avoid the extra computation time needed for taking the square root again.

   i. The first case (5.1) simply leads to the conclusion that the cell is far enough away to use it as a virtual particle without looking any further at its daughtercells.
   ii. The second case (5.2) shows that the cell is too close and that we should go down one level and repeat everything starting with point 5a for *each* of the 2,4 or 8 daughtercells. How exactly the factor Θ_p changes as we go to a lower level and thus how Θ (which is constant during the simulation !) can separate faraway and nearby cells is shown graphically in figure 5.2.

6. If we have calculated the total force on all particles, we can use again one of the integration schemes already discussed in chapter 2.1 and calculate new positions and speed.

7. And then we can increase the timestep and start again at 2 where we have to repeat everything again, including building the tree and computing the reduced masses and centre of mass.

The whole algorithm is described in the flowchart in figure 5.2.

5.3 Choice of the Θ parameter

As should be clear from above, the choice of the parameter Θ is very important. The larger it will be, the further away the cell has to be, before we can use the virtual particle approach and thus the more the algorithm will behave like PP: accurate but slow. For $\Theta \to \infty$ TC simply turns into a very complicated way to do plain old PP.

But how small can we make Θ before running into real problems ? That is not really difficult : we should avoid under all circumstances that we use a cell from the same branch as the particle we want to know the total force on, as in that case the mass of the examined particle itself can contribute to it's own acceleration ! (usually called self-acting forces) So, basically *all* cells from the branch where the particle belongs to are forbidden, *no matter how far away or how large they are,* except of course for the lowest level, where only real other particles are found.

In theory, with the examined particle having a mass of almost 0 and with very heavy other particles at a strategic point in the same cell, there is small chance that the centre of mass for the cell lies in the upperleft corner, while the examined particle is in the lowerright corner. In that case the distance will be $r \approx \sqrt{w^2 + h^2} = A$ and thus Θ_p will be approximately 1. (although still slightly smaller than 1 !) So, as long we keep $\Theta > 1$, we can be sure to avoid the self-acting forces under all circumstances, at least with my definition for Θ_p .

Note that in the case of the definition for Θ_p as given by Barnes and Hut Θ should be bigger than $\sqrt{2}$ to be on the safe side.

One more remark: unfortunately I choose the definition of Θ_p to be just the opposite as in the Barnes-Hut article (stating $\Theta_p = \frac{A}{r}$), so where BH says $\Theta_p < \Theta$ I say $\Theta_p > \Theta$ and vice-versa.

Sorry for the extra confusion this may cause, but I only found out this when I was almost finished with the report and changing the code at this point doesn't seem a good idea to me.

Resuming :

- Values for $\Theta < 1$ are strictly forbidden as they can cause self-acting forces (but this happens only in rare cases).
- Large values for Θ yield a more accurate simulation, but also causes a longer computation time.

I choose the Θ to be default 1. To give an idea about the effect of variations in Θ, I ran my TC program on a random set of 500 bodies , using 50 iterations and leapfrog (Linux on a Pentium 75) :

Discriminating factor Θ^2	1	5	10
Energy leak over 50 iterations (%)	-0.336531	-0.336423	-0.336426
Runtime for 50 iterations (sec)	15.51	34.92	49.24
Momentum leak X direction (%)	-0.77	-0.03	0.001
Momentum leak Y direction (%)	-0.248	-0.008	0.005

As was to be expected the runtime increases rapidly with increasing Θ but as a compensation the accuracy improves at the same time.

In normal situations a value of 1 gives acceptable results with reasonable speed. Recently it was however shown by Salmon and Warren ([14]) that it is possible for the error to become unbounded in some cases, so it is always wise to pay a little attention to this point. They also give some other possibilities for the definition of Θ that can be interesting, but that are not further discussed here.

5.4 Implementation aspects

5.4.1 Sorting techniques

As said before: my implementation for TC is slightly different from most others as I found out later. This is caused by the fact that by reading the article you only get the basic idea of an algorithm, so the practical problems is something you have to solve by yourself and sometimes you (or in this case I !) simply do not make the best choice. Let's view this difference a little closer :

- Barnes-Hut just loops through all particles and inserts each particle in the daughtercell that is currently on the lowest point in the appropriate branch, splitting up that daughtercell as soon as it contained more than 1 particle after the insertion. This has

the advantage that it is not necessary to count how many particles there are in a cell, as when the tree is finished and a counter is updated every time a new particle is inserted the total number would be automatically correct.
There is however some extra work to be done as we need to determine in what daughter-cell (that should be always at the bottom of a branch) the particle should be inserted. This makes it necessary to walk through one complete branch of the tree for each particle.

- My Tree-Code version starts at the root cell, counts all particles in the cell and splits recursively into a number of daughtercells, where again all particles in each daughtercell have to be counted. This counting involves comparing the coordinates of each particle with the current daughtercell dimensions.
  This counting determines whether or not we have to split up the cell any further and as such is a crucial step in the program.

How important counting really was I only discovered after some analysis with the fantastic profiler option provided by the GNU-C compiler (-p) : I found that most of the total computation time was spent on comparing the coordinates if I ran a sufficiently large simulation ! As this made my program incredibly slow I started thinking of something to speed things up and decided to use a sorting algorithm for these coordinates.

I choose to sort only on either X or the Y-coordinate, as this could already reduce the number of particles to be examined considerably. If you know that the particles are sorted on one of their coordinates (how this could be done will be discussed later), the following procedure could be used to compare the coordinates with the cell dimensions (let's assume the particles are sorted on the X-coordinate) :

1. Search the *first* particle in the sortlist that has an **X-coordinate** larger or equal than the X coordinate of the left edge of the cell ($x_n \geq xl$).
2. From here you can start comparing the **Y-coordinate** of each particle and check if it is larger or equal than the lower edge and smaller than the upper edge and increase the particle counter for each particle that fits this condition ($y_n \geq yb \wedge y_n < yt \wedge x_n < xr$).
3. You can repeat the previous step until you reached a particle with **X-coordinate** larger or equal than the other edge of the cell ($x_n \geq xr$).
4. As soon you find such a particle you can be sure that all remaining particles in the sortlist are outside the cell and so you could stop counting the particles for this cell.

Note that the differences between "larger than" and "larger or equal" are very important, as this is the only way to avoid particles being included in 2 cells if they happen to be exactly on the border between them.

In an average distribution you can assume that the number of particles to be compared in step 2 is about the square root of the total number so this method indeed can be a time saver.

But a few problems are still left to be solved :

- Searching for the first valid particle. Just starting at number 1 and then count sequentially until the valid one is found could be very time consuming and the performance improvement wouldn't be very impressive (about a factor 2). This is why I later changed this part into something that resembles the "**Quicksort**" algorithm : Here we start half way the list and if $x < xl$ you take the next half in the left direction, else you take the next half in the right direction and so forth until you found the right starting point. The principle is shown in figure 5.3.

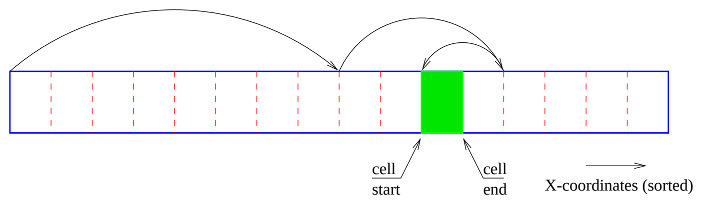


Figure 5.3: Fast search of the starting point in the sortlist.

  The computation time for this method scales with $O(\log N)$ instead of $O(N)$ and thus immediately increased the performance dramatically !

- The particle list has to be resorted after every iteration. Using an "**Insertionsort**" algorithm would scale the time for this part with $O(N^2)$ in the worst case. Using a "**Quicksort**" would in most cases only take $O(N \cdot \log N)$ operations, so it seemed more appropriate.
  But be careful, don't be mislead (as I was initially !) : sorting the first time would be difficult, as then nothing could be said about the order of the coordinates (unless we would use presorted parameterfiles !), but after the first iteration we can be *sure* the particles only moved a little compared with the previous step, provided the timestep was properly chosen. This implies that the sortlist will still be more or less accurate and only needs minor adjustment, and for minor adjustments **Insertionsort** only scales with $O(N)$ while **Quicksort** is known to be even slower on sorted lists than on unsorted lists ($O(N^2)$) !!
  This leads to the conclusion that the simple **Insertionsort** is the best algorithm for building the sortlist, provided of course the timestep is small enough to ensure the sortlist will still be more or less correct after one iteration.

- The Y-coordinate isn't sorted, so here we still have to do the time consuming comparison for each particle. For the problem sizes I used this proved not to be a problem, but for problems of several magnitudes larger than I used ($> 10^5 ... 10^6$) it may be a good idea to sort the Y-coordinate as well and use the same tricks to jump to the correct Y-coordinate starting point as described for the X-coordinate. I didn't implement this, but if necessary it should be a relatively trivial task: "cut and paste" would do most of the work :-)

Note however that sorting only one coordinate reduces the number of computations for the tree generation to be about $O(N\sqrt{N})$, while sorting both coordinates reduces the order for this part to only $O(N)$.

