Cosmological Applications of Multi-Grid Methods

Andrew David Green
St. Peter's College
D. Phil

Trinity Term, 2000

Cosmological Applications of Multi-Grid Methods

Andrew David Green
St. Peter's College
D. Phil
Trinity Term, 2000

Abstract

A new N-body simulation code, Multi-Level Adaptive Particle Mesh (MLAPM) is described. MLAPM is a software tool designed to conduct numerical simulations of large scale structure formation in the universe. MLAPM is an adaptive particle mesh code where adaptive sub-grids of successively higher resolution are placed in regions of high density. Each level of sub-grid is formed from cells which have a spacing of half the next coarser grid. The sub-grids can have arbitrary shape and closely follow the particle distribution. The code, which is written in the C programming language, uses a data structure designed to support such an adaptive grid, while minimizing the amount of computer memory used. The potential solver is based around the adaptive multi-grid method first described by Brandt (Brandt, 1977a), which allows the Poisson equation to be solved on an irregularly shaped grid. The multi-grid methods use mesh-relaxation techniques combined with the use of coarser grids to speed up convergence. To integrate the particle equations of motion accurately, multiple timesteps are used. Particles which lie within the boundaries of a sub-grid, use a timestep half that of the particles which lie within the next coarser grid. The code is fully described in the text. Test results are also presented, including those from Zel'dovich plane waves, which have an analytic solution.

Contents

List of Figures

List of Tables

Preface & Acknowledgements

The Road goes ever on and on
Down from the door where it began.
Now far ahead the road has gone,
And I must follow, if I can.

Pursuing it with weary feet,
Until it joins some larger way.
Where many paths and errands meet,
And whither then? I cannot say.

The Fellowship of the Ring, J. J. R. Tolkein.

As I add the finishing touches to this thesis, I can say it has been a long arduous journey, although one that is now coming to an end. There are many people I wish to thank.

Professor J. J. Binney was my supervisor for this research project. I would like to thank him for all his help during the project. The many discussions we held have helped make this work possible.

I would also like to thank the following people: Dr. Alexander Knebe, for providing some test data and some useful discussion; Dr. Suvendra Dutta for many useful conversations during the early part of this project, and his continued friendship thereafter and Dr Tom Theuns, who helped get me started with N-body simulation codes. I would also like to thank the many other friends and colleagues in the departments of Theoretical Physics and Astrophysics with whom I had useful conversations.

I would also like to thank St. Peter's college for its support, and for the St. Peter's Society scholarship. I also acknowledge the receipt of a PPARC graduate student-ship.

On a more personal note I would like to thank: Michael Punt and Rowena Wilkinson, for keeping me going when all looked bleak, and many colleagues at Barclays Capital, for encouraging me to get the project completed. Finally, I would like to thank my parents for all their encouragement over the years, although sadly my mother did not live to see this work finished.

Andrew Green,
London,
September 2000.

Chapter 1

Introduction

The study of the evolution of the universe provides the scientist with a set of unique problems. Like all astronomers, the cosmologist cannot perform experiments, only observe the universe around us and draw conclusions. For the theorist modeling the universe provides an opportunity to "experiment", allowing the scientist to modify the model to fit observation and so gain understanding. Modelling the universe however, is far from simple. By definition cosmology encompasses all science, but clearly cosmologists cannot make any headway by including every detail of the all the physics we know from terrestrial studies into models of the Universe. In addition the range of length scales that needs to be modeled is vast, stretching from the inner workings of stars to the visible universe and beyond. Hence the models that cosmologists build are about compromise and are in general simplistic. As time progresses and computational power increases, the models become more sophisticated. These models, in general, involve N-body simulations, where the matter in the universe is modeled as a series of discrete masses. In the simplest case the only interactions the masses are allowed to have is gravitational, although recent attempts have included the effects of gas hydrodynamics.

This thesis describes one such N-body simulation code. This code tries to make efficient use of computational resources, i.e. it tries to optimize the use of CPU time and memory. As the reader will find described later (see chapter 3) there are several approaches to N-body simulations which basically differ in the way in which inter-particle forces are determined. The code described here uses a grid on which to solve for the gravitational potential of the "N-body" particle distribution, the forces being obtained by differencing the potential and then by interpolation. The resolution of this grid is adaptive, refining in regions of high density to maintain the accuracy of the calculation. The potential solver uses *Multi-grid Methods* which are able to cope with a non-uniform grid, unlike the more familiar *Fast Fourier Transform.* In addition the code uses a set of many timesteps, rather than a single one, reducing the number of calculations required for some of the particles.

The outline of the thesis is as follows. In chapter 2, there is a brief description of the history of cosmology, with an emphasis on the study of the large scale structure of the universe. The following chapter, 3, describes the theoretical underpinnings of the standard approach to modeling the gravitational evolution of mass in the universe. Chapter 4 describes *Multi-grid Methods* in detail. Chapter 5 describes the data structure used with the code to store intermediate results, this is of particular importance given the aim of efficiency. Chapter 6, describes how the equations of motion of the particles are integrated, including the use of multiple timesteps. The code is described in detail in chapter 7, this chapter will prove useful to the reader wishing to use the code. The tests and results of using the code can be found in chapter 8 and conclusions can be found in chapter 9.

Chapter 2

A Brief History of Cosmology

2.1 "Island Universes"

The size, form and age of the Universe has been a matter of debate throughout human history. The first recognizably scientific approaches to the problem begin with Newton, who argued for an infinite universe in a letter to Bentley in 1692 (Berstein and Feinberg, 1986). However, Newton did take seriously the age of the Universe derived by Bishop James Ussher. In 1654 Ussher determined that the Universe began on October 26, 4004 BC, after studying biblical chronology. The size of the Universe was not known in the seventeenth century although the stars were known to be at much greater distances that the planets as no parallax had been observed. Huygens used a relative brightness argument between the Sun and Sirius to obtain a distance to the star that agreed to within an order of magnitude with modern estimates.

The nature of the "spiral nebulae" was a matter of debate for several centuries. Herschel believed that the nebulae were stellar systems in their own right and that stars would eventually be resolved by more powerful telescopes. This belief originated in the philosophical works of Wright, Lambert, and Kant (Lang and Gingerich, 1979). Kant, in his book *Universal Natural History and Theory of the Heavens*, took up the idea of Thomas Wright, that the Milky Way was a disk-shaped collection of stars, which we view from the inside, and is slowly rotating. Kant proposed that the nebulae were similar objects viewed externally. Kant also assumed that the collection of nebulae was infinite in extent.

The nineteenth century was dominated by arguments over the age of the Earth. A body of geological evidence had build up suggesting that the Earth was much older that Bishop Ussher's estimate. Charles Lyell proposed his *Doctrine of Uniformity* in his book *The Principles of Geology*[1]. Lyell proposed that the geology of the Earth could be accounted for by slow processes working uniformly over time. Lord Kelvin, however, argued against this idea in 1866. Kelvin argued that the Earth could be no older than 20 million years as this is the time scale he calculated for the Earth to cool from a molten state. Kelvin had left out any assumptions about Radioactivity and hence obtained an estimate that was much too short. This was later corrected by Rutherford, to give a value close to the modern estimate of 4.5 billion years.

By 1914 the standard model of cosmology was the *Kapteyn system* (Eddington, 1914). The distribution of stars in the Milky Way was known to be a flatten spheroid and the sun was believed to be close to the centre of the stellar system. The idea that the spiral nebulae were "island universes" had become less popular when observations of nebulae in general showed them to be gaseous in nature. Large numbers of spiral nebulae were known to exist and also that there were many more faint nebulae than bright ones (see Fath (1914), Sanford (1917) and Curtis (1918)). Discussions over whether the spiral nebulae were internal or external to the stellar system culminated in the Curtis-Shapley debate at the National Academy of Sciences in 1920 (Lang and Gingerich, 1979).

[1] Darwin took this book on his first voyage on the *Beagle*

Shapley firmly believed that the nebulae were part of the galaxy. He cited proper motion measurements by van Maanan in 1916 that suggested huge linear velocities for the nebulae if they were at great distances. These proper motion measurements were not found by other observers and no motions were found when Hubble re-examined the plates from which van Maanan took his measurements. Shapley also argued against the Kapteyn system and its heliocentric view. Shapley placed the sun at around 50,000pc from the centre of the stellar system. This distance was later shown to be a large over estimate as Shapley took no account of interstellar absorption, but Shapley's model of the Milky Way was largely correct.

Curtis argued for both the island universe hypothesis and the Kapteyn system. His arguments for the island universe hypothesis were clearly put in his paper (Curtis, 1919). Curtis used an estimate of the angular size of the milky way when placed at the distances of the spiral nebulae (derived from novae) to uphold his case. One popular argument against the island universe hypothesis was the distribution of the spiral nebulae, which seemed to be concentrated in the north and south galactic poles, suggesting they were part of the stellar system. Curtis observed that many of the spiral nebulae that are orientated with the disk edge on to the observer have absorbing matter in the plane of the disk. Curtis suggested that the Milky way would also contain such material and that absorption accounted for the absence of the nebulae from the plane of the Milky Way's disk.

The argument was finally settled by Hubble in (1925). Hubble observed Cephid variables in M31 and M33 and derived a distance of 285,000pc. An earlier paper by Öpik (1922) derived a distance of 480,000pc to M31, a value close to the modern figure. He used rotation measurements and assumed a mass to light ratio equal to that of stars in the Milky Way. The following year Hubble (1926) introduced his classification of galaxies.

2.2 The Homogeneous Universe

In 1917 Einstein introduced his homogeneous model of the universe using General Relativity and Mach's principle (Einstein, 1917). Einstein rejected the model of the Universe with a stellar system isolated in a flat background space-time as this violated Mach's principle, although it is not clear how much he knew about the debate over the spiral nebulae. Einstein initially proposed that the metric g_{ij} would become singular far from the matter in the universe but then rejected this in favour of his closed homogeneous model.

Also in 1917 de Sitter (1917) proposed a second "static"[2] of the homogeneous solution of Einstein's equations. de Sitter's solution was based on ideas of Newcomb and was a space empty of matter. The components of the metric, $g_{\mu\nu}$, vanish at spatial and time infinity and this is de Sitter's covariant statement of Mach's principle. The solution still requires a cosmological constant. de Sitter freely admits that while the solution "satisfies the 'mathematical postulate' of relativity of inertia" it "does not appear to admit of a simple physical interpretation".

Observational evidence was slow in coming. It was not until Hubble's 1926 paper that questions about the distribution of extra-galactic nebulae were finally asked (Hubble, 1926). In 1931 Hubble (1931) produced a preliminary paper on the results from a survey of extra-galactic nebulae and later in 1934 the full results from the survey (Hubble, 1934). Hubble concluded that the distribution of the nebulae was uniform as it followed the law that the number of galaxies N brighter than magnitude m was given by

$$N(< m) \propto 10^{0.6m}. \tag{2.1}$$

Later Hubble used a deeper survey and found the distribution diverged from uniform (Hubble,

[2]We should be careful how we interpret the word static. The de Sitter model can be viewed as static in two ways, firstly de Sitter used a coordinate system in which the coefficients of the metric are independent of time, secondly the three dimensional curvature (which is zero) and the four-dimensional curvature (which is positive) are both constant in time. de Sitter clearly viewed his solution as a static one for these reasons, although he was also aware that test particles would expand away from each other. Indeed de Sitter later went on to predict cosmological red-shift would occur. We are more familiar with the de Sitter solution used in the context of *inflationary* models and would thus view the solution as a special case of an expanding (and thus non-static) universe.

1936) as the galaxy counts fell below the $10^{0.6m}$ law. Hubble still concluded that the overall distribution of the nebulae was uniform as the discrepancies were small.

While Hubble was happy to conclude the universe was homogeneous despite his 1936 results, others had differing views. Shapley (1938) suggested large-scale density irregularities may be responsible for Hubble's result. Eddington (1939) and later Tolman (1949) also suggested large-scale inhomogeneity could account for Hubble's 1936 results. Tolman, while considering the age of the Universe, concluded on those grounds that a homogeneous world model was not consistent with observations, as it gave a Universe younger than the apparent age of the Earth. Tolman remarks that it was popular at the time to suggest that the problem lay with General Relativity and not with the model, and he concludes that there are two apparent reasons for this, firstly that General Relativity is hard to understand, and secondly that it was an "open-ended kind of suggestion which allows free rein to unbridled fancy".

In general, after the conclusion of the Second World War, the debate over the large scale homogeneity or otherwise of the Universe was over. Of course perhaps the most telling piece of evidence for the homogeneity of the Universe came later with the discovery of the *Cosmic Microwave Background Radiation* (Penzias and Wilson, 1965).

2.3 The Expanding Universe

The first person to consider non-static[3] relativistic cosmologies was Aleksandr Friedmann. Friedmann considered the case of a non-static homogeneous model with positive spatial curvature (Friedmann, 1922). In the same paper Friedmann also shows that Einstein and de Sitter have exhausted all possible static homogeneous models. In a second paper, Friedmann (1924) discussed static and non-static homogeneous models with negative spatial curvature. Friedmann's work had little immediate impact on observational cosmology. Einstein, however, did take note and issued a critique in the same journal (Einstein, 1922) arguing that Friedmann had made an error in his calculations. He retracted his objection with similar speed in (1923).

Other theorists had objections to the Einstein and de Sitter models[4]. Weyl and Eddington (1924) pointed out that a physical variable, ρ, is set equal to a constant of nature, Λ. Much later Tolman showed that the model is unstable (Tolman, 1934).

The Einstein and de Sitter models held sway with observers from some time. It is perhaps Lemaître's (1927) paper that changes the situation although it is not clear how widely Lemaître's work was known before the 1927 paper was translated into English, appearing in *Monthly Notices of the Royal Astronomical Society* in 1931. Lemaître seeks for an intermediate solution between that of Einstein and de Sitter, recognizing that the de Sitter solution provided an explanation for the observed red-shifts of the nebulae while not admitting an easy physical interpretation because it contains zero mass. Lemaître generalizes the Einstein model with a non-zero pressure term and obtains a model that expands away from a static start. The recession velocities of the nebulae are interpreted as the expansion of the Universe. A reference to Friedmann's (1922) paper appears at the end of the Monthly Notices translation but the work is not mentioned within the paper itself.