I think the difference between Barnes-Hut and my method - with the improvements described above - wouldn't be very significant, but I have to admit I didn't check this yet and so it remains to be done. So there is still a possibility that my method is faster .....

5.4.2 Memory considerations

Empty cells As could be seen in picture 5.1, highly non-uniform problems could result in a considerable amount of empty daughtercells (leaves). In some descriptions of Tree-Code these empty cells are removed from the list to avoid them wasting memory.
I made removing empty cells a compile-time option and to my surprise I found that removing the empty cells makes the algorithm in some cases faster instead of slower !
This could be explained by the fact that - although removing empty cells takes more work in building the tree - the possibility to skip them immediately in the Centre-of-Mass computation and in the computation of the acceleration can save quite some time. The latter only happens of course if there are enough empty cells to be removed from the list or else we'd just do extra work for nothing !

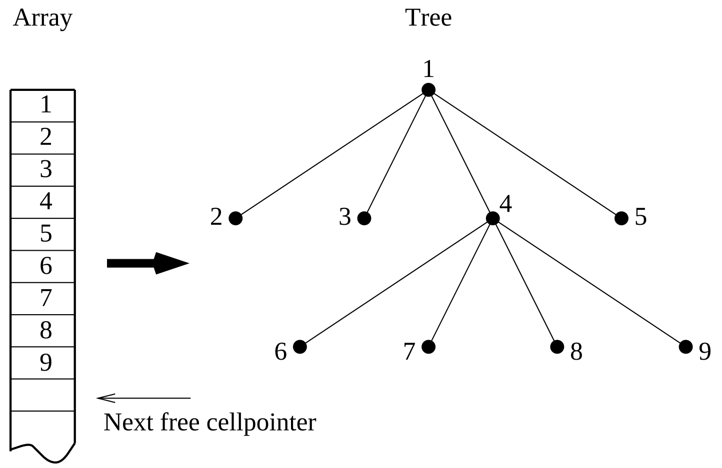


Figure 5.4: Cell array to cell tree mapping.

Dynamic tree allocation If we would do a `malloc()` for each new cell and do a `free()` for the whole tree after every iteration, most of the computation time would be spent in memory management as I discovered. On the other hand : using an array of static dimensions would lead to problems as it is difficult to predict how many cells would be necessary to build a tree for a certain problem (see figure 5.1 for a bad example !), so this is also not a satisfactory option.

I decided to use a "best of both worlds" approach: I **malloc** an initial array of cells of length N and as soon as I run out of cells at a certain point in the program, I do a **realloc** to increase the array with again N elements. This ensures the array will allways be large enough to supply all the necessary cells for a certain tree.
A decreasing/increasing "nextfree cell" counter is used to decide which cell is going to be the next one to be used to build the tree.
Note that storing pointers for the daughtercells in the tree is **not** a good idea : after each **realloc** the whole array may be (but not has to be !) shifted to another memory location, making the pointers useless. The only good way is storing the relative offset to the start of the array (the rootcell). This took me several hours to find out, so a warning at this point doesn't seem to me a complete waste of paper !
The relation between memory and tree is shown in figure 5.4.

Cellsize The dimensions of each cell can either be stored in the cell record or be computed every time it is needed by using it's level in the tree. The first option uses more memory, the second takes more computation time. I have put the choice in a compile time parameter, so both can be used.

5.4.3 Tunable parameters

As with PP and PM, there are many variables that can be used to alter the specific behaviour of the simulation. In the datafile the stepsize, the number of iterations and the number of iterations between a screen update can be specified. For the exact meaning of the datafile layout, see the appropriate appendix in A.

Commandline options :

-i,a,x,f,o Same function as with PP (integration method, animation, no graphics, no energy error computation and output coordinates to a file).

-w <nr> Width of the root cell, expressed in the same units as the particle positions.

-h <nr> Height of the root cell, expressed in the same units as the particle positions.

-t <nr> Value for the discriminator Θ^2 (default equals to 1).

-p Plots the new tree after every screen update. Gives a nice idea how the tree is built , especially if combined with the animate (-a) option.

-u <nr> It is not absolutely necessary to update the tree after every iteration. With this parameter the number of iterations between 2 tree updates can be changed. It is set to 1 by default. To give an idea of the influence of this parameter I ran it twice with u=1,3 and 10 on the now wellknown set with 4 particles and a random set of 500 particles (Linux on a Pentium 75) :

4 particles:	u=1	u=3	u=10
Runtime (7580 iterations)	3.02	2.16	1.66
Energy leak (%)	0.061	0.063	0.067
Momentum leak (X) (kg.m/s)	-10.96	-224	-957
Momentum leak (Y) (kg.m/s)	-15	-57	-124

500 particles:	u=1	u=3	u=10
Runtime (50 iterations) (%)	15.35	14.25	13.48
Energy leak (%)	-0.3365	-0.3364	0.06145
Momentum leak (X) (kg.m/s)	42.06	47.67	-10.96
Momentum leak (Y) (kg.m/s)	-12.2	-12.6	-15.41

-d <file> Write the initial tree to a file and exit. Only useful if combined with -o option: The two files can be used to build a picture of the initial situation at the start of the simulation. (i.e. to use it in this report :-)

At compile time there are several other parameters that can be set :

NSUB This sets the number of daughtercells for each parentcell. A value of 2 yields a binary tree, 4 a quadtree and 8 an octtree.

BLOCKSIZE This is the initial size of the tree array. Every time the system runs out of cells, the array will be increased with BLOCKSIZE new cells. If it is set very low, the memory allocations may take up a significant time at the first iteration. However, as the array is only enlarged and never shrinked, this will only affect the first iteration. Note that if two particles come very close the array size may increase very fast: it may take quite some splitting before the two are separated.

NO_EMPTYLEAVES This removes all daughtercells that contain no particles. Removing them takes more time, but then these leaves can be skipped in the centre of mass computation and the force evaluation. This is the default as in most cases it is the fastest method.

RECALC_CELLSIZE The exact cell dimensions can be directly computed from the size of the rootcell and the level the cell belongs to: in 2D a cell at level 2 has exactly $\frac{1}{4}$ the size of the cell at level 1. Computing the cellsize each time it is needed instead of storing it for each cell saves memory, but increases the computation time.

5.5 Collision management

Of course the collision management from PP works fine here, but there are 2 remarks to be made :

1. In theory it is possible that a collision takes place between a real particle and a cell **not** being a leave (a virtual particle). Although this can only happen if Θ is set to a much too small value, I only report the fact that a collision has occurred, but not between which particles to avoid confusion.

2. Non-elastic collisions by maintaining the 2 collided particles at the same place may cause large troubles in building the tree : it is impossible now to separate the 2 particles into different tree-cells, no matter how often we subdivide the tree !

5.6 Visualisation techniques

The same graphical interface as used for PP is also used for the TC program, so the remarks in the PP chapter also apply here. The only addition is that also the tree itself is plotted and even can be animated if necessary, which yields quite a spectacular picture, at least in my opinion !

(use the command : `treecode -a -p <parfile>` to get this animated tree)

5.7 Restrictions

For PM the restrictions were fairly straightforward : only uniform distributions with a large number of particles. For PP things were comparably straightforward : use it only for a limited number of particles or where accuracy is of major importance.

For TC it is slightly more complicated : TC produces most of the time very accurate solutions in a reasonable time. But unfortunately this is not always the case, as shown in [14]. It only is a bit difficult to find out if the worst case situation as described in their article may arise while investigating a certain problem. Methods like I used in my analyser (auto-correlation) (see 6) are not able to say anything useful about this.

An indication may be the error in the energy and the momentum, but even there it is possible that cumulative errors may cancel each other partially, especially in the energy where only RMS (Root Mean Square) or average errors occur.

On the other hand, the solutions suggested by Salmon and Warren may increase the computation time so dramatically, that it sometimes simply doesn't seem worth the trouble in most cases.

So I think we can say that TC produces reasonable results as long as we bear in mind that these worst cases - although rare - may occur from time to time. If we really want to be on the safe side Salmon and Warren suggest to use a small Θ : less than 0.5. Because of the differences between my definition of Θ and the one used by Barnes-Hut and Salmon-Warren this implies for my method $\Theta > \frac{1}{0.5\cdot\sqrt{2}} = \sqrt{2}$ or $\Theta^2 > 2$.

An indication of what this means for the computation time can be found in the table from section 5.3.

Chapter 6

Analysis: Which method for what problem ?

6.1 Introduction

In the previous chapters we have discussed in detail the properties of 3 basic particle simulation methods and we have seen what their typical advantages and disadvantages are. But now comes the question what method should be chosen for a certain problem to obtain an optimal result.

In the literature about particle simulations there was surprisingly little I could find about this topic. There is a lot being said about collisionless and collisional systems, but how to distinguish between these two types given a certain dataset is a question that is left largely unanswered. But exactly this is the most crucial point in deciding which method would be best suited. I started thinking about this subject myself and tried to do the following :

- Determine which parameters/properties of a given dataset could be used to make a reasonable decision.
- Design a simple algorithm - or better : a heuristic - that could be used to make the choice without the need of many complicated computations. Using all 3 methods and see which one gives the best results is certainly not the best option ...
- Write the appropriate code, that should be fast enough to be able to give a clear indication for even very large datasets within reasonable time.

6.2 Parameters and properties

If we need to distinguish between parametersets so to be able to make the right choice, what parameters could then be used to make the best decision ? When I tried to tackle this problem

I started to think of several characteristics that might be interesting enough to be used to predict something about the simulation. I'll discuss the most important ones briefly hereafter :

1. The **number of particles** (N). It may sound a little trivial, but of course if the number of particles is lower than a certain threshold it is useless to use either a Tree-code or a Particle-Mesh method, as Particle-Particle is definitely the most accurate and in that case undoubtedly also the fastest choice.

2. The **distances between each particle and it's closest neighbour** (MD) and the variation in these distances. For each particle we can compute the distance to all other particles. The shortest one found is the minimum interparticle distance (MD) for *that single particle.* The shortest value for all these MD values is the *shortest* minimum interparticle distance (SMI) and the average of all these values is the *average* minimum interparticle distance (AMI).
   The difference between the *shortest* minimum interparticle distance (SMI) and the *average* minimum interparticle distance (AMI) can give a rough indication for the uniformity of the mass-distribution. Note however that 2 very close particles in an otherwise reasonably uniform distribution can give very unreliable information about the uniformity of the *particle* distribution.
   Also note that this value doesn't say anything about the *density* distribution, unless all particles have the same weight !

3. The method mentioned before can be improved by computing the **standard deviation in the minimum distances** (MD) found for all particles. A large standard deviation implies a large variation in these minimum interparticle distances, while a small one implies a more or less uniform distribution. Note that for a grid of particles - which could be considered as very uniform - the standard deviation should be always 0 !

4. The **potential and kinetic energy and the ratio between them**. The potential energy is a measure for the amount of acceleration that would occur during a simulation. The kinetic energy on the other hand says something about the velocity of the particles. If the kinetic energy dominates, the approximated displacement during the whole simulation can be computed using *only* the kinetic energy and the intended duration of the simulation.
   If on the other hand the potential energy dominates we could give an indication about the displacement by *only* using the initial acceleration and assuming it would be constant during the whole simulation.

5. The **mass (density) distribution and it's autocorrelation** (DA). The density distribution can be obtained in a similar way as with the Particle-Mesh method. If we use this result to compute the *autocorrelation* function of the density distribution (see hereafter) we can obtain information about the mass uniformity.
   If the system appears to be more or less uniform according to this observation, the Particle-Mesh method could be used, provided the ratio between the average minimum

interparticle distance and the average displacement (AMI/AD) is large enough to guarantee that many collisions are unlikely (see hereafter).

6. The **average displacement (AD) and it's variation during the intended time of the simulation**. If we have an indication of the size of the per-particle displacement, we know also how likely collisions can be if we combine it with the average minimum inter-particle distance from above.

7. The **ratio between the average minimum interparticle distance (AMI) and the average displacement (AD)**. The combination between the average minimum distance and the average displacement can give useful information about the number of collisions that may occur during the simulation.
   This can make the distinction between collisional and collisionless systems and thus between the choice for either Tree-Code or Particle-Mesh, but of course this only makes sense if the variation in either the average minimum distance and the average displacement is not too large.
   In a distribution that was already found to be reasonably uniform, this could be useful extra information to improve the quality of the decision.

8. **Number of detected collisions** during one simulation step. What we actually do is compute one timestep using a Tree-code like algorithm. In this simulation collisions can occur. If we find a significant number of collisions within this single timestep Particle-Mesh is certainly not the best option. Note that it may also be wise to reconsider the timestep in this case ....

Many of the observations mentioned above were combined into a piece of software that is able to give an indication about the properties of a certain dataset and it also can give a hint what method would be best suited according to these observations. The complete heuristic as was implemented in my analyser program is shown in figure 6.1.

6.3 Autocorrelation

A very important part of the program is the computation of the autocorrelation function. It can give us valuable information about the uniformity of the interparticle distances or about the density distribution and that is why I want to give it a little more attention here.

The autocorrelation of a function $f(x)$ can say something about how regular this function $f(x)$ is over a certain interval Δx : it is the amount of "overlap" we find if we shift the function over a distance Δx to the left or the right. We just measure how large the overlapping area is for a certain value to find the autocorrelation for that distance.
A large overlap indicates a large auto-correlation for this value of Δx and a small overlap on the other hand means a small auto-correlation for Δx.

Of course there exists for every value of Δx a corresponding value for the "overlap" and all these values together gives us a new function : "the autocorrelation function" $A(f(x))$ for

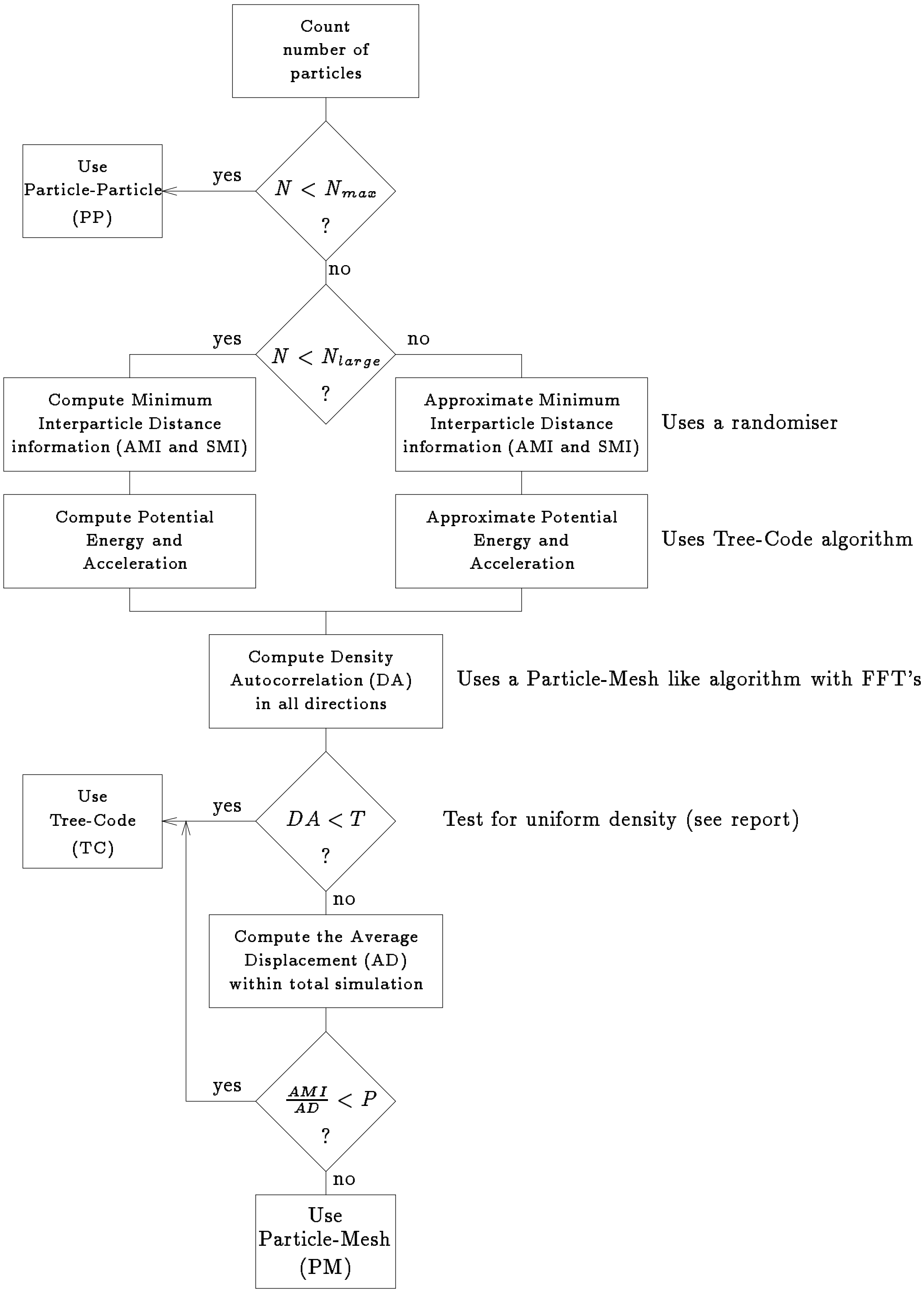


Figure 6.1: Flowchart for analyser program.

our function $f(x)$.