The experimental evidence for expansion had been building for some time culminating in Hubble's (1929) paper. Hubble re-examined the question of a velocity–distance relation for the nebulae, and, with the benefit of reasonable data, derived a roughly linear relationship between velocity and distance. Hubble noted that this result may represent the de Sitter effect.

In the early 1930's theoretical debate centred on Lemaître's models. Lemaître reports (1931) that Eddington has suggested that the formation of structure may have precipitated the expansion of the Universe, while McCrae and McVittie were of the opposite opinion. The emphasis was still on models which expanded away from the static Einstein case. McCrae and McVittie (1931) report Einstein wishing to drop the Λ term from his field equations, and note that this would eliminate the problem of how the Universe would depart from an equilibrium state as it would never have

[3]See footnote in §2.2 for the use of the word "static".

[4]Here I refer to the Einstein (1917) and de Sitter (1917) models and not the later Einstein-de Sitter model proposed in 1932, which is the familiar spatially flat expanding universe

been in one! Subsequently, the standard cosmological models became those of Friedmann which were re-derived with rigor from symmetry arguments by Robertson (1929).

2.4 Departures from Homogeneity

Although early discussions about the nature of the Universe centred on questions of overall isotropy and homogeneity, there was a recognition that this was only a first step. Lemaître ((1933a), (1933b) and (1934)), like Jeans (1928) before him, supposed that gravitational instability was responsible for the growth of structure. Lemaître thought that their would be small irregularities in the distribution of matter, early in the Universe's history. Lemaître left the origin of these fluctuations to later theories. The irregularities would remain in the quasi-static phase of Lemaître's expanding Universe model longer than the surrounding lower density material. The over dense regions would then collapse to form galaxies and clusters of galaxies.

The next major theoretical contribution to structure formation through gravitational instability was made by Lifshitz (1946) in his general analysis of linear perturbations of the Friedmann-Lemaître model. Lifshitz concluded, incorrectly, that gravitational instability was not the process behind the formation of galaxies. Later Bonner ((1956), (1957)) pointed out that under gravitation, the density contrast grows as

$$\delta\rho/\rho \sim t^{\frac{2}{3}}$$

which would not account for the structure in the universe if clustering began at time 1s.

Lifshitz and Bonner's arguments dominated discussion of the growth of clustering throughout the 50's and 60's. Alternative theories for the growth of structure were proposed such as *primeval turbulence* (e.g. Gamow (1952)). Hoyle (1949) suggested the thermal instability might form galaxies in the steady state model.

Novikov eventually pointed out the error in Lifshitz' analysis (Novikov, 1964) although others had also concluded in favour of gravitational instability. Oort (1958) concluded that gravitational instability could account for cluster formation and van Albeda (1960) was able to find a numerical solution that resembled a real cluster while modeling gravitational evolution of the single galaxy distribution function.

2.5 The Hot Big Bang and the Cosmic Microwave Background

The story of the discovery of the Cosmic Microwave Background Radiation [5] comes in two, disconnected, parts. The first part of the story is that of Gamow and his collaborators Ralph Alpher and Robert Herman.

Just after the second world war Gamow set out to explain the chemical abundances in the Universe by using nuclear physics [6]. With the recognition that the early phase of the expanding Universe would have been *Hot*, Gamow (1948) showed that the processes that led to the formation of the elements must have began with deuterons at a temperature of $T_0 \simeq 10^9 K$. With this temperature the Universe was only minutes old and electromagnetic radiation would have been in equilibrium with matter. Later the radiation goes out of thermal equilibrium with the matter ($T \approx 1000K$). Gamow recognized the significance of this epoch as the only time when gravitational instability can begin to work. Gamow's collaborators, Alpher and Herman, reworked Gamow's ideas (1949) and predicted the temperature of the background radiation today would be $T \approx 5K$, remarkably close to the accepted value. Sadly the work was forgotten, and it was another fifteen years before the background radiation was discovered.

[5] CMBR

[6] Of course nuclear and particle physics took a huge leap forward during the second world war through the Manhattan project

Throughout the fifties three cosmological models competed for popularity (although at the time physicists as a whole were more concerned with developments in particle physics). These three were Big Bang theory, the Steady State theory of Gold, Hoyle and Bondi (see for example Hoyle (1958)) and Milne's kinematic cosmology. Milne's non-relatistic cosmology had been formulated during the thirties as a response to the horizon problem and other perceived difficulties of relativistic cosmologies. The discovery of the background radiation would prove the death knell for the Big Bang model's competitors.

In the early sixties, Dicke, started to considered oscillating cosmological models with periods of expansion and contraction (1965). In dense phases, Dicke and Peebles noted that the Universe would have been filled with blackbody radiation and that subsequent expansion would be adiabatic, cooling the radiation but preserving its thermal character. An upper limit of $40K$ could be placed on the radiation from measurements of the Hubble constant and with that in mind Roll and Wilkinson began to construct a radiometer at Princeton to study radiation at 3cm. When the group received a telephone call from Penzias and Wilson (1965) at Bell Labs they knew they had lost the race to discover the background radiation.

2.6 Primordial Nucleosynthesis

Early debate over the origin of the elements, in the 1940's and 50's was linked to the debate over cosmological models. Two candidates were available for element synthesis, the interior of stars, and the early stages of a hot big bang. In both cases the temperatures would be sufficient to overcome the Coulomb barrier and allow nuclear reactions to take place. It turns out that primordial nucleosynthesis is one of the great successes of the hot big bang theory.

The idea of primordial nucleosynthesis goes back to Gamow's paper (1948) (see §2.5). Early work was done by Hayashi (1950), but the work of (Alpher et al., 1953) contains most of the features of more modern numerical codes. Primordial nucleosynthesis was revisited by Peebles (1966) and others when, for example, knowledge of weak interactions became available and better estimates of interaction cross-sections become known.

The most significant feature of the theory of primordial nucleosynthesis is the sensitivity to the neutron-proton ratio, η. As practically all neutrons in the early universe end up in Helium-4, with remainder in primordial Deuterium, Helium-3, Tritium, Lithium and Berilium. Knowledge of the abundances of these elements places limits on the value of η in the early universe. In addition, η is directly proportional to the baryon number of the universe and so primordial nucleosynthesis can be used as a probe of Ω_b. An attempt to match the observed Helium-4 abundance was made by Peebles (1966). A review by Primack (1999) summarizes recent measurements of primordial abundances, and concludes that $\Omega_b \approx 0.3$.

2.7 Evidence for Dark Matter

The first attempt at measuring the mass density in the universe was made by Hubble (1926). Hubble made dynamical estimates of galaxy mass from observations of stars and gas within galaxies. Zwicky (1933) was the first to discover a mismatch between visible matter and the mass of clusters determined by dynamical estimates. Zwicky estimated the mass of the Coma cluster of galaxies by measuring the velocity dispersion of galaxies in the cluster. He then used a virial theorem argument to estimate the mass. Zwicky then compared this with the mass-to-light ratio of galaxies within the cluster obtained from the rotation curve, finding that most of the mass of the cluster was not directly observed in the galaxies. Smith (1936) repeated the process for the Virgo cluster, obtaining similar results.

Although the existence of dark matter was known from the 1930's, dark matter did not become a hot topic in cosmology until the early 1970's. The mass density of the universe was regarded as just another parameter awaiting measurement. Despite this the implications for cosmology of a neutrino with a non-zero mass were recognized from the mid 1960's.

The next burst of activity on dark matter arose from improved observations of galaxy rotation curves. Observations of the outer regions of galactic disks were now possible. These observations did not show the expected *Keplerian region* where the speed of rotation of the disk would fall off as $R^{-1/2}$. This would indicating that the potential felt by objects was tending to that of a point mass, as the edge of the galaxy's mass distribution was reached. Instead what was observed was a flat rotation curve as far as could be observed, indicating that the edge of the galaxy had not been reached. Given that the luminosity was tailing off rapidly at large distances from the galactic centre, most of this mass component was "dark" (see for example (Rubin and Ford, 1970)).

The final significant push to the notion of dark matter came from theoretical quarters, with *inflation* (Guth, 1981)[7].

2.8 Inflation

Advances in particle physics left cosmologists with a problem. The grand unified theory [8] phase transition associated with the symmetry breaking

$$GUT \rightarrow SU(3)_{colour} \otimes [SU(2) \otimes U(1)]_{electro-weak} \tag{2.2}$$

produces mono-poles. These mono-poles would *over-close* the universe if not dealt with. Alan Guth (1981) realized that a period of rapid expansion in the early universe would get rid of the unobserved mono-poles. The period of de Sitter-like expansion became known as *inflation*.

Inflation also produced answers to a number of other problems in cosmology. The *Horizon* problem arises out of the observed homogeneity of the universe, the CMBR in particular. How can regions, which according to the Friedmann model would not have been causally connected at the time of decoupling, produce such a uniform temperature for the microwave background. Inflation solves this problem as the whole of the present horizon would have come from a single small causally connected region before the period of exponential expansion.

A second problem in the standard cosmology is the *flatness* problem. In the Friedmann framework if the density parameter, Ω_0, has a value other than unity it is rapidly driven away from unity. For Ω_0 to take the range of observed values

$$0.1 \leq \Omega_0 \leq 1.0 \tag{2.3}$$

requires that at the Planck scale

$$0 \leq 1 - \Omega_{Pl} \leq 10^{-60}. \tag{2.4}$$

Hence we require fine-tuning of the initial conditions to give a universe anything like that we observe. This is solved by inflationary models as the period of inflation drives the density parameter rapidly towards unity. If, as is expected, more inflation occurs than is absolutely necessary to solve the mono-pole and horizon problems, then we should expect today that

$$\Omega_0 = 1. \tag{2.5}$$

The final problem solved by inflationary models is the origin of the fluctuations that led structure to form. Inflation models naturally produce a near Zel'dovich spectrum of fluctuations. These arise out of quantum fluctuations in the scalar potential that drives inflation (Hawking, 1982).

The original inflation model (Guth, 1981) was subsequently discovered to suffer from the *graceful exit problem* in that the inflation never ended (Guth and Weinberg, 1981). The *new inflation* model was then proposed where the inflation occurs after quantum tunneling and not before (Linde, 1982), (Albrecht and Steinhardt, 1982). Further models such as *chaotic inflation* followed (Linde, 1983) soon after.

[7] see 2.8

[8] GUT

2.9 The Nature of the Dark Matter

The nature of the dark matter has been central to debate in cosmology since the mid 1970's (see §2.7). *Inflation* led many cosmologists to the conclusion that we are living in a critical density universe. It is clear from *primordial nucleosynthesis* (§2.6) that most of the mass in the universe cannot be in baryonic form if the universe has a mass density in the observed range of $0.3 \leq \Omega_0 \leq 1.0$. Dark matter is usually categorized into "Hot"[9] and "Cold"[10], depending on its thermal velocity. What then is dark matter made of?

The obvious HDM candidate is a neutrino with mass. Gershtein and Zel'dovich recognized that relic neutrinos with mass would have cosmological consequences. The number density of relic neutrinos is close to that of microwave background photons, which places an upper limit on the mass of a few eV's. Interest in neutrinos as a dark matter candidate peaked following evidence for neutrino mass from a beta-decay experiment in the early 1980's. The result was not repeated and interest waned. Free streaming of HDM wipes out adiabatic fluctuations in small scales (Bond et al., 1980), and pure HDM models do not fit the observed shape of the power spectrum. However, evidence is mounting from particle physics that neutrinos do indeed have mass. There is considerable evidence for neutrino oscillations (Solar neutrino deficit etc.) which require at least one type of neutrino to have a non-zero rest mass. However, the evidence from these experiments points to a relatively small mass, which is insignificant for cosmology.

There are many different CDM candidates from particle physics, Axions and Super-symmetric WIMPS are probably the best two. Axions are associated with the Pecci-Quinn symmetry which provides a solution to the strong CP problem. Super-symmetric theories, which provide solutions to many problems in the standard model of particle physics, provide a massive stable neutral particle (hence a *Weakly Interacting Massive Particle*). Laboratory experiments are underway to detect CDM particles, to date none have been found.

Baryonic dark matter, although it cannot form the bulk of the dark matter, must still be present. Early evidence from the MACHO and EROS micro-lensing experiments suggest a significant amount of the milky way's halo is in the form of dark compact objects of around $0.5M_{\odot}$. However, more recent work has suggested that less than 20% of the halo is in the form of MACHOS.

2.10 COBE and 90's Cosmology

At the beginning of the 1990's the standard cold dark matter model (CDM) [11] seemed to be in reasonable agreement with observations. All this changed as data became available from the COBE[12] satellite mission. The great success of COBE was the discovery of fluctuations in the microwave background (see for example (Wright et al., 1992)). The COBE result also allowed the power spectrum of fluctuations to be normalized at the horizon scale.

Once CDM models had been normalized to the COBE amplitude it became apparent that CDM produced too much power on cluster scales (1993). Very soon it became apparent that there were other defects too. The shape of the CDM power spectrum is wrong (Baugh and Efstathiou, 1994) and N-body simulations show dark matter density profiles $\rho(r) \sim r^{-1}$ in halos (Dubinski and Carlberg, 1991) which is in conflict with observations [13].

Following the COBE result two CDM variants have come to the fore, MDM and ΛCDM. The first is a CDM model with a hot dark matter component. The warm dark matter reduces the power on cluster scales, although these models are quite sensitive to the number of massive neutrino species and the amount of hot dark matter (see §2.9). The ΛCDM models have $\Omega_0 + \Omega_\Lambda = 1$ with $\Omega_0 = 0.3$.

[9] HDM
[10] CDM
[11] $\Omega_0 = 1, h \approx 0.5, n = 1$ Zel'dovich spectrum of initial fluctuations
[12] Cosmic Background Explorer
[13] Although it is not entirely clear whether this a problem with the simulations or the CDM model

In the next few years observational programs should be able to determine the nature of the dark matter and tie down the cosmological model further. The small angle microwave background satellite experiments, MAP (NASA) and Planck (ESA) should be able to determine the nature of the dark matter. Improved simulation techniques with high-resolution hydrodynamics included with the dark matter particles should be able to answer some of the questions surrounding the formation and evolution of galaxies. The next generation of red-shift surveys (e.g. 2° field) should provide much better measurements of the power spectrum.