An autocorrelation function can be computed in the following way (note that I write functions of 2 variables now as that is what we are interested in for our analysis) :

$$f(x,y) \Rightarrow A(f(x,y)) = \int_l \int_k f(x,y) \cdot f(x+k, y+l)\, dk\, dl \qquad (6.1)$$

If we define a helper function $g(x,y) = f(-x,-y)$ we can rewrite this equation into a (2-dimensional) convolution of the functions $f(x,y)$ and $g(x,y)$, which is hopefully known to the reader. (if not you may find more about convolutions in [12] or [16]) This yields us for 6.1:

$$A(f(x,y)) = \int_l \int_k f(x,y) \cdot g(k-x, l-y)\, dk\, dl \qquad (6.2)$$

Agreed, this is still not really pleasant to look at, but this will change if we transform the equation above into the (2D) Fourier domain, just as we did with the Poisson equation in chapter 4:

$$\widetilde{A}(f(x,y)) = \widetilde{A}(k,l) = \widetilde{f}(k,l) \cdot \widetilde{g}(k,l) \qquad (6.3)$$

Equation 6.2 then changes into a straightforward multiplication of the two (transformed) functions $\widetilde{f}$ and $\widetilde{g}$.

6.3.1 Discretisation

As with the Poisson equations before, the functions f and g could be replaced with a set of numbers : the values of f and g at the gridpoints that specify the density distribution. The method to setup the density distribution is exactly the same as with the Particle-Mesh method.

That gives us the possibility to use the FFT again to obtain the autocorrelation distribution: transform the distributions f and g into the distributions $\widetilde{f}$ and $\widetilde{g}$, then multiply all corresponding values of $\widetilde{f}$ and $\widetilde{g}$ with each other and finally go back to the spatial domain again using an inverse FFT.

Special attention needs to be paid to the **spacing** and the **dimension** of the grid :

The **gridspacing** should be of approximately the same order as the mean interparticle distance or else we either wash out information or introduce high fluctuations in the autocorrelation function. I choose to make it about twice the size of the average minimum interparticle distance to smoothen the function somewhat and also to reduce the computation time. The gridspacing is also limited by the number of particles: it is useless to have a number of gridcells that is several magnitudes higher than the total number of particles.

As for the **dimension** of the grid : to avoid aliasing it is important that the grid is more than large enough to contain all particles. I calculate the maximum interparticle distance in both directions and set the grid size at about 4 times the size of the particle space.

6.4 Uniform and non-uniform problems

To test the software packages, I used a number of datasets, some of them designed earlier for the Parallel Computation Practice. However, for the larger problems ($> 10^3$) I didn't want to keep the datasets as they tend to occupy a lot of diskspace. For this reason I built 2 small helper programs:

mkgrid : A dataset generator for monotonous grid patterns.

mkrnd : A dataset generator for random patterns.

Both helper programs can also generate random initial speeds if desired. For the random sets applies that the seed for the random generator is always 1, so this should result in every time the same pseudo random sequence if called with the same arguments (if everything works well).

Let's take a closer look now at the differences and similarities for the patterns being generated by these helper programs.

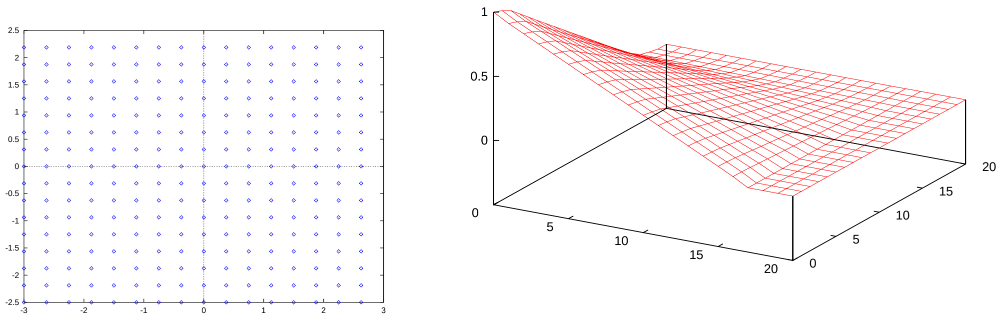


Figure 6.2: A small gridpattern and it's autocorrelation function (256 particles)

For the gridpatterns we can say that they are always uniform, no matter how large the problem is, provided of course that the masses are identical. To see what this means for the autocorrelation function of these patterns, I have plotted a relatively small one and a larger one, together with their autocorrelation functions (see figures 6.2 and 6.3). Apart from a few scale factors you can see that they are virtually the same.

Note that it is not a good idea to have a smaller spacing on the autocorrelation grid than the meshspacing itself, or else we'll see large fluctuations for the autocorrelation function: some cells will be empty while others will still contain 1 particle.

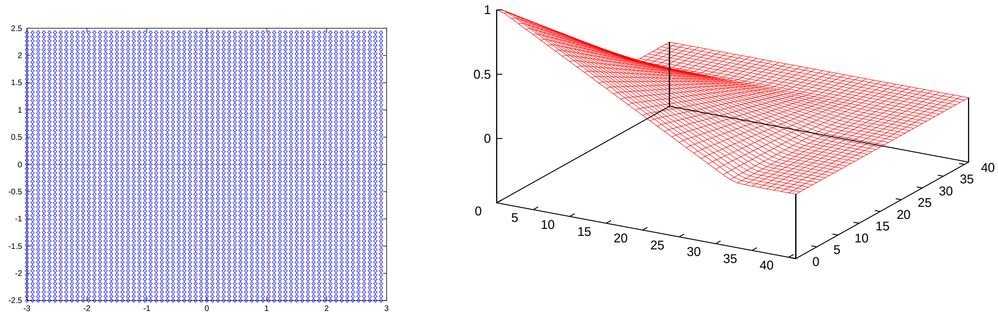


Figure 6.3: A larger gridpattern and it's autocorrelation function (4096 particles)

For the random patterns (datasets where the positions are generated at random), we'll see that the results will be a little different. Examples can be found in figure 6.4 and 6.5. Now the smoothness of the autocorrelation function also depends on the gridsize that is used to determine the density distribution !

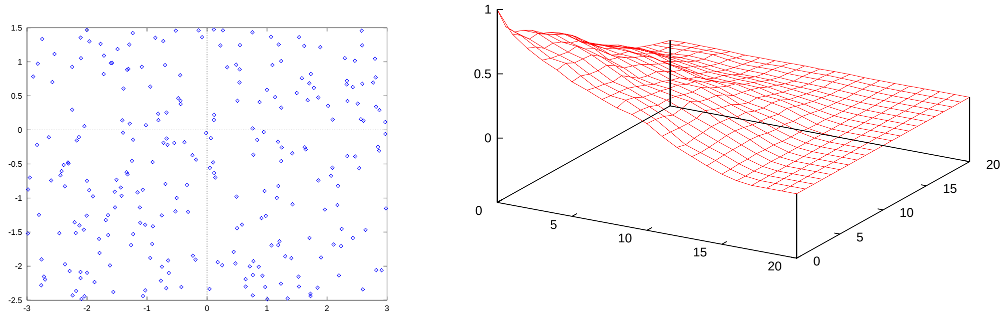


Figure 6.4: A small random generated pattern and it's autocorrelation function (256 particles)

This is caused by the following effect: as soon as the size of a gridcell is large compared with the average interparticle distance, we'll find approximately the same amount of particles in each cell, no matter at which cell we are looking. But if the size of a gridcell is too small, we'll see large variations as the number of particles per cell is going to vary much more in that case than with a monotone grid.

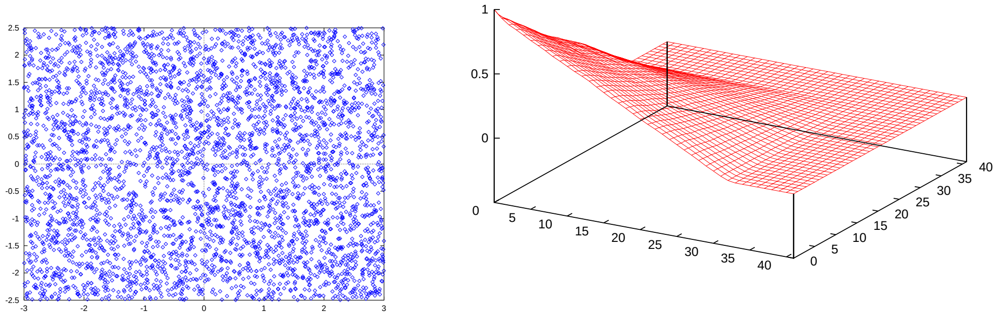


Figure 6.5: A larger random generated pattern and it's autocorrelation function (4096 particles)

Nevertheless we can see that the random pictures also have something in common with the monotonous grid pictures : especially for large randomly generated sets, the autocorrelation function is almost identical with the sets that consist of monotonous grids, as the variation in the density is only noticeable on a very small scale. For random sets larger than approximately 10^5 the difference with monotonous grids of the same size is neglectable, as long as we are not interested in events that take place on a very small scale and of course if the average particle displacement during the simulation is not too large.