2.11 N-body simulations in Cosmology and Astrophysics

N-body simulations have been used as a tool in astrophysics since the early 1960's, although most of the early technical developments were led by other applications of particle methods. Abernathy and Kronauer (1962) used *Particle-Particle*[14] simulations in work on vorticity in fluids. A *Particle-Mesh* code[15] was used by Levy and Hockney (1968) in simulations of electron flow in magnetic fields. Later the *Particle-Particle / Particle-Mesh* code[16] was applied to molecular dynamics (Hockney et al., 1973) before being later applied in cosmology.

The first application of N-body methods in astrophysics was in the simulation of star clusters. Aarseth and others used PP codes to simulate small star clusters. Clearly if star clusters could be simulated then so could galaxy clusters and Aarseth produced a series of papers, (1963) (1966) (1969). In the 1970's more simulations of galaxy clustering were performed using PP simulations, by Peebles (1970), Groth *et al.* (1977) and others. Aarseth, Gott and Turner (1979) used PP simulations with 4000 particles, which pushed the technique to its limits. To obtain higher mass resolution, other methods would need to be used. Efstathiou and Eastwood (1981) used a P^3M code to produce galaxy clustering simulations with 20,000 particles.

Particle simulation methods were also applied to the simulation of galaxies. PP methods were used by Lindblad (1960b), (1960a) and Ostriker and Peebles (1973). Later two and three-dimensional PM codes were used, with Hockney and Brownrigg (1974) producing 32x32x16 grid PM simulations of spiral galaxies.

The 1980's saw extensive use of PM (Klypin and Shandarin, 1983) and P^3M (Efstathiou et al., 1985) simulations in cosmology. Over time increasingly large numbers of particles were used to simulate largely CDM models of large-scale structure. At the start of the 1980's simulations with $O(10^4)$ particles was the norm, by the end of the decade $O(10^6)$ particles was the norm. In 1986 a new type of N-body code was developed, the TREE code[17] of Barnes and Hut (1986). The tree code can be thought of as an extension to the PP method, with the forces between widely separated particles only approximately calculated. The TREE code found application in galaxy simulations and galaxy collision simulations; it has also been used for cosmological simulations but not widely.

Many new types of N-body code were developed in the 1990's as many cosmologist sought greater dynamic range for their simulations. Adaptive grids have been used in many codes, with adaptive particle mesh codes by Villumsen (1989), Suisalu and Saar (1995a), Kravtsov, Klypin and Khokhlov (1997) and MLAPM, and then adaptive p^3m code of Couchman (1991). Attempts were also made to include gas hydrodynamics into cosmological simulations. Two approaches have been tried, the Eulerian grid approach of Cen and Ostriker[18], and the Lagrangian *Smoothed Particle Hydrodynamics* approach used by several different research groups[19].

New massively parallel supercomputers became available to researchers during the 1990's with the Cray T3D and T3E machines using the DEC Alpha processor, and later other similar machines from major computer companies worldwide. These resources led to the formation of large simulation consortia with a NASA grand challenge project and the Virgo consortium in the

[14] See §3.3.1.
[15] See §3.3.2.
[16] See §3.3.3.
[17] See §3.3.4.
[18] See for example (Cen, 1992)
[19] An example would be (Navarro and White, 1993)

UK. Both these groups sought to run large scale simulations including gas hydrodynamics.

Computer simulations have advanced much from their early origins as simple PP codes with a few hundred particles. However, it also fair to say that even current simulations lack the dynamic range needed, to be able to answer all the questions about both structure formation and galaxy formation that cosmologists pose them. Although, with the ever rapid advance of computer technology, the day when they do have the required dynamic range, is not many years away. Gas hydrodynamics on the other hand, may take some time to perfect as many of the processes the researcher tries to capture, such as star formation, are uncertain. Of course, the very development process itself will lead to greater understanding.

Chapter 3

Modelling the evolution of Large Scale Structure in the Universe

3.1 Large Scale Structure

The study of the large scale structure[1] of the universe is the study of the "late" evolution of the Universe. By "late" I refer the epoch in which gravity alone is responsible for the growth of structure on all but the small scale. Models of the LSS should match the distribution of mass inferred from the distribution of galaxies in the sky [2]. Hence the models are tested by their fit to observations. In doing so they also test the initial conditions provided by theories of the early universe..

3.1.1 Modelling Gravitational Evolution

The basis of all cosmology is Einstein's General relativity. The Friedmann-Robertson-Walker metric provides a homogeneous and isotropic expanding universe model. This cannot be the whole story as we know that the universe is not homogeneous on small scales or we would no structure in the universe at all. However, we know that from observations of the cosmic microwave background that on very large scales departures from homogeneity are of the order of 10^{-5}.

Given the weak gravitational fields we can model the structure using linearized General Relativity. Using the FRW metric as a background, we find that the scalar perturbation to the Einstein tensor is given by

$$\delta G_{00} = 2\Delta\phi \tag{3.1}$$

where Δ is the Laplacian. The Einstein equations yield

$$\nabla^2\phi = 4\pi G\rho(\mathbf{r}, t) \tag{3.2}$$

hence the gravitational evolution is governed by the Newtonian Poisson equation. To proceed the integral

$$\phi(\mathbf{x}) = \int \frac{\rho(\mathbf{x}')}{|\mathbf{x} - \mathbf{x}'|} d^3\mathbf{x} \tag{3.3}$$

must be evaluated either analytically or numerically.

Analytic solutions of equation (3.3) can be found as long as the density fluctuations are small,

$$\frac{\delta\rho}{\rho} < 1 \tag{3.4}$$

[1] hereafter LSS

[2] Of course how one goes about "inferring" a mass distribution from a galaxy distribution is itself of great debate

Once $\delta\rho/\rho \sim 1$ the (still analytic) Zel'dovich approximation must be used. In the "non-linear" regime $\delta\rho/\rho > 1$ the integral must be solved numerically by Monte-Carlo sampling. This type of Monte-Carlo sampling is in fact an N-body simulation.

3.2 The N-body Simulation

The N-body simulation as applied to cosmology models the gravitational evolution of the universe. The mass density of a region of the universe is represented by a distribution of equal mass particles. Typically a set of particles is displaced from a lattice to represent the density fluctuations. After the initial conditions have been set the particles are allowed to interact gravitationally and the distribution is evolved forwards in time through a series of discrete time-steps. A variety of techniques have been used to calculate the forces between particles (see §3.3)

Whatever the techniques used, the equations of motion are the same. I will derive these in §3.2.1. In the remainder of this section I will discuss the conservation of energy through the Layzer-Irvine equation (§3.2.3) and the boundary conditions on the simulation (§3.2.4).

3.2.1 The Equation of Motion

From equation (3.2) we see that the Newtonian Poisson equation governs the motion of matter in the universe. Hence the Newtonian force law can be used to calculate the forces on each particle [3],

$$\dot{\mathbf{v}} = -\nabla\phi, \tag{3.5}$$

where $\mathbf{v} = \dot{\mathbf{r}}$.

To proceed further it is useful to introduce the concept of "co-moving coordinates". This coordinate system is one in which the expansion of the universe is factored out.

$$\mathbf{r}'(t) = \mathbf{r}/a(t) \tag{3.6}$$

where $a(t)$ is the scale factor and $a(0) = 1$. Substituting this into (3.5) I obtain

$$\dot{\mathbf{v}}' + 2\mathbf{v}'\frac{\dot{a}}{a} = -\frac{1}{a}\left(\nabla\phi + \mathbf{r}'\ddot{a}\right), \tag{3.7}$$

where I have defined $\mathbf{v}' = \dot{\mathbf{r}}'$. $\dot{a}$ may be obtained from the Friedmann equation,

$$\dot{a}^2 = \frac{8\pi G\rho_0}{3}\left(\frac{1}{\Omega_i} - 1 + \frac{1}{a}\right) \tag{3.8}$$

where ρ_0 is the mean mass density, and Ω_i is the density parameter at $a = 1$ (a derivation of the Friedmann equation may be found in appendix A).

At this stage it is useful to introduce a co-moving gravitational potential

$$\phi' = a\phi + \frac{a^2\ddot{a}}{2}\mathbf{r}'^2 \tag{3.9}$$

Using $\nabla = a^{-1}\nabla'$ I find that $\nabla'\phi'$ is given by

$$a^2\nabla\phi + \mathbf{r}'a^2\ddot{a}, \tag{3.10}$$

which when substituted in (3.7) gives

$$\dot{\mathbf{v}}' + 2\mathbf{v}'\frac{\dot{a}}{a} = -\frac{1}{a^3}\nabla'\phi. \tag{3.11}$$

[3] I have here and subsequently made the assumption that the mass of each particle is one. In cosmological simulations all the particles usually have the same mass and setting that mass to unity reduces the computational cost.

The co-moving Poisson equation may now be obtained

$$\begin{aligned} \nabla'^2 \phi' &= a^3 \nabla^2 \phi + 3a^2 \ddot{a} && (3.12) \\ &= 4\pi G(\rho'(\mathbf{r}', t) - \bar{\rho}'(0)), && (3.13) \end{aligned}$$

where $\rho' = a^3 \rho$ is the density in the co-moving frame and I have used the Raychauduri equation to eliminate $\ddot{a}$

$$\ddot{a} = -\frac{4\pi G \bar{\rho}(0)}{3a^2}. \tag{3.14}$$

(See appendix A for a derivation of the Raychauduri equation)

3.2.2 Integrating the Equations of Motion

The Newtonian equations of motion may be integrated in their present form, however it is more convenient to change the time variable to be a power of the scale factor $p = a^\alpha$ (Efstathiou et al., 1985). With this change of variable, equation (3.11) becomes

$$\frac{d\mathbf{u}}{dp} + \frac{(1 + \alpha + \ddot{a}a/\dot{a}^2)}{\alpha a^\alpha}\mathbf{u} = -\frac{\nabla\phi}{\alpha^2 \dot{a}^2 a^{2\alpha+1}}, \tag{3.15}$$

where

$$\mathbf{u} = d\mathbf{r}'/dp. \tag{3.16}$$

To integrate the equations of motion numerically for each particle the derivatives in equations (3.15) and (3.16) must be replaced by difference equations. For example at the nth timestep we replace $d\mathbf{r}'/dp$ by the approximation

$$\frac{\mathbf{r}'_{n+1} - \mathbf{r}'_n}{\Delta p} = \mathbf{u}_{n+1/2} \tag{3.17}$$

dt is approximated by Δp a finite time interval or timestep. The size of the timestep governs the accuracy of the integration. A suitable timestep size is chosen with reference to a range of criteria based on the characteristic time scales of the problem.

Using this discrete approximation I obtain the time-centred leapfrog integration scheme,

$$\begin{aligned} \mathbf{u}_{n+1/2} &= \frac{(1 - A_n \Delta p)\mathbf{u}_{n-1/2} + B_n \mathbf{F}_n \Delta p}{1 + A_n \Delta p} && (3.18) \\ \mathbf{r}'_{n+1} &= \mathbf{r}'_n + \mathbf{u}_{n+1/2} \Delta p && (3.19) \end{aligned}$$

where A_n and B_n are given by the expressions

$$\begin{aligned} A_n &= A(p_n) = \frac{(1 + \alpha + \ddot{a}a/\dot{a}^2)}{2\alpha a^\alpha}, && (3.20) \\ B_n &= B(p_n) = \frac{1}{\alpha^2 \dot{a}^2 a^{2\alpha+1}}. && (3.21) \end{aligned}$$

evaluated at timestep n. The leapfrog scheme is symplectic and the errors are of order (Δp^3) in position and velocity. A higher order scheme could be used but would require the particle accelerations to be stored, a considerable additional memory requirement. Particle number dictates the overall accuracy of the simulation and so the simple leapfrog scheme is usually used because of its lower storage requirements. The symplectic nature of leapfrog is very desirable for its energy conserving properties.

3.2.3 The Layzer-Irvine Cosmic Energy equation

Conservation of energy for a particle distribution in co-moving coordinates is expressed by the Layzer-Irvine energy equation,

$$\frac{d(a^4T)}{dt} + a\frac{dU}{dt} = 0 \tag{3.22}$$

(see (Layzer, 1963) and (Peebles, 1980)). The kinetic and potential energy of the particle distribution is given by

$$T = \frac{1}{2}\sum_i v_i^2 \tag{3.23}$$

$$U = \frac{1}{2}\sum_i \phi_i, \tag{3.24}$$

where ϕ_i is the co-moving gravitational potential at the location of the ith particle. Equation (3.22) may be integrated to give

$$a^4T + aU - \int U da = C, \tag{3.25}$$

which, in discrete form, becomes

$$(a^4T)_m + (aU)_m - \sum_{n=1}^{m} \Delta p U_n = C. \tag{3.26}$$

The degree to which the integration satisfies equation (3.26) provides a check on its accuracy.

3.2.4 Boundary Conditions

A cosmological simulation can only simulate a finite volume of the universe. To represent the rest of the universe a set of appropriate boundary conditions for the computational box must be chosen. The natural boundary conditions are triply-periodic. As we have already discussed all the models of the universe are homogeneous and isotropic on large scales, and periodic boundary conditions are an easy way to satisfy this requirement. Triply periodic boundary conditions also ensure the mean mass density in the computational box remains constant as particles that leave one side of the box re-enter at the opposite face. The importance of the $-\bar{\rho}'(0)$ term in the co-moving Poisson equation (3.13) is clear as there is no solution to Poisson's equation with triply periodic boundary conditions unless the average density in the computational box is zero. Other sets of boundary conditions have been used in cosmological simulations in cases where the type of code used has made triply-periodic boundary conditions difficult to implement. Care should also be taken with smaller scale simulations where the assumptions of isotropy and homogeneity may cease to be appropriate as the simulation progresses.

3.3 N-body techniques

N-body simulation codes come in many different types as discussed above. Code types are distinguished by their approach to the force calculation. Most of the codes I will discuss were not designed specifically for cosmology or even astrophysics and were developed from codes used to model plasmas or other particle based systems. Many of the codes were first applied to the study of star clusters and other dynamical problems in Astrophysics before being adapted for use in cosmology. Each approach has its own advantages and disadvantages in the context of a cosmological simulation, which the following discussion will address.

3.3.1 The Particle-Particle (PP) Method

Earlier we saw that the force law was Newtonian albeit with a modified equation of motion. The simplest method of calculating the force between each particle pair in a distribution of N particles is to use the "Coulombic" force law, where

$$\mathbf{F}_{ij} = G\frac{(\mathbf{r}_i - \mathbf{r}_j)}{|\mathbf{r}_i - \mathbf{r}_j|^3} \tag{3.27}$$

gives the force between the ith and jth particles. The net force on each particle is given by the vector sum of the $(N-1)$ pair-forces acting on that particle. The equations of motion are then integrated in the usual way.