A special situation occurs for the random set when the mean particle displacement lays *between* the smallest minimum interparticle distance (SMI) and the average minimum interparticle (AMI) distance. In that case there will be a noticeable difference in the number of collisions between random sets and monotonous sets : for the monotonous sets we may say $SMI = AMI$ so no collisions would be found, while for the random sets $SMI < AMI$ possibly resulting in a certain number of collisions.

Conclusion : On a large scale both monotonous grids and random sets can be considered to be **uniform** as long as the displacement is small enough.

Non-uniform distributions Now let's see what happens with the autocorrelation if we have clearly non-uniform distributions. These distributions can be created by adding some heavy masses on a random set or a grid, by generating a few random sets at different locations in space and merge them into one new set (clusters) or simply by using a set with only a few particles.

Some examples are given in figures 6.6, 6.7 and 6.8. Note that in all these cases the autocorrelation function drops very fast as soon we drift away from the centre. This leads to the conclusion that the average value of the autocorrelation function, especially around the centre, can provide us with the information we are searching for.

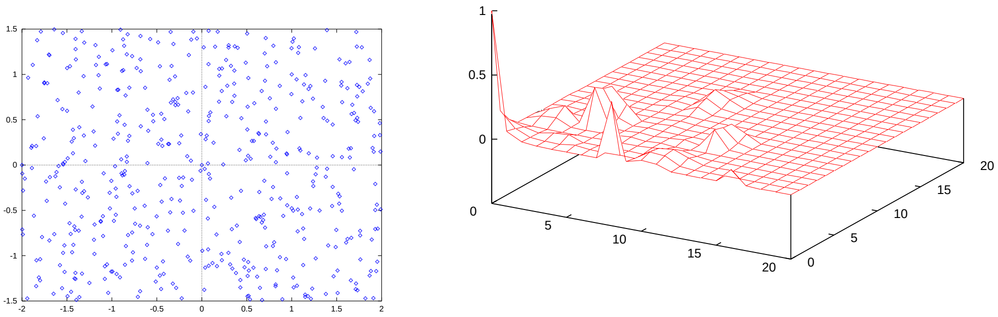


Figure 6.6: 500 particle random set with a few extra heavy masses causing non-uniformity.

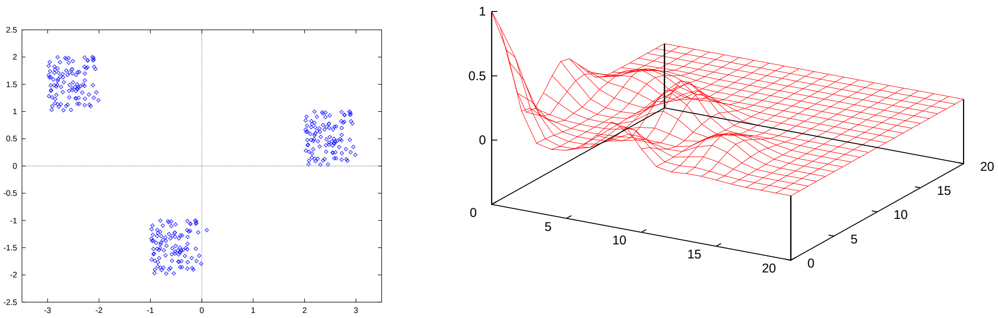


Figure 6.7: 300 particle set built by clustering 3 smaller random sets of 100 particles each.

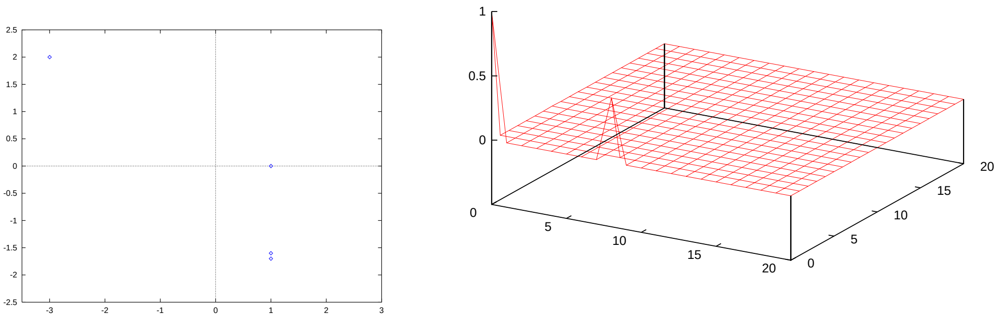


Figure 6.8: Autocorrelation for the 4 particle problem from figure 1.1.

I decided to use the average value of the autocorrelation function in both directions from 0 to $\frac{1}{4}$ of the grid dimension, to make the distinction clear enough. Far away from the centre all pictures show that the autocorrelation is small, so using these values only makes the distinction less clear. This property was given the symbol T in the flowchart from figure 6.1.

In formula the rather simple definition I used for the discriminator T looks like :

$$T = \frac{2}{N}\sum_{x=1}^{N/4} A(x,0) + \frac{2}{N}\sum_{y=1}^{N/4} A(0,y) \tag{6.4}$$

Where the autocorrelation function $A(x, y)$ has to be rescaled to be always between 0 and 1.

Let's now summarise some values for T for true random and grid patterns and compare them with the values found for the 3 non-uniform examples. First we'll look at some gridpatterns generated by **mkgrid** (where SMI and AMI should be essentially the same) :

Particles	T	SMI	AMI
1024	0.512243	0.156250	0.156250
4096	0.505994	0.078125	0.078125
16384	0.500418	0.039062	0.039062
65536	0.489416	0.019531	0.019531
131072	0.496926	0.013810	0.013810

And now for some random sets generated by **mkrnd** (where SMI should be of course less than AMI) :

Particles	T	SMI	AMI
1024	0.262296	0.022565	0.086043
4096	0.402716	0.004072	0.035512
16384	0.469110	0.003728	0.020702
65536	0.489416	0.000439	0.011138
131072	0.492564	0.001258	0.007941

It is clear from the table that the larger the number of particles, the closer the value of T comes to the value found in the gridpattern examples (about $T \sim 0.5$). In the first table we see a very slow drop in the value of T for large numbers of particles. This is probably caused by the fact that I limited the dimension of the mesh to a rather low maximum value to improve the performance.

Finally we also need the T values for the non-uniform examples, to see if the discrimination made by using the value of T would be sufficient :

Particles	T	SMI	AMI
4	0.007852	0.100000	1.568034
300	0.043666	0.002539	0.056207
500	0.003150	0.011130	0.076142

As shown in the table, the values of T are much smaller now. I decided to to call every set with $T > 0.4$ a *uniform* distribution, every set with $T < 0.1$ a *NON-uniform* distribution and all sets with $0.4 > T > 0.1$ sets that need closer inspection. Of course these values are just more or less arbitrary chosen, but nevertheless they are good enough to serve the purpose: a *rough* discrimination between uniform and non-uniform sets.

The sets belonging to the "borderland" could be done using Particle-Mesh as long as the required accuracy is not too high, in the other case it would be better to use Tree-Code, although this may take significantly longer as may be seen in the results from the next chapter.

6.5 Implementation aspects

As usual there were several practical problems to be solved while building the code. I will summon the most interesting ones hereafter :

- It should be possible to compare the factor T between all different types of sets that should be analysed, no matter if they are large or small, uniform or non-uniform. For that reason it was necessary to scale the autocorrelation function back to alllow only values between 0 and 1 (vertical axis).
  The same applies for the horizontal axis : we need the average value of the summation, so it has to be scaled back with the number of particles N.
- The interparticle distance information (SMI and AMI) requires $O(N^2)$ operations. As this would take too much time for the larger sets, I decided to use a randomiser to draw a subset M of particles in this case and do the computation only for this subset, reducing the number of computations to $O(N \cdot M)$.
- A comparable problem as with the interparticle distance information occurs for the computation of the initial acceleration to be able to determine the average displacement. This was solved by using a Tree-Code like algorithm to calculate this acceleration as soon as the number of particles exceeds a certain threshold.
- The average displacement contains important information about the system being collisionless or not. But suppose the system as a whole would be translating with a high speed. As this could make the information totally useless, I decided to always transpose the problem to the Center-Of-Mass system. This of course sets the total momentum to 0 in both directions.

- The meshspacing for the autocorrelation function is important : if it is too large, the function may contain large fluctuations and if it is too small we loose information (this is in fact exactly what happens with the large random sets compared with the meshes !).
  Also the overall meshsize is important: if the dimension is too small this may cause aliasing, while if it is too large we loose information as the autocorrelation drops very fast in that case.

Some examples of the analyser output are given in appendix B.

6.6 Tunable parameters

-v Verbose flag. This will cause extra information to be produced during the analysis.

-w <nr> Width of the mesh for the autocorrelation function.Overrides the automatic computation of the grid and may result in non-comparable values for T.

-h <nr> Height of the mesh for the autocorrelation function. See above.

-n <nr> Number of mesh cells in one direction. See above.

-p Write the autocorrelation function data into a file called "autocorr.dat" and plot the result on the screen using GNUplot.

-l <nr> Threshold for large sets. Above this value the analyser switches to use a random generator for the interparticle distance information and a Tree-Code algorithm for the acceleration computation.