The particle-particle[4] method is simple and easy to program, while the accuracy of the force calculation is limited only by level of accuracy of the computer being used. The PP method can easily be adapted for use with individual timesteps. An integration scheme with individual timesteps allows each particle to have a individually "tailored" timestep[5]. The size of timestep is determined by a range of criteria based on physical time scales. Particles in high density regions need to have a smaller timestep than those in low density regions. Maximum efficiency can be achieved by moving the particles which require a small timestep more often than those that do not.

PP techniques are now rarely used to perform cosmological simulations as they have a much higher operations count per particle than other methods. As can be seen by analyzing the force calculation above, the operations count is proportional to (N - 1) per particle. Hence the operations count scales like

$$\text{Op. count} \propto N(N-1) \tag{3.28}$$

with particle number. The number of particles is limited to O(10^4) on current workstations, far too few to adequately sample the density field. Special purpose hardware (the GRAPE machine) enables larger numbers of particles to be used. The second major problem with the PP technique is the lack of naturally periodic boundary conditions. Periodicity is best imposed through the use of Ewald summation (see both §8.1.1 and (Sangster and Dixon, 1976)).

3.3.2 The Particle-Mesh (PM) Method

Particle-Mesh[6] codes allow many more particles to be used than PP codes do at the cost of limited force resolution. Today 10^6-10^7 particles can be used on a UNIX workstation, limited usually by the availability of computer memory. This increase in particle number arises from a more favourable scaling of the operations count with particle number than PP codes have (see below). MLAPM is ultimately a PM-type code and I will now describe the approach in some detail.

Introducing the Grid Approach

Recall the Poisson equation (3.13) in co-moving coordinates. Instead of solving for the forces between particles directly we could solve the Poisson equation to obtain the potential field associated with the underlying mass distribution. The gradient of the potential field would then give a force vector-field. The force field could then be assigned to the particles. This is the approach PM codes use in their force calculation.

To solve the Poisson equation numerically, the density distribution, the potential field and the force vector-field are represented on a regular[7] three dimensional lattice, usually referred to as a *grid* or *mesh* (hence particle-mesh code). Triply-periodic boundary conditions are usually used. The standard potential solver used is the *Fast Fourier transform* this automatically imposes triply periodic boundary conditions.

[4] Hereafter PP
[5] I will return to the use of multiple timesteps in chapter 6
[6] hereafter PM
[7] see the discussion of the adaptive nature of MLAPM in chapters 5 and 7.

To obtain a density distribution, mass from each particle must be assigned to the surrounding grid-points by an assignment scheme. Similarly forces must ultimately be interpolated back to the particles from the grid. Differential operators such as ∇^2 and ∇ must be discretized and replaced with difference operators. Once the forces are obtained the code proceeds in the usual way.

The Timestep Loop & Operations Count

A single timestep proceeds as follows:

1. Assign mass to grid to produce density field.
2. Solve Poisson's equation on the grid to give gravitational potential.
3. Difference potential to obtain forces on the grid.
4. Interpolate the grid force field to the particles.

The first and last step are particle operations and have computational costs that are linearly proportional to the number of particles, N_p. This is a big improvement on the quadratic scaling of the PP method. The second and third steps are grid operations which have a scaling of $N_g log N_g$ or better, where N_g is the number of grid points. Hence the operations count is given by

$$\text{Op. Count} = \alpha N_p + \beta N_g log N_g \tag{3.29}$$

Which term dominates depends sensitively on the values of the coefficients, α and β and the ratio of N_p to N_g. Typically PM simulations are limited by memory, not CPU time.

Resolution Limitations

The major disadvantage of the PM-type code is limited, fixed resolution. Clearly the code cannot accurately calculate the forces between particles in the same grid cell. This can best be seen in terms of the reciprocal lattice. Components of the force with spatial frequencies greater than the Nyquist frequency of the lattice cannot be represented on it. Numerical experiments show that the "grid force" between two particles, falls below the true $1/r^2$ force law at separations of order two grid cell lengths depending on the mass and force assignment schemes chosen (Efstathiou et al., 1985) (MLAPM's force law is illustrated in figure 8.1). The grid will also introduce anisotropies in the forces across a grid cell. The level of these anisotropies is again determined by the mass and force assignment schemes.

The limited resolution should not be viewed as a total disadvantage. In a cosmological simulation each particle will typically represent say 10^9 solar masses of material, with a hundred such particles ultimately representing a spiral galaxy. It is only large groups of particles forming such objects that can be relied upon for statistical information. Hence highly accurate force calculations will waste CPU time calculating spurious two-body orbits. Having a "softer" force law avoids this.

A more important disadvantage is the fixed nature of the resolution. As a typical cosmological simulation evolves in time clustering of the particles increases, leaving large regions of the computational box with few particles and small volumes containing the majority of the particles. To achieve the resolution required at the end of a simulation a very fine grid must be used throughout. Typically for a simulation with 64^3 particles a grid with 128 grid points in each dimension will be used. Such a fine grid presents a considerable storage overhead. If each grid point contains a minimum of four floats (three for the force components and one for the potential and density) with each float occupying four bytes, a typical 256^3 grid would occupy

$$4 \times 4 \times 256^3 = 256\text{Mb}. \tag{3.30}$$

To improve the resolution of PM codes, Melott (1986) introduced a staggered grid, with the grids used for the potential and force calculations being offset by half a grid cell in each direction. This improved the resolution of the code at the expense of particles gaining spurious self-forces. I will discuss other attempts to improve upon the PM code in subsequent sections.

Mass Assignment and Force Interpolation Schemes

The particle distribution is converted to a density distribution on the grid through a mass assignment scheme, with particles assigning mass to the neighbouring grid points. There is a hierarchy of such schemes of increasing "smoothness", with particles being assigned to the nearest n grid points in each direction. Each scheme has a weighting function which can be expressed as the product of weighting functions for each dimension

$$W_{ijk}(\mathbf{x} - \mathbf{x}_{ijk}) = w_i^x w_j^y w_k^z \tag{3.31}$$

where $\mathbf{x}_{ijk}$ is the position vector of the grid point with indices (i,j,k) and $\mathbf{x}$ is the position vector of the particle.

The simplest scheme is the *Nearest Grid-point Scheme* (or NGP). This scheme assigns all the mass of a particle to the nearest grid point. Hence the weighting function is

$$w_i = 1 \qquad \forall \qquad |x - x_i| \leq L/2 \tag{3.32}$$

where L is the grid-spacing. This scheme gives discontinuous forces.

The next scheme is the *Cloud in Cell Scheme* (or CIC). This scheme spreads the particle across the two neighbouring grid points in each dimension. The particle is thus spread out as a "cloud". The weighting function is

$$w_i = 1 - \frac{|x - x_i|}{L} \qquad \forall \quad |x - x_i| \leq L \tag{3.33}$$

The resulting forces are continuous but with discontinuous first derivative.

The next highest order scheme is the *Triangular-shaped cloud* (or TSC). This is the scheme used by MLAPM (see 7.3.9). The particle is spread over three grid points in each dimension and the scheme's weighting function is

$$\begin{aligned} w_i &= \frac{3}{4} - \frac{|x - x_i|^2}{L^2} \qquad \forall \quad |x - x_i| \leq L/2 \\ w_i &= \frac{1}{2}\left(\frac{3}{2} - \frac{|x - x_i|}{L}\right)^2 \qquad \forall \quad L/2 \leq |x - x_i| \leq 3L/2 \end{aligned}$$

The forces and there first derivatives are continuous but all higher-order derivatives are discontinuous.

Higher order assignment schemes are possible but rapidly become extremely computationally expensive and in practice only the CIC and TSC schemes are used. The hierarchy of schemes are increasingly smooth with "smoothness" defined in terms of the continuity of the density field and its derivatives. However such smoothness results in a degradation of resolution as the effect of the higher-order schemes is to spread out each particle into a "cloud".

To ensure the conservation of linear momentum it can be shown (see for example (Hockney and Eastwood, 1988)) that force interpolation must use the same weighting scheme as mass assignment. If this approach is not chosen unphysical self-forces are the result. Hence in most PM-type codes the above comments on the continuity of the density field apply equally to the derived force field. The CIC and TSC schemes are typically those used as they give the best combination of resolution, smoothness of the forces and low computational cost. Illustrations of the MLAPM force-law and force anisotropies can be found in chapter 8.

Solving Poisson's Equation

There are many numerical techniques for solving Poisson's equation. However, because of the requirement of triply-periodic boundary conditions, PM codes usually use a method involving fast Fourier transforms (see (Press et al., 1992) for a discussion of FFT).

Recall the Poisson equation

$$\nabla^2 \phi = 4\pi G \rho(\mathbf{x}) \tag{3.34}$$

this has a solution

$$\phi(\mathbf{x}) = \int_{-\infty}^{\infty} G(\mathbf{x}')\rho(\mathbf{x}')d\mathbf{x}', \tag{3.35}$$

where $G(\mathbf{x})$ is the Green's function. This is of course a convolution, so in Fourier space

$$\hat{\phi}(\mathbf{k}) = \hat{G}(\mathbf{k})\hat{\rho}(\mathbf{k}) \tag{3.36}$$

That is Poisson's equation may be solved by Fourier transforming the density field and multiplying it with the Fourier-space Green's function. The real-space potential is obtained by inverse Fourier transform. The real-space Green's function used is not usually the "continuous" Green's function for Poisson's equation

$$G(\mathbf{x}) = 1/|\mathbf{x}| \tag{3.37}$$

The Green's function is usually "shaped" to help eliminate force anisotropies.

3.3.3 The Particle-Particle Particle-Mesh (P^3M) Method

The particle-particle particle-mesh method[8] is a hybrid of the PM and PP methods. P^3M attempts to overcome the limited resolution of PM codes by introducing a short-ranged PP step. The overall force law is split into two components, a long ranged one calculated by a PM code, and a short ranged one calculated by direct summation. The range of the PP component is limited to a radius r_s around each particle, all particles within r_s of a given particle contribute a "PP-force" in addition to the "PM-force" contributed by all particles.

The PM force calculation is carried out using the FFT technique discussed above using an appropriate mass assignment scheme. The Green's function is usually modified so that the "join" between the two force components is smooth. The PP force is also modified so that it falls off to zero at radius r_s from a given particle. The PP component also has a "softening length" η typically 0.3 of a grid cell length. The softened force again limits the resolution of the calculation but is introduced to avoid the spurious two-body effects discussed above.

The Operations Count

The addition of the PP step adds an additional term to equation (3.29). This equation becomes

$$\text{Op. Count} = \alpha N_p + \beta N_g log N_g + \gamma N_n N_p \tag{3.38}$$

where N_n represents the average number of nearest neighbours. The major weakness of the P^3M approach is that N_n increases as the simulation proceeds, degrading the codes' performance. This results from the appearance of clustered regions with many particles within length r_s, giving many PP calculations and slowing the force calculation considerably.

P^3M represents a significant advance over PM codes. P^3M codes were first applied to cosmology by (Efstathiou and Eastwood, 1981). P^3M codes were successfully applied to cosmology throughout the 1980's (Efstathiou et al., 1985) . However, P^3M has now been superceeded by the more efficient *adaptive* codes.

Introducing the Linked-List

It is worth mentioning the method by which P^3M codes identify which particles need to be included in the PP calculation. This is done by using a slightly lower resolution grid than that used for the PM calculation called a *chaining-mesh* and a series of *linked-lists*. The chaining-mesh is a three-dimensional array of pointers, each pointing to the first particle within that mesh cell with all the particles x-sorted. Each particle then has a pointer to the next particle in the list and so on until the last particle in the list which has a `NULL` pointer.

MLAPM uses a similar scheme for particle organization although no chaining-mesh is used.

[8] usually known as P^3M

3.3.4 The TREE Code

The TREE code was developed principally for galactic simulation by Barnes and Hut in 1986 (Barnes and Hut, 1986). It has subsequently been used to perform cosmological simulations. TREE codes are much more computationally expensive than PM codes but have significant advantages as they are naturally implemented on parallel computers.

The TREE code is perhaps the most elegant of particle codes. The TREE code is basically a modified PP code. The operations count is limited by grouping distant mass concentrations into a single particle for force calculation purposes. Particles are so grouped by building a "tree" of nested grids. The whole calculation is recursive in nature.

Like the PP code the TREE code does not have naturally periodic boundary conditions. Triply periodic boundary conditions are best imposed through the used of Ewald summation (see both §8.1.1 and (Sangster and Dixon, 1976)). Other approaches have been used, (Warren and Salmon, 1992) used vacuum boundary conditions and then discarded the outer 30% of the particles.

3.3.5 APM and AP^3M

APM and AP^3M are *adaptive* grid codes introduced by Villumsen (1989) and Couchman (1991) respectively. APM may be viewed as an alternative approach to P^3M to improve the resolution of PM type codes while AP^3M may be viewed as an attempt to reduce the operations count of P^3M.

Introducing Adaptive Grids

Recall the discussion of PM-type codes above. The resolution of a PM code is limited because the grid is a fixed size and stays so throughout the simulation. At the start of a simulation the resolution of a PM code is fine as the particles are widely spaced. However at the end of a simulation much of the computational box is over-resolved with few particles in *void*-like regions and a few regions of the box are very under-resolved with large concentrations of particles in a small volume.

This resolution problem could be overcome by using a grid which has fine resolution where required and coarse resolution where there are few particles. As the particle distribution changes during the simulation this grid will need to *adapt*[9] accordingly. Although a simple extension in principle, the use of an adaptive grid presents many practical difficulties.

There are essentially two approaches one can take to the adaptive grid. One can use a series of nested cubical or cuboidal grids which makes the data structure simple and easy to program. However, the adaptive sub-grids must be placed appropriately and this requires a complex algorithm to find the most suitable size and location for the refinement. This first approach is the one taken by most adaptive particle codes, including those of Villumsen and Couchman.