As many of the routines were taken from either the Particle-Mesh or the Tree-Code programs, several of the changeable macros from these programs also apply here: i.e. COLLISIONDISTANCE, LAMBDA etc..

Most of the other parameters that can be changed at compile time can be found in the `Initialize()`routine:

- The Θ factor for the Tree-Code part is currently set to 1 by default.
- The number of samples that is used to determine interparticle distance for large sets is currently set to be $R \cdot \log N$, where the factor R has a default of 10.
- The threshold for the distinction between PP and PM/TC is set to 100 particles by default.

Most of the other values set in the `Initialize()`routine are used to give a reasonable advise after analysis. These numbers were mostly obtained empirically.

6.7 Restrictions

Although this analyser heuristic seems to give reasonable clear indications at this time, it isn't perfect and it may still contain many flaws, as it is still very young and only little tested. Also many of the used characteristics need further investigation. Nevertheless : it is much better than trying to run a set through all 3 methods and see which one is best, if it is kept in mind that the results only give rough indications.

As already said earlier in this chapter : especially the sets that are told to be in the "borderland" need further examination before the final choice for a certain method is made.

Chapter 7

Results and conclusions

To get an idea of how TC compares with PM, I have done a number of tests on both non-uniform sets and uniform sets, the latter consisting of either random generated sets or just monotonous grid patterns. I also ran a number of these tests with PP, but as was to be expected it wasn't possible to run the larger sets within a reasonable time, so I skipped PP for the ones with more than about 2000 particles. Note that in all tests from this chapter *only* 2D forces were used, to ensure that the results of TC and PP could be compared with the PM output.

Some remarks about the time measurements in this chapter : the runtime listed in all results is the so-called "user-time" that was spent when computing on the processor of our HP 755/125 systems. It's value was obtained using the `clock()` function, and as such it doesn't show the "real-time" or "system-time" spent on the program. However, the ratio is about the same for all different sorts of time and that's what we are most interested in.

But unfortunately I found that the values on HP were complete nonsense for the larger sets (taking more than about 1 hour), although on a Linux PC the same values were fine. After a long time of searching I found that the resolution on the HP is so high (10^6), that the available space for the clock value (4 bytes= 2^{32}) only allows runtimes of less than $\frac{2^{32}}{10^6} = 4295$ seconds to be measured. As this can render many hours or even days of computation to be totally useless (and in fact it did so in my case !) I decided to give this some special attention: "forewarned is forearmed" :-)

7.1 Results

First let's take a look at some non-uniform sets (only runtime and energy are shown, the error in the momentum is omitted to avoid giving too much rather meaningless information). Also the results for the 3 non-uniform sets from the previous section can be found in here (see table 7.1).

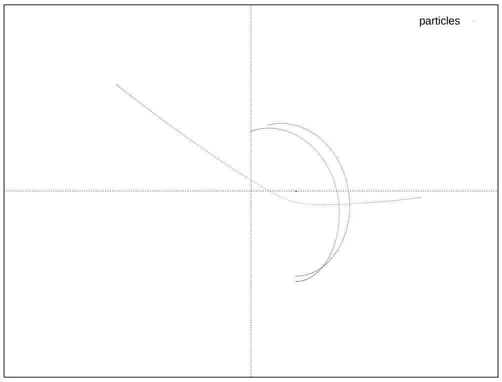
Non-uniform		PP	TC	PM		PP	TC	PM
Particles	Steps	Time per 10000 steps			Meshsize	Energy leak (%)		
4	8000	0.19	1.05	85.0	32x32	27.02 %	27.02 %	-9.540 %
16	2500	2.72	7.20	85.1	32x32	0.198 %	0.199 %	0.188 %
50	10000	26.36	34.53	86.3	32x32	12.20 %	12.26 %	-3.435 %
300 (cluster)	400	955.0	313.3	95.5	32x32	1.319 %	1.319 %	1.025 %
500	50	2700.0	748	106	32x32	0.222 %	0.222 %	0.160 %
2000	30	43880	3907	467	64x64	0.0009 %	0.0009 %	0.0009 %

Table 7.1: Comparison between the 3 methods for several non-uniform sets.

The run for 4 particles couldn't be completed with PM as one particle moved outside the mesh. The error in the energy is now for only 2624 iterations, so about one third of the other 2.

Note that PM always uses a meshsize of at least 32x32, which results in runtimes of about the same order for all problems except for 2000 particles, where the meshsize was increased to 64x64.

There is another important remark to be made : in all cases we can say that - as long as the number of iterations and the timestep are identical - PP must necessarily have the largest accuracy. If there are large errors in the PP results they are almost always caused by collisions and close encounters.



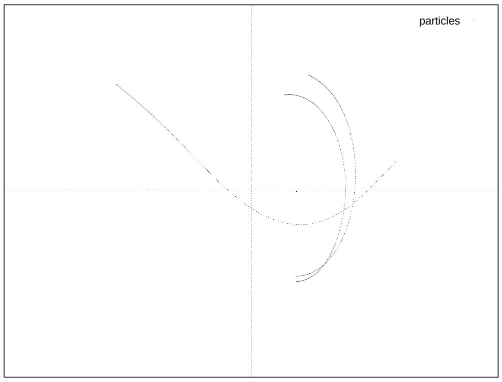


Figure 7.1: Output for the 4 body problem from chapter 1 if run with PP (left) and with PM (right). In both cases a 2D force is used to make them comparable.

Well, TC is as vulnerable for close encounters as PP and so should have more or less the same error as PP, as long as the error made by the tree approximation is small compared with this error. So, the *closer* the error to the error made in PP, the more likely it is that the TC approximation is indeed a good one !

PM does not suffer from collisions and so also doesn't have the problem with energy leaks in close encounters. Energy (and also momentum) are often reasonably well preserved. However, all details caused by the close encounters are completely washed out or blurred too. I.e. the 4 body example from the introduction (with the sun, the planet and the moon: see picture 1.1) runs fine with both TC and PP, but shows something that only vaguely resembles these pictures if run with PM (see picture 7.1).

Now let's take some uniform sets and do the same comparison. Table 7.2 shows the results for the smaller meshes (where PP was still practically possible).

Meshes		PP	TC	PM		PP	TC	PM
Particles	Steps	Time per 10000 steps			Meshsize	Energy leak (%)		
16	10000	2.68	6.50	85.5	32x32	1.094 %	1.093 %	0.956 %
64	2500	43.16	44.04	87.0	32x32	1.460 %	1.460 %	1.182 %
256	1000	694.1	273.5	93.7	32x32	4.702 %	4.699 %	3.564 %
500	100	2682	632.0	104.0	32x32	0.009 %	0.009 %	0.007 %
1024	100	11314	1543	123.0	32x32	5.590 %	5.589 %	4.078 %

Table 7.2: Comparison between the 3 methods for several smaller uniform sets (meshes).

If we put all results from the tables 7.1 and 7.2 in a picture we could compare the efficiency for PP, PM and TC. We see in figure 7.2 that indeed the computation time for PP increases much faster than for PM and TC. It is also nice to see that TC computation time increases smoothly with the number of particles, while PM stays constant for a while, but suddenly jumps if the meshsize has to be increased. The latter stems from the fact that the FFT routines only accept meshsizes that are a power of 2.

I also ran some larger uniform problems - both meshes and random sets - on only PM and TC, to see what their relative accuracy is and what the scaling of the computation time looks like. These larger problems - all generated by the 2 helper programs as mentioned earlier - are the same ones as already discussed in the Auto-Correlation section of chapter 6.

A special problem arose with the computation of the energy error : the potential energy computation is unfortunately an order $O(N^2)$ operation. This implies that for the large problems the bulk of the time would be spent in the energy error computation and not in the simulation part ! To give an idea : the total runtime for 131072 particles while doing the energy error computation was about 10 hours for PM and about 10.5 hours for TC, so this computation of the potential energy made the difference between PM and TC almost completely being washed out if only the total running time is being used.

For real large problems it may be an idea to use a Poisson equation and an FFT to compute an approximation of the potential energy, just as was done in the previous chapter (Analysis).

The output for Tree-Code and the generated sets can be found in table 7.3, while the same results for Particle-Mesh can be found in 7.4. For reasons explained before the runtimes given in these tables are the runtimes without the initialisation and energy error computation.