A second and potentially much more efficient approach is to use a grid that is more like a "swiss cheese", with grid points only where the density of particles is high enough to warrant refinement[10]. This approach requires only a simple placement algorithm and potentially uses a minimum of computer memory. However, this approach requires a complex data structure to support such flexible placement of grid points. Complex data structures often require the use of high level programming languages such as C and C++ to make their use practical. The use of a complex data structure also has implications for debugging and program development. A flexible grid also requires the use of alternative techniques to solve Poisson's equation, Fourier transform techniques cannot be used. MLAPM uses this latter approach: the adaptive multi-grid algorithm it uses as its potential solver, can be used on any shape of grid (see chapters 4 and 5).

[9] Early attempts to improve the resolution of PM codes have not been truly adaptive but have re-simulated at higher resolution areas of interest in a previous low resolution simulation. An example of this using a TREE code after a PM code can be found in (Dutta, 1993)

[10] An alternative refinement indicator may be more appropriate, such as a measure of the truncation error in the gravitational potential (see chapter 4)

Chapter 4

Multi-grid Methods

MLAPM uses a Poisson solver based upon the adaptive multi-grid "algorithm" of Brandt (1977a). I say "algorithm" as no single multi-grid algorithm exists, it is rather an approach to solving elliptic boundary value problems. In this chapter I will describe multi-grid methods in detail.

The multi-grid approach is central to what I have tried to achieve in MLAPM. Without the capabilities of adaptive multi-grid, the flexibility that MLAPM shows simply would not be possible.

4.1 Iterative Methods

One approach to multi-grid methods is to consider the traditional *relaxation* methods for solving boundary-value problems, Jacobi and Gauss-Seidel iteration. To consider the convergence properties of these techniques it is useful to write the finite-difference operators in matrix form.

Recall the 2D Finite-difference approximation to the Poisson equation

$$\frac{\phi_{i-1,j} - 2\phi_{i,j} + \phi_{i+1,j}}{h^2} + \frac{\phi_{i,j-1} - 2\phi_{i,j} + \phi_{i,j+1}}{h^2} = \rho_{i,j}, \tag{4.1}$$

which with some rearrangement becomes

$$\phi_{i-1,j} + \phi_{i+1,j} + \phi_{i,j-1} + \phi_{i,j+1} - 4\phi_{i,j} = h^2\rho_{i,j}. \tag{4.2}$$

The 2D array $\phi_{i,j}$ can be re-written as a vector by using a single index defined by

$$r \equiv j(I+1) + i \quad \text{with} \quad j = 0, 1, ...J \quad \text{and} \quad i = 0, 1, ...I. \tag{4.3}$$

Hence equation (4.2) becomes

$$\phi_{r-1} + \phi_{r+1} + \phi_{r-I-1} + \phi_{r+I+1} - 4\phi_r = h^2\rho_r. \tag{4.4}$$

The finite-difference equation may now be written in matrix form

$$\mathbf{A}.\phi = \mathbf{b}, \tag{4.5}$$

where $\mathbf{b}$ is a constant vector containing the information from the boundary points. The matrix $\mathbf{A}$ is a sparse matrix and can be readily split into two parts. Relaxation methods take advantage of this.

The matrix $\mathbf{A}$ is split into two parts

$$\mathbf{A} = \mathbf{E} + \mathbf{R}, \tag{4.6}$$

where $\mathbf{E}$ is invertible. The equation (4.5) becomes

$$\mathbf{E}.\phi = -\mathbf{R}.\phi + \mathbf{b} \tag{4.7}$$

$$= -(\mathbf{A} - \mathbf{E}).\phi + \mathbf{b}. \tag{4.8}$$

Pre-multiply (4.7) by the inverse matrix $\mathbf{E}^{-1}$ to obtain an expression for ϕ

$$\begin{aligned} \phi &= -\mathbf{E}^{-1}(\mathbf{A}-\mathbf{E}).\phi + \mathbf{E}^{-1}\mathbf{b} && (4.9)\\ &= \mathbf{M}.\phi + \mathbf{E}^{-1}\mathbf{b}, && (4.10) \end{aligned}$$

where

$$\mathbf{M} = -\mathbf{E}^{-1}(\mathbf{A}-\mathbf{E}). \tag{4.11}$$

We can use this as the basis of an iterative process,

$$\phi^{n+1} = \mathbf{M}.\phi^n + \mathbf{E}^{-1}\mathbf{b}, \tag{4.12}$$

where n is the number of iterations.

4.1.1 Rate of Convergence

Define an error vector, $\mathbf{e}^n$,

$$\mathbf{e}^n = \phi^n - \tilde{\phi}, \tag{4.13}$$

where $\tilde{\phi}$ is the true solution. Using equation (4.12) this becomes

$$\mathbf{e}^{n+1} = \mathbf{M}\mathbf{e}^n, \tag{4.14}$$

which leads us to

$$\mathbf{e}^n = \mathbf{M}^n\mathbf{M}^{n-1}\dots\mathbf{M}^0\mathbf{e}^0, \tag{4.15}$$

where $\mathbf{M}^n$ is the iteration matrix at each iteration step. In the simple case that the matrix remains constant, this reduces to

$$\mathbf{e}^n = \mathbf{M}^n\mathbf{e}^0. \tag{4.16}$$

Define a norm of the error vector by

$$\|e\| = \sum_{i=1}^{N_g} e_i^2, \tag{4.17}$$

where e_i is the error in the ith mesh point.

Taking the norm of both sides of equation (4.16) gives

$$\frac{\|e^n\|}{\|e^0\|} \leq \|\mathbf{M}^n\|. \tag{4.18}$$

Hence for convergence to occur the iteration matrix must have a norm less than unity, so matrix $\mathbf{E}$ must be chosen to minimize $\|\mathbf{M}\|$. The norm of a matrix as defined above is given by the modulus of its largest eigenvalue. Unfortunately relaxation methods usually have iteration matrices with several eigenvalues close to unity and convergence is slow.

4.1.2 The Iterative Procedure in Practice

The matrix formalism used above is useful for the theory of convergence and clearly shows the distinction between different relaxation methods. However, for the purposes of computer programming it is more useful to express the new values of the solution at a given grid point in terms of the old values of the solution at that point and at surrounding grid points. The five-point difference approximation to the Poisson equation, equation (4.1), may be rearranged to give

$$\phi_{i,j} = \frac{1}{4}(\phi_{i-1,j} + \phi_{i+1,j} + \phi_{i,j-1} + \phi_{i,j+1} - h^2\rho_{i,j}). \tag{4.19}$$

This rearrangement may now be iterated, giving a new value on the left-hand side in terms of old values of the solution on the right hand side.

The simpler relaxation methods, Jacobi and Gauss-Seidel, may be distinguished by the order in which grid points are processed and the time in the iteration in which the values of the solution are updated. More complex methods, such as *Successive Over-relaxation (SOR)* [1] , update the value of the solution at a grid point with a linear combination of the value given by equation (4.19) and the old value of the solution at that grid point

$$\phi_{i,j}^{new} = \omega\phi_{i,j}^{*} + (1-\omega)\phi_{i,j}^{old}, \tag{4.20}$$

where $\phi_{i,j}^{*}$ is the value of $\phi_{i,j}$ obtained from equation (4.19).

4.1.3 The Jacobi Method

The Jacobi method uses a matrix, **E**, that is the diagonal part of **A**. For a model problem using the Poisson equation on an $N \times N$ grid with Dirichlet boundary conditions, the spectral radius is given by

$$\rho_S \simeq 1 - \frac{\pi^2}{2N^2}. \tag{4.21}$$

Given that N is usually large, the Jacobi method is not practical in most circumstances. The Jacobi method may be understood in the terms of equation (4.19), by using the result as the new value of the solution at (i, j) but not updating any of the points until all the new values have been obtained.

4.1.4 Gauss-Seidel

The Gauss-Seidel method uses a matrix, **E**, that is formed by the lower triangular elements of **A**. For the same model problem as used in the Jacobi case, the spectral radius is just the square of the spectral radius for the Jacobi case.

$$\begin{aligned} \rho_{GS} &= \rho_{Jacobi}^2 && (4.22) \\ &\simeq 1 - \frac{\pi^2}{N^2}. && (4.23) \end{aligned}$$

Equation (4.23) show a slight improvement in the convergence rate over the Jacobi method, but not a significant one.

Gauss-Seidel again uses the expression (4.19) to update the solution, but updates the values of the solution as they become available. The ordering of the *Gauss-Seidel Sweep*, or the order in which the grid points are processed, now becomes important. The ordering is usually chosen to match the problem. A typical choice in 2D is *red-black* or *chess board* ordering. The "red" points are first updated using the values of the solution at the surrounding "black" points and then vice-versa. There is no preferred orientation with this choice.

4.2 Multi-Grid

4.2.1 Multiple grids

Consider the 5×5 grid illustrated in figure 4.1. If we remove the grid points from even columns and rows we are left with a 3×3 grid, covering the same domain. We can do this with any grid with odd numbers of grid points in each dimension[2]. Given that coarser grids can be readily identified, can they be used in the solution process? Could the iterative methods of the previous section be accelerated by first solving on a coarse grid (which is easier to solve as it contains fewer grid points)

[1] I will not discuss *SOR* further, the reader will understand why later in this chapter

[2] or with any periodic grid with even numbers of grid points in each dimension

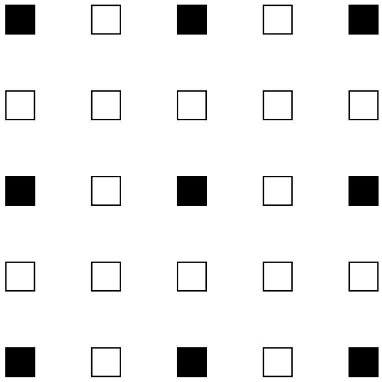


Figure 4.1: Eliminating the even columns and rows from this 5×5 grid gives a 3×3 grid.

and using this coarse grid solution as a first approximation on a finer grid? There is nothing new in this approach, schemes like this have been in use for many years and were applied even before computers were widely available. Brandt's multi-grid method goes beyond this approach by not just using the coarser grids to generate a first approximation to the fine grid solution but also in the process of improving the fine grid solution.

4.2.2 The Multi-Grid Method

Suppose we are trying to solve a linear differential problem of the form

$$L\Phi(x) = U(x) \text{ in } \Omega,\ \Lambda\Phi(x) = V(x) \text{ on the boundary } \partial\Omega., \tag{4.24}$$

on the domain Ω. We have a hierarchy of grids covering Ω, $G^0, G^1, \dots, G^M$ with corresponding grid spacings $h^0, h^1, \dots, h^M$[3]. The differential operators, L and Λ, are replaced be difference operators L^k and Λ^k acting on the grid G^k giving

$$L^k\Phi^k(x) = U^k(x) \text{ for } x \in G^k, \Lambda^k\Phi^k(x) = V^k(x) \text{ for } x \in \partial G^k. \tag{4.26}$$

We are trying to solve the difference equation to the accuracy of the finest grid, G^M. I noted above that one approach would be to solve equation (4.26) on each grid $G^0, G^1, \dots, G^M$ in turn, interpolating the solution obtained on one grid to the next to act as a first approximation for the relaxation method on the next. Multi-grid takes this one step further and all the grids are involved throughout the solution process. As we will see it is necessary to return to coarse grids after relaxation has been used on the finer grids.

If ϕ^M is an approximate solution of equation (4.26) then we may write

$$L^M\phi^M = U^M - u^M, \qquad \Lambda^M\phi^M = V^M - v^M, \tag{4.27}$$

where u^M and v^M are the *residuals*. Assuming that L and Λ are linear operators[4] and writing $\Phi^M = \phi^M + \psi^M$, where ψ^M is the *correction*, gives the *residual equations*,

$$L^M\psi^M = u^M, \qquad \Lambda^M\psi^M = v^M. \tag{4.28}$$

[3] MLAPM uses the ratio

$$\frac{h^k}{h^{k+1}} = 2 \tag{4.25}$$

but other choices are possible

[4] Non-linear multi-grid is possible, see for example (Press et al., 1992)

How may we best solve these equations for the "error"? Let us use an approximation in which we "coarsify" the equations and represent them on grid G^{M-1}.

$$L^{M-1}\psi^{M-1} = u^{M-1}, \qquad \Lambda^{M-1}\psi^{M-1} = v^{M-1}. \tag{4.29}$$

These equations are of smaller dimension and hence are easier to solve. To do this we must introduce a *restriction* operator $\mathcal{R}$,

$$u^{M-1} = \mathcal{R}u^M, \qquad v^{M-1} = \mathcal{R}v^M. \tag{4.30}$$

Once a solution to equation (4.29) has been obtained by whatever means a *prolongation* operator is required to give the corresponding fine-grid version of ψ^{M-1}

$$\psi^M = \mathcal{P}\psi^{M-1}. \tag{4.31}$$

The solution can now be updated to include the coarse-grid correction

$$\phi^M_{new} = \phi^M_{old} + \psi^M. \tag{4.32}$$

Of course the solution to equation (4.29) may itself be found by a similar approximation and so on, up to grid G^0 where solution by direct matrix inversion or other methods is computationally feasible.

Unfortunately expressing a problem defined on grid G^M on coarser grids has inherent dangers. Consider the residuals, u^M and v^M, in a Fourier sense. They will contain *smooth* components which vary only slowly across grid G^M and *non-smooth* components which vary on the length-scale of the grid itself. The non-smooth components of the residuals have frequencies which lie beyond the Nyquist frequency of grid G^{M-1} and cannot be represented on that grid. Consequently no coarse-grid approximation will ever reduce these components of the error. More dangerous is the problem of aliasing (Hockney and Eastwood, 1988) [5]. Ideally the non-smooth components of the error should be removed before using a coarse-grid correction technique. Interpolation of the correction will also lead to fluctuations in the residual on the scale of the grid. Hence smoothing should also take place after the coarse-grid correction.

Analysis of relaxation techniques has shown that for the first few iterations these methods produce rapid convergence. The speed of convergence then rapidly decreases. As I will show in the next section, Fourier analysis of Gauss-Seidel relaxation shows that convergence is rapid while the residuals have *non-smooth* components and that the smooth components are reduced only slowly. Relaxation methods are good *smoothing* operators.

Hence we now see the core two-grid iteration of all multi-grid methods.

- Pre-smooth ϕ^M using n_{pre} steps of a relaxation method.
- Coarse-grid correction.
- Post-smooth ϕ^M_{new} using n_{post} steps of a relaxation method.

In general the "Coarse-grid correction" will itself be a two-grid iteration, and the algorithm recursively applied up to some grid, G^0, where a direct solution method is inexpensive. All that remains is some method to organize the iterations and to test for convergence to the desired accuracy.

4.2.3 Restriction and Prolongation Operators

The choice of *restriction* and *prolongation* operators is dictated by three factors; the order of the differential equation, limitations on available CPU-time, and the desired accuracy of the solution. The more grid points that are involved in the interpolation process the higher the accuracy of the interpolation, but as more grid points are used the CPU-time required increases.

[5] Waves can only be represented on a grid if they have a frequency below the Nyquist frequency of the grid. If a wave with a frequency higher than the Nyquist frequency is represented on a grid, aliasing occurs and a spurious low frequency wave is produced

The usual choice for the prolongation operator, P, is bilinear interpolation. The number of fine grid points that each coarse grid point is interpolated to is is 3^n, where n is the dimensionality of the problem. Examine the following two-dimensional example.Consider ψ^k at some location (x, y), assuming that ψ^k is non-zero only at (x, y), the "prolonged" values of ψ^{k+1} are given in table 4.1, where $H \equiv h^{k+1}$, the fine-grid spacing. This is sometimes represented by the *symbol*

$$\begin{pmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{pmatrix}. \tag{4.33}$$

Table 4.1: Bilinear Interpolation

location	value of ψ^{k+1}
(x, y)	ψ^k
$(x-H, y), (x+H, y), (x, y-H), (x, y+H)$	$\psi^k/2$
$(x-H, y-H), (x-H, y+H), (x+H, y-H), (x+H, y+H)$	$\psi^k/4$

Several choices of *restriction* operator, $\mathcal{R}$ are immediately apparent. The first is *straight-injection*, we simply copy the values of ψ^k to grid G^{k-1} at co-spatial grid points. A better choice is *full-weighting*, which in 2D is given by

$$\psi^{k-1}(x, y) = (\mathcal{R}\psi^k)(x, y) = \frac{1}{4}\psi^k(x, y) + \frac{1}{8}\psi^k(x + h^k, y) + \frac{1}{16}\psi^k(x + h^k, y + h^k) + \cdots, \tag{4.34}$$

the symbol notation being,

$$\begin{pmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{pmatrix}. \tag{4.35}$$

A further choice in 2D is called *half-weighting*, lying between the two above examples.

$$\begin{pmatrix} 0 & \frac{1}{8} & 0 \\ \frac{1}{8} & \frac{1}{2} & \frac{1}{8} \\ 0 & \frac{1}{8} & 0 \end{pmatrix}, \tag{4.36}$$

MLAPM uses the three-dimensional equivalents of operators (4.33) and (4.35).

Note that the restriction operator, $\mathcal{R}$, and the prolongation operator, $\mathcal{P}$, should always be chosen as a pair. However, in the general case it should not be assumed that

$$\mathcal{PR} \neq 1, \tag{4.37}$$

although MLAPM has this property.

4.2.4 Smoothing Operators

Now let us consider the smoothing properties of relaxation methods. The example I will use is Gauss-Seidel relaxation with lexicographic ordering[6]. The example PDE will be the 2D Poisson equation. Expanding the error locally as a Fourier series will enable the effect of relaxation on different components of the error to be analyzed.

For the 2D Poisson equation, the usual discretization is

$$\frac{\Phi_{n-1,m} - 2\Phi_{n,m} + \Phi_{n+1,m}}{h^2} + \frac{\Phi_{n,m-1} - 2\Phi_{n,m} + \Phi_{n,m+1}}{h^2} = \rho_{n,m}, \tag{4.38}$$

[6] i.e. the points are processed in increasing (i, j) order with the values being used as soon as they become available

where $\Phi_{n,m}$ is the exact solution of this difference equation. Given an approximate solution, $\phi_{n,m}$, a better one can be found using Gauss-Seidel relaxation. The new value of $\phi_{n,m}$, $\bar{\phi}_{n,m}$, satisfies the equation

$$\frac{\phi_{n-1,m} - 2\bar{\phi}_{n,m} + \bar{\phi}_{n+1,m}}{h^2} + \frac{\phi_{n,m-1} - 2\bar{\phi}_{n,m} + \bar{\phi}_{n,m+1}}{h^2} = \rho_{n,m}. \tag{4.39}$$

Let us introduce a new error function, ε defined by

$$\begin{aligned} \varepsilon_{n,m} &= \Phi_{n,m} - \phi_{n,m}, \\ \bar{\varepsilon}_{n,m} &= \Phi_{n,m} - \bar{\phi}_{n,m}. \end{aligned}$$

Subtracting equation (4.39) from equation (4.38) gives

$$(\varepsilon_{n-1,m} + \varepsilon_{n,m-1}) + (\bar{\varepsilon}_{n+1,m} + \bar{\varepsilon}_{n,m+1} - 4\bar{\varepsilon}_{n,m}) = 0. \tag{4.40}$$

Now let is expand the error functions, ε and $\bar{\varepsilon}$, as Fourier series, considering only the $\theta = (\theta_1, \theta_2)$ component,

$$\begin{aligned} \varepsilon_{n,m} &= A_\theta e^{i(\theta_1 n + \theta_2 m)} \\ \bar{\varepsilon}_{n,m} &= \bar{A}_\theta e^{i(\theta_1 n + \theta_2 m)} \end{aligned}$$

Substituting these expressions into equation (4.40) gives

$$A_\theta(e^{i\theta_1} + e^{i\theta_2}) + \bar{A}_\theta(e^{-i\theta_1} + e^{-i\theta_2} - 4) = 0. \tag{4.41}$$

This gives us a relationship between A_θ and $\bar{A}_\theta$. Hence the convergence rate for the θ component is given by

$$\mu(\theta) = \left|\frac{\bar{A}_\theta}{A_\theta}\right| = \left|\frac{e^{i\theta_1} + e^{i\theta_2}}{4 - e^{-i\theta_1} - e^{-i\theta_2}}\right|. \tag{4.42}$$

The *smoothing* factor is given by

$$\bar{\mu} = \max_{\frac{\pi}{2} \leq |\theta| \leq \pi} \mu(\theta), \tag{4.43}$$

where $|\theta| = \max(|\theta_1|, |\theta_2|)$. The range of frequencies is such that $|\theta|$ lies between the Nyquist frequencies of the current and next coarser grid. Evaluating (4.43) yields $\bar{\mu} = 0.5$ and in this case Gauss-Seidel is clearly a very effective smoothing operator.

The smoothing rates can similarly be calculated for other cases. The choice of ordering of the sweep depends on the problem concerned. For general problems red-black ordering is used. If, however, one direction is dominant in the problem, then *line-relaxation*[7] would be more appropriate.

4.3 Multi-grid Cycles

A multi-grid *cycle* is one complete iteration of a multi-grid method from finest grid top coarsest and back again. As we shall see the exact structure of the cycle varies considerable among the different approaches to multi-grid.

4.3.1 A Simple Fixed Multi-grid Cycle

Recall the two-grid iteration from §4.2.2. The simplest multi-grid cycle would be one in which we perform a simple "two-grid" iteration on every level, starting with the finest grid. In this scheme the algorithm will proceed up the hierarchy of grids pre-smoothing on each level until the coarsest grid is reached. Exact solution on the coarsest grid follows with subsequent post-smoothing on every grid-level as the algorithm proceeds back down the hierarchy of grids to the finest grid. A

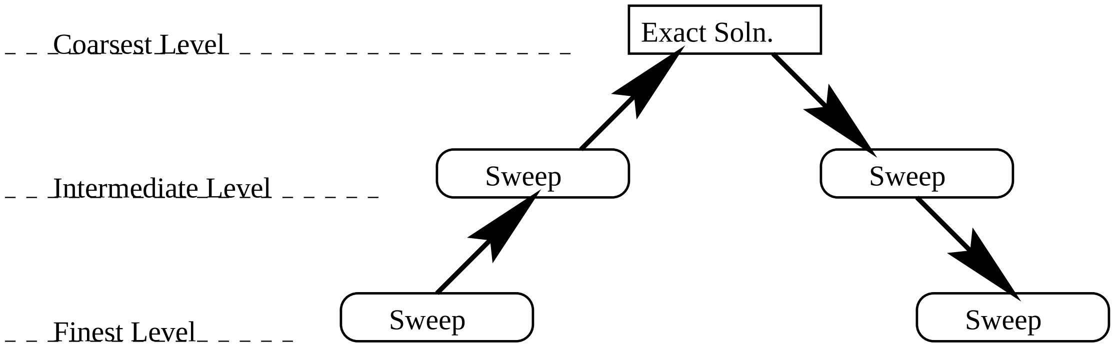


Figure 4.2: A *V-cycle* using three grid levels.

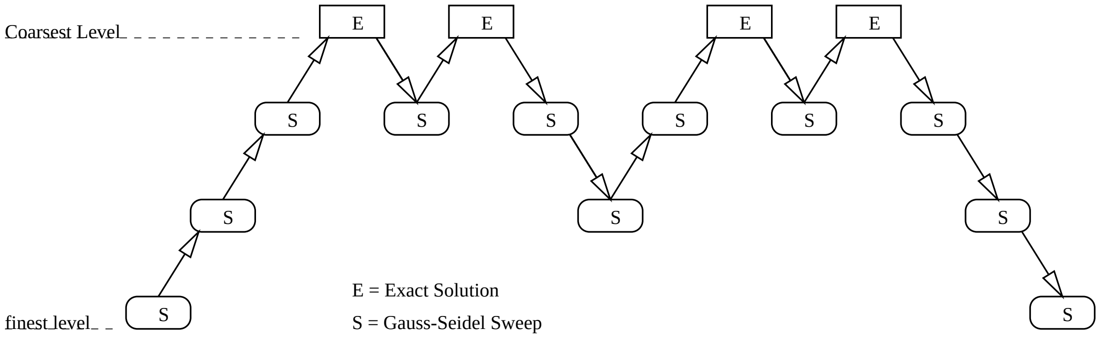


Figure 4.3: A *W-cycle* using four grids.

three-grid example is illustrated in figure 4.2. This is known as a *V-cycle*. Typically several V-cycles will have to be performed to obtain satisfactory convergence. A fixed number of cycles may be performed or a convergence test used on the finest grid and cycles performed until the test is satisfied.

It can sometimes be more efficient to increase the number of two-grid iterations performed on every level. Increasing the number to two gives a W-cycle. A four-grid example is illustrated in figure 4.3.

4.3.2 The Full Multi-grid Algorithm

Many modern multi-grid codes use a more efficient algorithm than the simple V and W cycles illustrated above. This algorithm is known as *Full Multi-grid (*FMG*)* or *nested iteration*. In this case the fine-grid approximation, ϕ, is obtained from a coarse-grid approximation

$$\phi^k = \mathcal{P}\phi^{k-1}. \tag{4.44}$$

The coarse-grid approximation will have been obtained from a similar FMG process. Hence in this approach finer and finer grids are probed as the algorithm proceeds. FMG is illustrated in figure 4.4. A couple of multi-grid cycles are need before proceeding down to the next finer grid. Again the number of two grid iterations on each level may be varied to give V and W-style FMG.

The major disadvantage to the Full Multi-grid Algorithm is that it requires the source terms, f to be defined on every level. For homogeneous boundary conditions it is safe to use *restricted* values. For inhomogeneous boundary conditions the source terms must be discretized afresh on every level.

---

[7] All the points along a line are simultaneously updated.

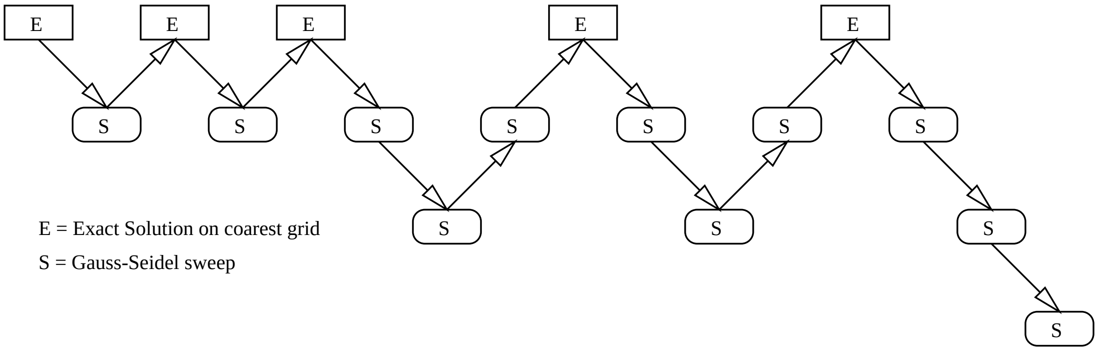


Figure 4.4: The *Full Multi-grid Cycle* using four grids.

4.3.3 *Self-Adaptive* Schemes

Instead of fixing the number and order of the iteration process, a *self-adaptive* scheme changes this "on-the-fly". The algorithm starts on the finest grid in the same way as the simple V-cycle. The decision to go to up to a coarser grid, down to a finer grid, or to remain on the current grid is determined by the results of tests on the current grid. If convergence is slow then the code's logic prompts the algorithm to go proceed with the solution process on the next coarser grid. If convergence according to some criteria has been met then the algorithm goes to the next finer grid (unless the current grid *is* the finest grid in which case the solution has been obtained and the algorithm stops). If neither of the above conditions are met then the algorithm conducts more smoothing on the current grid. The flow diagram in figure 4.5 illustrates Brandt's cycle C, a version of the above scheme.

In Brandt's self-adaptive scheme, we can view the multi-grid in a slightly different manner. The coarse-grid corrector step may be viewed as an alternative to the relaxation sweep. When the relaxation sweep's convergence rate becomes low, it is appropriate to apply a coarse-grid correction as it is likely that the residuals contain frequency components best solved on a coarser grid. The equations to be solved approximately are

$$L^k \Phi^k = u^k, \qquad \Lambda^k \Phi^k = v^k, \tag{4.45}$$

where the residuals, u^k and v^k, are given by

$$\begin{aligned} u^k &= \mathcal{R}(u^{k+1} - L^{k+1}\psi^{k+1}), \\ v^k &= \mathcal{R}(v^{k+1} - \Lambda^{k+1}\psi^{k+1}). \end{aligned}$$

The algorithm is best understood by following the above flowchart through.

MLAPM uses a self-adaptive scheme.