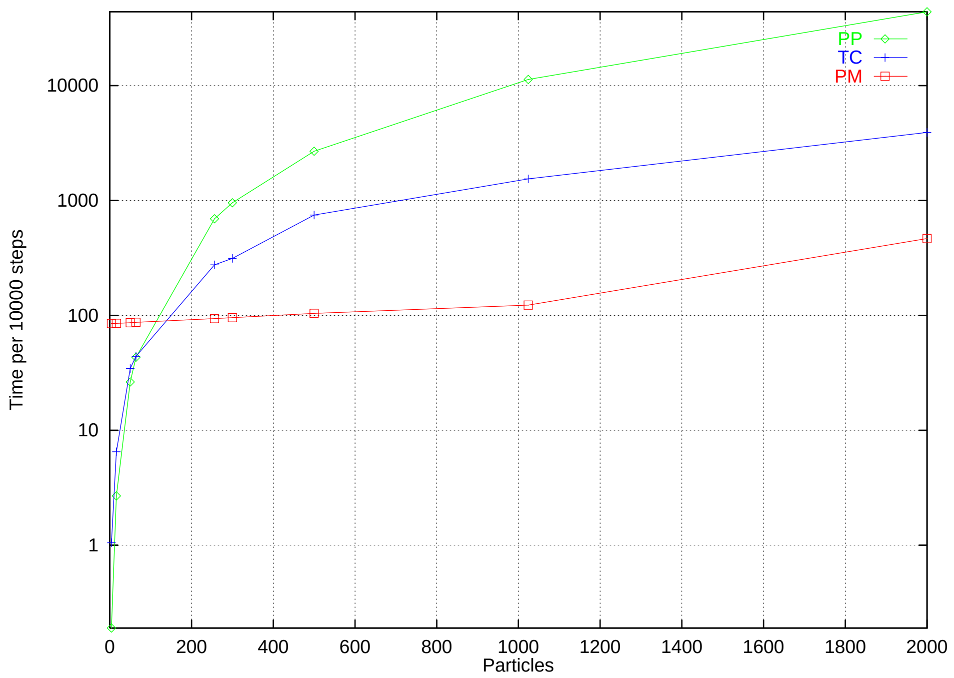


Figure 7.2: Comparison of the PP, TC and PM runtimes for the problems from tables 7.1 and 7.2.

Particles (P)	Steps (S)	Runtime	Energy leak	Mom. leak X	Mom. leak Y
Random sets and Tree-Code ($\Theta = 1$) :					
256	1000	28.32	5.1247 %	3905	-5891
1024	200	31.02	0.7801 %	3373	-2504
4096	100	102.50	0.0397 %	96	-61
16384	50	398.86	0.0067 %	19	11
65536	10	629.77	0.0008 %	31	34
131072	5	943.93	0.0002 %	0.15	46
Monotonous sets and Tree-Code ($\Theta = 1$) :					
256	1000	27.66	4.9800 %	-2750	3103
1024	200	30.52	0.7547 %	-4621	7284
4096	100	86.53	0.0739 %	-177	328
16384	50	381.33	0.0717 %	-663	1165
65536	10	681.62	0.0085 %	-1073	1928
131072	5	812.01	0.0002 %	-20	45

Table 7.3: Results for generated sets and Tree-Code.

Particles (P)	Steps (S)	Meshsize	Runtime	Energy leak	Mom. leak X	Mom. leak Y
Random sets and Particle-Mesh :						
256	1000	32x32	9.43	3.6068 %	36918	39992
1024	200	32x32	2.44	0.5712 %	10176	28111
4096	100	64x64	5.69	0.0322 %	-66.23	-168.8
16384	50	128x128	15.35	0.0048 %	-8.52	-9.75
65536	10	256x256	16.88	0.0067 %	-15.97	-8.83
131072	5	512x512	34.49	0.0002 %	0.611	-3.48
Monotonous sets and Particle-Mesh :						
256	1000	32x32	9.38	3.7670 %	-435.6	-484.1
1024	200	32x32	2.45	0.5713 %	-310.3	-251.0
4096	100	64x64	5.65	0.0559 %	-1.135	-0.928
16384	50	128x128	15.07	0.0543 %	-4.257	-3.532
65536	10	256x256	15.46	0.0070 %	-0.913	-0.755
131072	5	512x512	30.12	0.0002 %	-1.620	-1.692

Table 7.4: Results for generated sets and Particle-Mesh.

To give an idea about the complexity of these problems, the number of computations for $O(N^2)$ and $O(N \cdot \log N)$ operations for all the used combinations of particles and iteration steps is given in table 7.5.

Particles (P)	Steps (S)	$P^2 \cdot S$	$P \cdot \log P \cdot S$
256	1000	$6.5 \cdot 10^7$	$2.0 \cdot 10^6$
1024	200	$2.1 \cdot 10^8$	$2.0 \cdot 10^6$
4096	100	$1.7 \cdot 10^9$	$4.9 \cdot 10^6$
16384	50	$1.3 \cdot 10^{10}$	$1.1 \cdot 10^7$
65536	10	$4.3 \cdot 10^{10}$	$1.0 \cdot 10^7$
131072	5	$8.6 \cdot 10^{10}$	$1.1 \cdot 10^7$

Table 7.5: Total number of computations for both $O(N^2)$ and $O(N \log N)$ for the generated problems.

As should be clear when comparing the numbers from table 7.5 with the results from the tables 7.3 and 7.4, the runtimes increase a little faster than should be expected if we consider that both methods are supposed to scale with $O(N \log N)$. There are however several explanations for this phenomenon :

- Most modern computers have one or more levels of fast cache memory between the processor and the relatively slow main memory. This implies that the performance can drop dramatically as soon as the size of the problem becomes larger than the cache size

on one or more levels.
I checked this sometime ago and found that increasing a problemsize with only 10 %, in worst case situations sometimes results in a runtime that could be more than 10 (!!) times as long as before, only because of these cache effects. As the difference in our case is on average only about a factor 2 to 3, it is reasonable to assume this was indeed caused by caching.

- The number of computations for PM does not directly depend on the number of particles, but on the number of gridcells. Going from 256x256 to 512x512 (as with the step from 65536 to 131072 particles) does not double the number of computations, but of course increases them with a factor 4. So the runtime is in this case almost exactly what we expect for a $O(N \log N)$ behaviour if we take N as the number of *cells* instead of the number of *particles*.

- The recursion with Tree-Code makes heavily use of the stack, sometimes leading to a lot of extra time spent in pushing and pulling return addresses to and from the stack. Furthermore the coordinates are currently only sorted on their X-coordinates and not on both, so in reality the order for the Tree-Code is currently something like $O(N\sqrt{N})$, provided of course that a reasonable uniformity exists.
On the other hand: the weighting factor for the particle count is only small as the time needed for one operation (a particle count in the sortlist) is short compared with the much heavier other operations like the acceleration computation.

7.2 Conclusions

In the introduction I have listed some of the targets for this research project. I think I can say that most of the targets have been reached. However, although I did work already 2 years ago on the parallelization of the PP method described in this report, and although I have already done some work for the parallelization of the TC and PM programs as well, I didn't find the time to complete this unless I would be satisfied with only doing half work, resulting in i.e. a very poor speedup.

For this reason and also because this report is already quite a bit longer than I intended it to be at the beginning, I decided to omit the parallelization part completely and put the results in a separate report, to be submitted probably next year.

Ok, now it is the time to summarize some of the conclusions resulting from this report :

- For large problems Particle-Mesh is certainly the fastest method, but it blurs many of the details. It behaves very bad for relatively small and for non-uniform problems as well as for problems where many collisions are likely to happen. This is why it should be only recommended for more or less uniform, collisionless systems.

- The Tree-Code method described in this report is quite accurate, provided the discriminating factor Θ is large enough and it is also reasonably fast even for these larger values of Θ. Another advantage is that it allows for the proposed collision management methods described earlier in this report.

- The "**sort and count**" technique to build a tree for the Tree-Code problems as described in this report - although discovered more or less accidently through a misunderstanding made by me - proves to be indeed a practically useful method, provided optimal sorting techniques are being used. However, it still needs to be compared with a Tree-Code algorithm according to the standard Barnes-Hut method to see if it really can be faster.

- Within uniform distributions, forces that act only over a short distance in a 2 dimensional simulation may turn into long-distance forces if the same problem is transferred to 3 dimensions. This has to be kept in mind if - as was done in this report - only 2 dimensional problems for gravity and electric forces are being studied just because they are easier to visualize.

- As shown in chapter 6 a rough distinction between sets suited for TC, PM or PP can indeed be made by looking at the auto-correlation for the density distribution, together with the ratio between the average displacement over the intended simulation interval and the average minimum interparticle distance.

- At the risk of reopening an already open door : The "*leapfrog*" integration method is the most simple integration method that guarantees a reasonable accuracy. "*Runge-Kutta*", although quite accurate for only a few integration steps, is not recommended for simulations over a longer period due to it's numerical instability.

- $O(N \cdot \log N)$ behavior - although theoretically available in both TC and PM - is practically difficult to realize, except for relatively small problems. As soon as memory cache sizes are exceeded, on most of the modern computer architectures the runtime tends to grow slightly faster with increasing N than the theory suggests.[1]

[1] A possible remedy for this problem is to partition the data to increase the locality of the computation. Such positioning techniques can also be applied on the parallelization which is to be a topic of future work.

Appendix A

Parameter files

The simulation programs described in this report are all working on parameter files with exactly the same layout, so all predefined problems can be directly run with each of the supplied programs. The parameter file is used to specify a number of particles with their initial speeds, positions and masses, the size of the timestep and the number of iterations without retyping them every time a simulation is run.