4.4 Full Approximation Storage Mode:FAS

The development of the *Full Approximation Storage Mode* or *FAS* mode is crucial to the use of multi-grid on non-uniform and adaptive grids. FAS also provides a way of solving non-linear equations directly, with non-linear equations only having to be solved on the coarsest grid. The major difference between FAS modes and ordinary multi-grid is that the full approximation, ϕ^k, is stored on all grids instead of just the correction, ψ^k:

$$\phi^k = \mathcal{R}\phi^{k+1} + \psi^k \qquad \forall\, k = 0, 1, \ldots, M-1. \tag{4.46}$$

The correction equations themselves are altered

$$L^k \Phi^k = \bar{U}^k, \qquad \Lambda \Phi^k = \bar{V}^k, \tag{4.47}$$

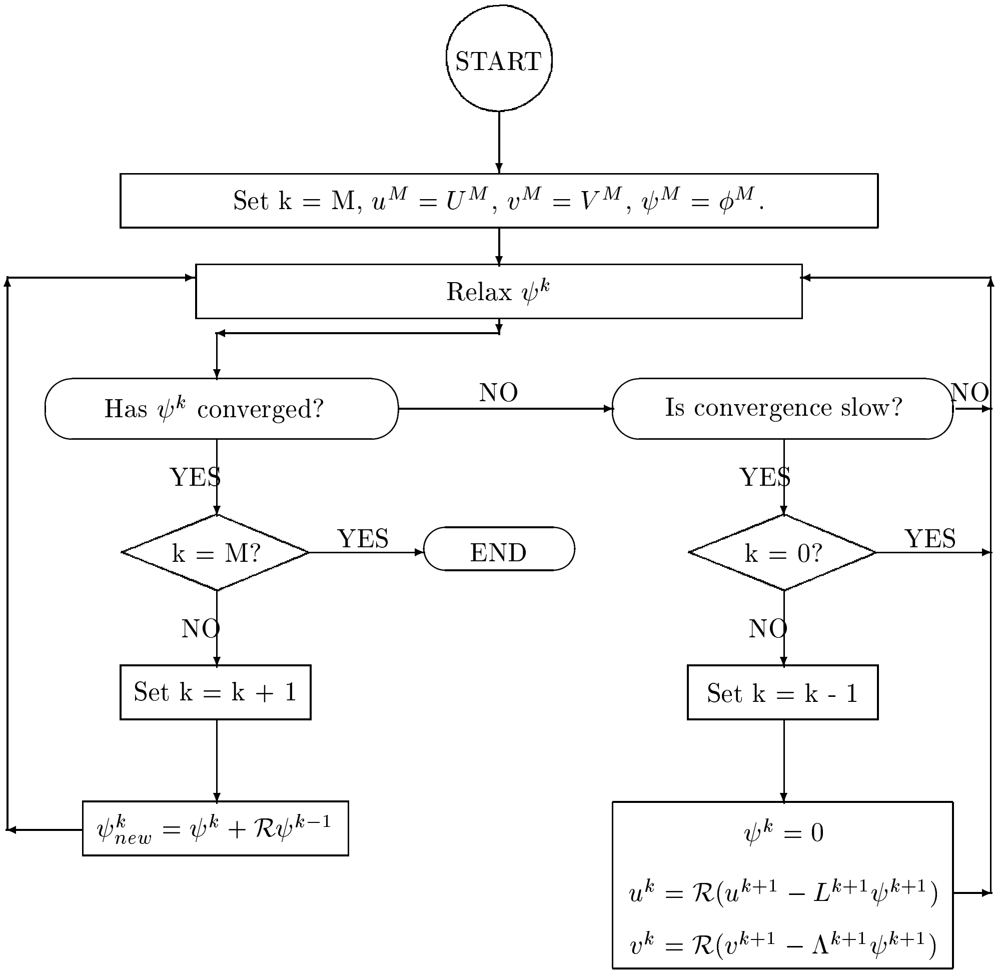


Figure 4.5: A flowchart illustrating Brandt's cycle C, self-adaptive scheme.

where the right-hand sides become

$$\bar{U}^k = L^k(\mathcal{R}\phi^{k+1}) + \mathcal{R}(\bar{U}^{k+1} - L^{k+1}\phi^{k+1}), \quad (4.48)$$
$$\bar{V}^{k+1} = \Lambda^k(\mathcal{R}\phi^{k+1}) + \mathcal{R}(\bar{V}^{k+1} - \Lambda^{k+1}\phi^{k+1}), \quad (4.49)$$

instead of the residuals. On the finest grid, $k = M$, and the above equations are replaced by the original equations,

$$\bar{U}^M = U^M, \qquad \bar{V}^M = V^M. \quad (4.50)$$

To see how we obtain equations (4.48) and (4.49) consider that we are seeking a correction, ψ^k, such that

$$L^M(\phi^M + \psi^M) = U^M. \quad (4.51)$$

Adding a term $-L^M\phi^M$ to both sides of (4.51) we obtain

$$L^M\Phi^M - L^M\phi^M = u^M. \quad (4.52)$$

The coarse grid approximation of this is given by

$$L^{M-1}\Phi^{M-1} - L^{M-1}\mathcal{R}\phi^M = \mathcal{R}u^M, \quad (4.53)$$

which gives the same right-hand side as equation (4.48) for general k. To interpolate back to grid G^M we should interpolate the function $\Phi^{M-1} - \phi^{M-1}$ rather than Φ^{M-1}. Hence the coarse-grid correction is given by

$$\phi^{k+1}_{new} = \phi^{k+1} + \mathcal{P}(\phi^k - \mathcal{R}\phi^{k+1}). \quad (4.54)$$

4.4.1 A Stopping Criterion

The truncation error, τ, is the error introduced by the discretization process. We can do nothing about the truncation error save increase the number of grid points. On each grid used in the solution process, it is pointless trying to continue if the residuals are significantly smaller then the truncation error. Hence we continue the solution until

$$\|u^k\| < a\tau \quad (4.55)$$

where $\|\cdot\|$ is a suitable norm and a is an appropriate fractional coefficient[8].

Unfortunately, we cannot measure the truncation error directly. We can estimate it, however. We define the *relative truncation error*

$$\tau^k \equiv L^{k-1}(\mathcal{R}\Phi^k) - \mathcal{R}L^k\Phi^k. \quad (4.56)$$

Using $L^k\Phi^k = U^k$ this becomes

$$L^{k-1}\Phi^{k-1} = U^{k-1} + \tau^k. \quad (4.57)$$

This expression shows that we can regard τ^k as the correction to U^{k-1} that makes the coarse-grid solution equal to the fine-grid solution. Again τ^k cannot be computed but an estimate is given by

$$\tau^k \simeq \tilde{\tau}^k \equiv L^{k-1}(\mathcal{R}\phi^k) - \mathcal{R}L^k\phi^k. \quad (4.58)$$

Recall the expression for the right-hand sides in FAS mode, equation (4.48). $\tilde{\tau}^k$ appears in this expression and hence is a natural by-product of the FAS mode multi-grid.

[8] In MLAPM $a = 0.025$.

4.5 Adaptive Multi-grid

As noted in chapter 1, the use of an adaptive grid would be very useful in cosmological simulations. In fact many problems require the use of a grid with variable resolution across the computational domain for maximum efficiency. There are several approaches to adaptive grids, as we shall see in §4.5.1.

4.5.1 Approaches to *Non-uniform* and *Adaptive* Grids

The *Finite Element* Approach

A totally flexible grid is usually used with the finite element method. Grid points may be placed anywhere within the computational domain. Organizing a grid like this is memory expensive as it requires a lot of pointers to store the locations of neighboring grid points. The coefficients of the difference equations do not remain constant and are now determined locally, this calculation may involve significant computational cost. Of course, if the grid is adapting as the solution proceeds, then these coefficients need to be recalculated repeatedly. Even with these disadvantages the finite element approach is very useful in certain contexts, particularly fluid dynamics.

Coordinate Transformation

Another alternative is to use a grid which effectively flexes to fit the problem. A regular grid is used, saving on the use of pointers, but a coordinate transformation is used, effectively giving more grid points where required. The difference equations have to be modified locally to take account of the transformation. As with the finite element approach this will need to be an ongoing process if the grid adapts as the solution proceeds.

A Hierarchy of Non-Coextensive Grids

It doesn't take much imagination to see how the hierarchy of grids we already have in use with multi-grid can be modified to be non-coextensive. The only rule we need to apply is that no grid covers more of the computational domain than the next coarser grid. A regular Cartesian grid can be used with the usual 2:1 ratio of grid sizes[9]. This scheme of course uses many grids, but this is required by multi-grid anyway, so there is no excess demand on storage. The difference equations remain constant so no expensive recalculation of coefficients is required. The grid can easily be altered both inside and outside the solution process by extending or shrinking existing grid levels or by adding further finer levels. In fact the computational domain itself can be extended if required by introducing new coarser levels. This is the scheme implemented in multi-grid.

4.5.2 Multi-grid on Non-Uniform Grids

Performing standard multi-grid on a set of grids which are not coextensive can lead to problems. A grid G^k would perform two different roles. In regions where a finer grid G^{k+1} were defined it has to act as a correction grid and ψ^k would be stored. However, in regions where no finer grid was defined it would act as the source grid and ϕ^k would be stored, thus leaving open ambiguities. The problem is circumvented with the use of the FAS mode as then the full approximation is stored on all grids and the correction only implicitly calculated.

The iteration equations are modified as follows to account for non-uniform grids although it is self evident how one should proceed. Let G^k_m, with $k < m$, be the set of points of grid G^k that lie within the region covered by grid G^m (i.e. not on the boundary on G^m). The difference equations become

$$L^k \Phi^k = \bar{U}^k, \qquad \Lambda^k \Phi^k = \bar{V}^k, \tag{4.59}$$

[9] A triangular based grid with a ratio of $\sqrt{2} : 1$ is also popular.

where

$$\bar{U}^k = U^k \quad \text{and} \quad \bar{V}^k = V^k \quad \text{in} \quad G^k - G^k_{k+1} \quad \text{and for} \quad k = M, \tag{4.60}$$

$$\bar{U}^k = U^k_{k+1} \quad \text{and} \quad \bar{V}^k = V^k_{k+1} \quad \text{in} \quad G^k_{k+1}, \tag{4.61}$$

where U^k_{k+1} and V^k_{k+1} are given by

$$U^k_m = \mathcal{R}(\bar{U}^m - L^m \phi^m) + L^k(\mathcal{R}\phi^m), \tag{4.62}$$
$$V^k_m = \mathcal{R}(\bar{V}^m - \Lambda^m \phi^m) + \Lambda^k(\mathcal{R}\phi^m). \tag{4.63}$$

The convergence criteria must also be modified[10]. As two neighboring grids no longer cover the same region, care must be taken in calculating norms etc.

4.5.3 Adaptive Grids in the Solution Process

Although MLAPM in its present form uses a grid that has its resolution set by external factors to the solver (the density in fact) it is worth a brief look at how adaptive griding could be introduced while actually solving the difference equations. As can be seen from the above discussion, the grid sizes can be easily changed and new grids readily introduced. There is no reason why the grids cannot be altered during the solution process. Fundamentally the decision to alter the grids has to be made with reference to two criteria

- Does the error associated with the discretization warrant a finer grid?
- Is a finer grid worth the addition work in relaxation sweeps?

Numerical estimates of the discretization error and the work involved in adding more grid points can be obtained from measures of the truncation error, the grid size ratio and the number of operations per grid point. For further details I refer the reader to (Brandt, 1977a).

[10] For *self-adaptive* schemes the *slow convergence* criteria must also be altered

Chapter 5

Data Structure Design: Supporting an Adaptive Grid

5.1 Implications of an Adaptive Grid

In chapter 4 I discussed how multi-grid methods may be used without much modification on an adaptive grid. However, the decision to move to an adaptive grid has far reaching consequences. Although the multi-grid *algorithm* does not change much, the multi-grid *program* certainly does. In subsequent sections I will discuss several alternative data structures that support adaptive grids, each with advantages and disadvantages.

There are some general considerations that apply in all cases. The first of these is the requirement for *dynamic memory allocation*. New grids have to be generated and old grids altered in size. For the code to be in any way efficient new memory for these grids needs to be claimed from the computers' operating system and memory that becomes surplus to requirements needs to be returned to the control of the operating system. Dynamic memory allocation is not implemented directly in older languages like *Fortran 77*[1 2]. The use of a language that properly implements dynamic memory allocation may be regarded as essential. An obvious choice of language would be ANSI C although any modern high level language could be used.

A second general consideration, which has implications for the choice of programming language, is the interface between the data structure and the user. Ideally the interface should be as transparent as possible for ease of programming and debugging. However, with the more complex data structures this requires the use of high level programming ideals such as modular programming, data abstraction and object-orientated programming. A language such as *C++* contains all these features and would the ideal choice of language for such projects. However, MLAPM is implemented in C, for historical reasons.

5.2 Rectangular Grids

The simplest data structures one could use for an adaptive grid are simple n-dimensional arrays. *Sub-grids*[3] have dimension (i, j, k) with i, j and k all odd. As with all adaptive multi-grid schemes, fine sub-grids must be placed so that the nodes on the edges of the grid are co-spatial with the nodes of the next coarser grid. The major advantage of this scheme is that it is very simple to implement. Code to identify neighbouring grid points for Gauss-Seidel sweeps and interpolation is easy to write. Debugging is also relatively easy as the grid data can be easily accessed inside a debugger.

[1] In practice many compilers contain these features, however they are not part of the ANSI standard and hence code using proprietary features is not necessarily portable

[2] A form of dynamic memory allocation can be implemented on Fortran 77 (see Fortran version of (Press et al., 1992)) although it is slow and unwieldy

[3] Defined as grids which do not cover the whole computational domain.