A.1 Parameterfile layout

The layout of a parameterfile is as follows :

line 1	Total number of particles in simulation
line 2	Number of iterations between 2 screen updates
line 3	Timestep between 2 iterations
line 4	Total number of iterations
line 5	Position (x,y,z) , Speed (x,y,z) and Mass of particle 1
line 6	Position (x,y,z) , Speed (x,y,z) and Mass of particle 2
....	etc.

It is almost the same format as I use already quite some time for the Multibody simulation problem in the Parallel Computation Practice at the Mathematical Department of Delft University. The only difference in this older format is that not the number of iterations (integer), but the total simulation time (floating point) is given.

A.2 Scaling parameter files

All parameter files I used were based on scaling factors I introduced a few years ago for an exercise in the Parallel Computation Practice. But sometimes it can be necessary to scale the coordinates for a certain problem to different dimensions - for instance with a scalefactor 'f' - without changing the problem topologically.

I found this wasn't as easy as it looked at first glance : not only the position, but also the velocity and/or mass have to be scaled to obtain the same (although scaled) picture. For that reason I give a brief overview hereafter : both as a reminder for myself and possibly for others as well.

What is the crucial thing to conserve the topology ? Simply that the *ratio* between kinetic and potential energy remains exactly the same ! That leaves us with 2 possiblities :

A.2.1 E_{kin} and E_{pot} both invariant

We know that $E_{pot} = \frac{-\gamma \cdot m_a \cdot m_b}{r}$ should remain invariant. But the distance transforms according to $r_2 = f \cdot r_1$, so the only way to keep E_{pot} constant is to scale all masses using the relation $m_2 = \sqrt{f} \cdot m_1$. What effect has this on E_{kin} ? The mass is scaled with $\sqrt{f}$, so v^2 should scale with $\frac{1}{\sqrt{f}}$ to compensate for this, and thus we find $v_2 = f^{-\frac{1}{4}} \cdot v_1$.

To have them altogether in one table (including the timestep scalefactor) :

$$x_2 = f \cdot x_1 \tag{A.1}$$
$$r_2 = f \cdot r_1 \tag{A.2}$$
$$m_2 = f^{\frac{1}{2}} \cdot m_1 \tag{A.3}$$
$$v_2 = f^{-\frac{1}{4}} \cdot v_1 \tag{A.4}$$
$$\Delta t_2 = f^{\frac{5}{4}} \cdot \Delta t_1 \tag{A.5}$$

A.2.2 E_{kin} and E_{pot} both scaled with the same factor

Now suppose we want to scale the energy with the same scale factor f as we used to scale the relative positions. To achieve that, we can scale all masses with f as well: This yields us $E_{pot_2} = f \cdot E_{pot_1}$ as one factor for mass cancels out. But this immediately results in $E_{kin_2} = f \cdot E_{kin_1}$ without doing anything with the velocity ! Again in one table :

$$x_2 = f \cdot x_1 \tag{A.6}$$
$$r_2 = f \cdot r_1 \tag{A.7}$$
$$m_2 = f \cdot m_1 \tag{A.8}$$
$$v_2 = v_1 \tag{A.9}$$

$$\Delta t_2 \quad = \quad f \cdot \Delta t_1 \tag{A.10}$$

So, obviously the easiest transformation is to scale both the mass, the timestep and the positions with the same scalefactor, and to keep the velocity at its original value.

Appendix B

Some examples of analyser output

B.1 Example of a 1024 particle mesh problem

```
**************************************************
**** Analyzer for particle simulation problems *****
****             (c) 1997 Kees Lemmens        *****
*************************************************

Problem to be analyzed     :  ../TF/1024p_mesh_1.par

Problem characteristics :
-------------------------
Number of particles        :  1024
Center of mass speed (x,y) :  +0.000000, +0.000000
Total kinetic energy       :  +0.000000
Total potential energy     :  -3196246.238663
Total energy               :  -3196246.238663
Total momentum X direction :  +0.000000
Total momentum Y direction :  +0.000000


Used randomizer for distance info    :  Yes
Used Tree-Code for acceleration info :  Yes
Tree dimensions (wxh)                :  12.40x9.30


Average minimum interparticle distance    :  +0.150000
Minimum interparticle distance found      :  +0.150000
Maximum interparticle distance found      :  +7.590949
Standard deviation interparticle distance :  +0.000000
```

```
Standard deviation (percentage)          :  0.000006 %


Average displacement in timeinterval :  +0.206770
Maximum displacement in timeinterval :  +0.492233
Average distance/displacement ratio  :  +0.725442


Number of detected collisions        :  0
Collision detect distance            :  0.001000


Mesh dimension for autocorrelation   :  128
Average value for density autocorrelation in testarea:  0.516667


Problem analysis :
------------------

This seems a quite uniform density distribution, suited for
Particle-Mesh.  The average particle displacement within the intended
interval seems to be of the same order as the average interparticle
distance.  This implies that some collisions may occur.
```

B.2 Example of a 300 particle non-uniform problem

```
*****************************************************
**** Analyzer for particle simulation problems *****
****              (c) 1997 Kees Lemmens       *****
****************************************************

Problem to be analyzed     :  ../TF/300p_clr_1.par

Problem characteristics :
-------------------------
Number of particles        :  300
Center of mass speed (x,y) :  +0.000000, +0.000000
Total kinetic energy       :  +0.000000
Total potential energy     :  -7013460.021299
Total energy               :  -7013460.021299
Total momentum X direction :  +0.000000
Total momentum Y direction :  +0.000000
```

```
Used randomizer for distance info    :  Yes
Used Tree-Code for acceleration info :  Yes
Tree dimensions (wxh)                :  11.94x7.95


Average minimum interparticle distance    :  +0.056207
Minimum interparticle distance found      :  +0.002539
Maximum interparticle distance found      :  +6.085522
Standard deviation interparticle distance :  +0.031029
Standard deviation (percentage)           :  55.205967 %


Average displacement in timeinterval :  +1.231851
Maximum displacement in timeinterval :  +41.471891
Average distance/displacement ratio  :  +0.045628


Number of detected collisions        :  0
Collision detect distance            :  0.001000


Mesh dimension for autocorrelation   :  512


Average value for density autocorrelation in testarea:  0.043666


Problem analysis :
------------------

This looks like a NON-uniform density distribution.
The average particle displacement within the intended interval seems
to be much larger than the average interparticle distance, so many
(near) collisions are very likely to happen.
```

Appendix C

Source code

C.1 Particle-Particle

C.2 Particle-Mesh

C.3 Tree-Code

C.4 Analyser

Bibliography

[1] J.H.J. Almering e.a., *Analyse* (Delft: Delftse Uitgevers Maatschappij, 1993).

[2] M. Alonso & E. J. Finn, *Fundamentele Natuurkunde deel 1: Mechanica*, (Overberg: Delta Press, 1989).

[3] M. Alonso & E. J. Finn, *Fundamentele Natuurkunde deel 3: Elektromagnetisme*, (Overberg: Delta Press, 1989).

[4] J. Barnes and P. Hut, *A hierarchical O(N log N) force calculation algorithm*, (Nature, vol. 324, p. 446- , Dec 1986).

[5] C.K. Birdsall and A.B. Langdon, *Plasma Physics via Computer Simulation*,(Greenbelt: McGraw-Hill Inc., 1985).

[6] W. E. Boyce and R. C. DiPrima, *Elementary Differential Equations and Boundary Value Problems*, (4th ed.), (New York: John Wiley & Sons, 1986).

[7] L. Greengard and V. Rokhlin, *A Fast Algorithm for Particle Simulations*, (Journal of Computational Physics, v 73, 1987).

[8] A. Graps, *Amara's Recap of Particle Simulation Methods*, (http://www.amara.com/papers/nbody.html, 1996 (taken from the WWW)).

[9] R.W. Hockney and J.W. Eastwood, *Computer Simulations using Particles*, (Bristol: Institute of Physics Publishing, 1994).

[10] C. van Loan, *Computational Frameworks for the Fast Fourier Transform*,Vol. 10 in series *Frontiers in Applied Mathematics*, (Philadelphia: Society for Industrial and Applied Mathematics, 1992).

[11] P. MacNeice, *Particle-Mesh Techniques*, (Greenbelt: Goddard Space Flight Center (NASA), 1996 (taken from the WWW)).

[12] A. V. Oppenheim, A. S. Willsky and I. T. Young, *Signals and Systems*, (London: Prentice-Hall International Editions, 1983).

[13] W.H. Press, S.A. Teukolsky, W.T. Vettering and B.P. Flannery, *Numerical Recipes in C*, (Cambridge University Press 1988, 1992).

[14] John K. Salmon and Michael S. Warren, *Skeletons from the Treecode Closet*, (Journal of Computational Physics, July 31, 1992; revisited March 10,1993).

[15] G. Strang, *Linear Algebra and its applications*, (Orlando: Harcourt Brace & Company Publishers 1988).

[16] W. A. Strauss, *Partial Differential Equations: An Introduction*, (Singapore: John Wiley & Sons, 1992).