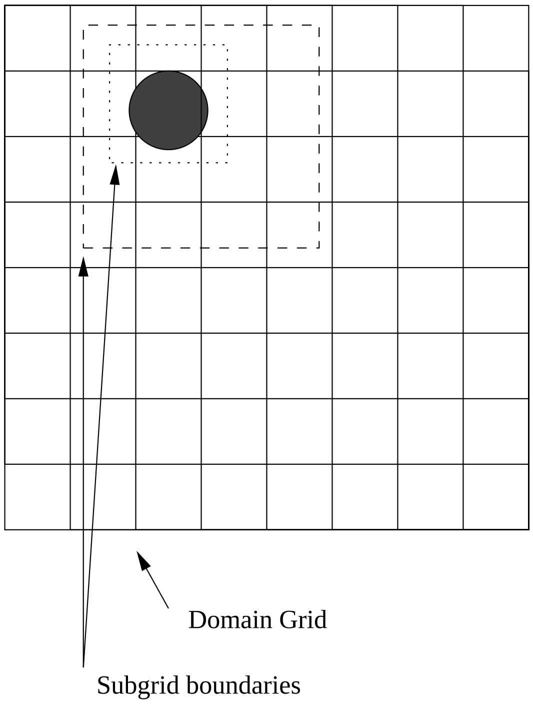


Figure 5.1: Rectangular grids are quite efficient for large single objects but are less so for filamentary structures.

Simplicity of coding, however, leads to complexity in the placement algorithm. To be efficient grids must be placed in the correct location and made to be the correct size. There may be more than one cuboidal sub-grid on the same *level*[4], however the grids cannot overlap and the placement algorithm must know how to deal with such a situation.

In the context of an N-body simulation, rectangular grids are not necessarily memory efficient. Galactic halos are usually spheroidal and many structures in the universe are filamentary in nature. Rectangular grids could leave large volumes of “wasted” space[5]. Nevertheless many adaptive N-body codes have used this sort of data structure. The notable examples are the codes of Suisalu & Saar (1995b) (multi-grid), Couchman (1991)(FFT based P^3M) and Villumsen (1989) (FFT based PM).

5.3 The Oct-Tree

The most significant problem any data structure used with a multi-grid is that of finding the neighbouring grid points. In an ideal world one would use a scheme where each grid point would have a record of its nearest neighbours. In C one would implement this with pointers

```
struct gridpoint{
  /* grid data */
  ⋮
  /* pointers to neighbours */
  gridpoint *neighbours[NO_NEIGHBOURS];
}
```

[4]i.e. having the same grid spacing

[5]See Figure 5.1

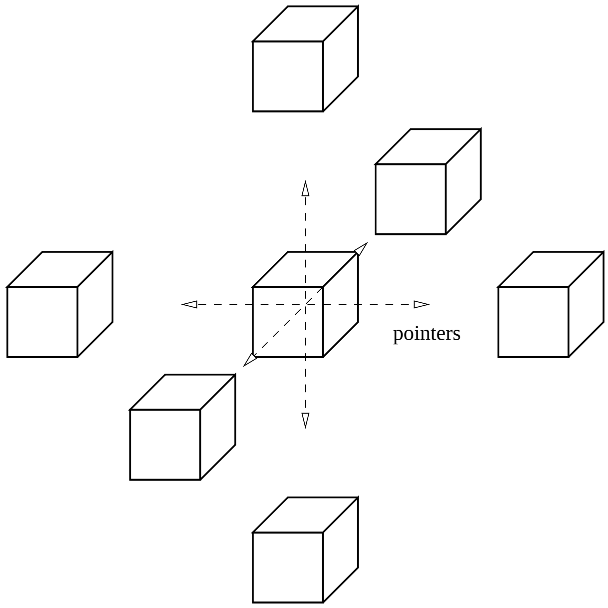


Figure 5.2: A scheme with six pointers per grid point would make accessing neighbours easy but it would also be memory expensive.

Such a scheme is useful only when storage is not the limiting factor on accuracy as the set of pointers uses a large amount of memory. At eight bytes per pointer on a 64 bit computer this will rapidly mount up. The ART code (see e.g. (Kravtsov et al., 1997)) uses a data structure of this type. Hence it will use a minimum of 48 bytes per node in pointers alone on a 32-bit computer.

The Oct-Tree essentially takes this idea, but makes it less memory expensive. Instead of using a set of pointers per grid point the oct-tree uses a set of pointers for a block of grid points. To see how this would work consider the one dimensional case of the binary tree.

Consider figure 5.3. We start with a 1-dimensional "grid" with 16 grid points which covers the whole computational domain. There are two high density regions which require higher resolution grids covering them. To introduce a refinement the grid is split into two halves. Taking each half in turn the grid is tested to see if refinement is needed. If refinement is needed a "daughter" grid of length 16 grid points is generated and a pointer belonging to the parent grid is set to point to the daughter grid. Similarly a pointer is set from the daughter grid to point to the parent grid. If no refinement grid is required in a particular half of the original grid, the sub-grid pointer remains `NULL`. Each daughter grid also has two pointers ("pointing" left and right) to identify neighbouring grids. After the new sub-grids have been generated a search through the tree for neighbours must be performed. In C this 1D example may be implemented as follows

```
struct gridpoint {
  /* grid data */
  ⋮
}

struct grid {
  struct gridpoint grid_data[16]; /* array of grid points */
  struct grid *parent;            /* pointer to parent grid */
  struct grid *daughter_left;     /* pointer to left subgrid */
  struct grid *daughter_right;    /* pointer to right subgrid */
  struct grid *left;              /* pointer to left neighbour */
  struct grid *right;             /* pointer to right neighbour */
}
```

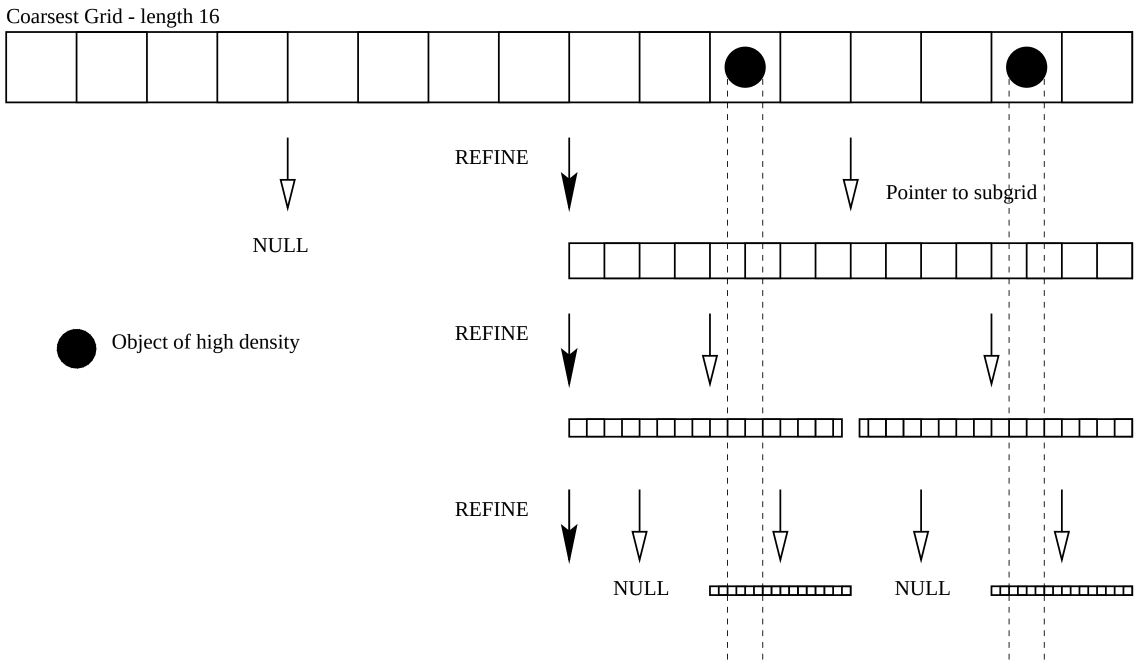


Figure 5.3: The binary tree with associated pointers is illustrated here. As refinement proceeds each half grid is selectively refined to place grids over the regions of interest.

In three dimensions the binary tree becomes an oct-tree. Eight pointers are required to point to daughter grids and six to point to neighbouring grids. The three dimensional case is illustrated in Figure 5.4. The refinement scheme is no more than an extension of the binary one described above.

The oct-tree data structure has great advantages. Firstly the grid structure is self-similar, this enables recursive programming to be used. Secondly, no complex logic is required to place grids. To determine if a corner of the grid needs refinement a simple criterion, such as testing whether the corner is over some density threshold, could be used.

Ultimately the oct-tree was not chosen as the data structure for MLAPM. The major reason for this lies in the somewhat artificial boundaries the data structure imposes. Objects near the centre of any particular grid would need all eight corners of that grid to be refined to ensure equal force resolution for all the particles comprising the object, this is illustrated in Figure 5.5. This could be inefficient as well as leading to potentially spurious results. Even with these faults the

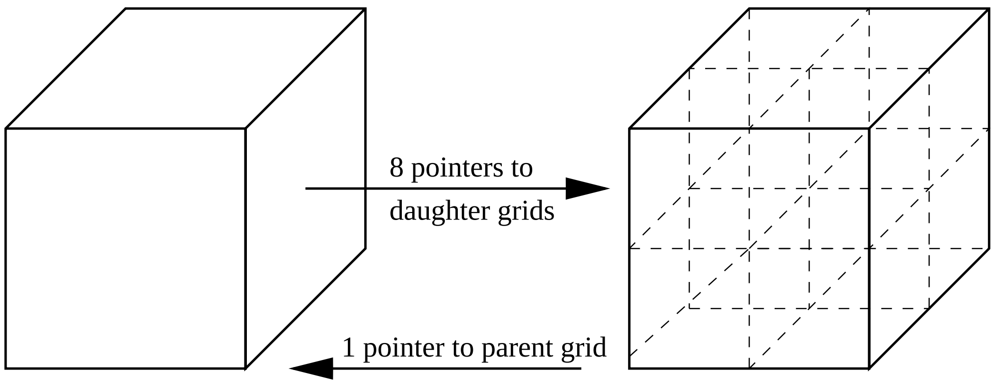


Figure 5.4: The oct-tree refinement scheme

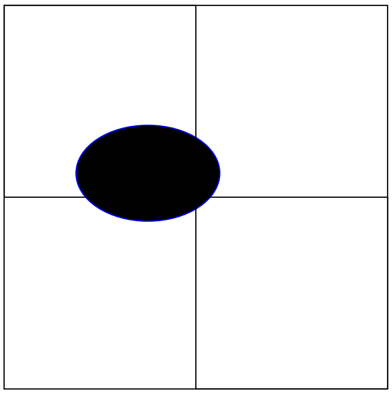


Figure 5.5: For objects which are close to the centre of a coarse grid, refinement in the n-tree data structures can be inefficient.

oct-tree is still a very viable option as it is much easier to use than the alternatives. For more details of the oct-tree see for example (Rosendale, 1983).

5.4 The QUAD structured grid

The ideal data structure would allow a grid that was "swiss cheese" like, ie a Cartesian-based grid but within which points may or may not be present. The only rule that need be imposed on a finer grid is that it cover no part of the computational domain that the next coarser grid does not. This "whole grid" approach to generating refinement avoids the artificial boundaries of the data structures discussed above. Such a grid would clearly be more efficient as fewer grid points would be generated upon refinement. Fewer grid points not only means a lower storage requirement but also less CPU time as less points need to be iterated over in the multi-grid cycle.

Can such an ideal grid be supported? Brandt's original data structure used with adaptive multi-grid allowed such a grid (Brandt, 1977b). This data structure was used in two dimensions and I have made the trivial adaptation to three for use with MLAPM. To see how the data structure works let us build up from one to three dimensions. A full description of the C implementation follows in section §5.5.

Consider the one-dimensional distribution of grid points illustrated in Figure 5.6. There are two groups of grid points separated by a gap. Assume there are no more groups to the left or right of those shown. It is easy enough to `malloc()` memory for each separate block of points, however a further logical structure is required to describe each. A pointer can be incremented down the row of points so only a pointer to the first point of a block needs to be stored. The absolute coordinate of the first point is also required to fix the block of points in the computational domain. The number of points in a block obviously must be stored. These three pieces of information are sufficient to describe each block of points individually but an additional pointer is required to link them. This *next* pointer is set to point to the logical structure describing the next block of points in increasing index order. If there are no more blocks of points this pointer is set to be `NULL`. As four pieces of information are required Brandt labelled these structures QUADs.

$$(location \mid start\ index \mid length \mid next)$$

The quad structures for the example are also illustrated in figure 5.6. A C implementation of a QUAD would be

```
struct quad {
  struct gridpoint *location; /* pointer to first point of block */
  int start;                  /* index of first point of block */
  int length;                 /* number of points in block */
```

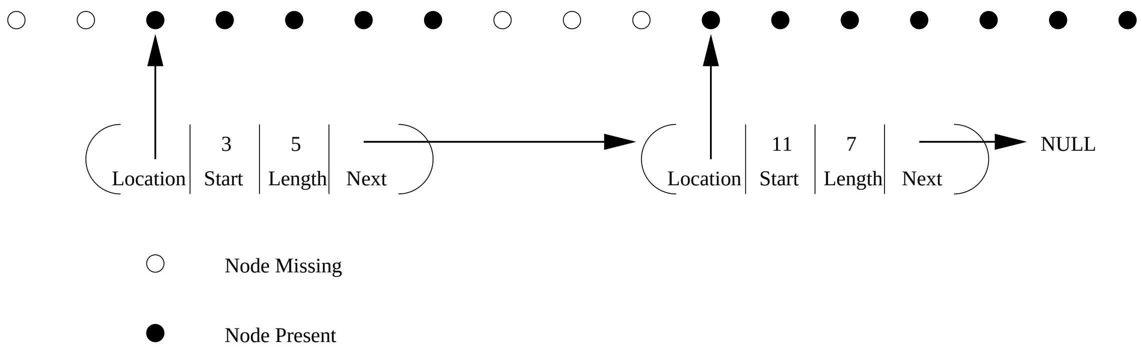


Figure 5.6: In this example the two QUAD structures can be clearly seen to fully describe the structure of the grid.

```
  struct quad *next;          /* pointer to next quad */
}
```

The next step is to extend this scheme to two dimensions. This is illustrated in figure 5.7. Let us regard each row of points as a single entity. The data in each row can be accessed from the first quad structure describing the points in that row. Just as with grid points some of the rows will be completely absent. Hence we are left with blocks of quads and as with grid points these blocks of quads can be described by a quad structure. Blocks of quads are `malloc`ed together and hence are adjacent in memory. A pointer may be incremented to access all the quads if the address of the first and the number of quads in the block is known. The "y" index of the first quad enables the real location of the quads to be identified. In is now trivial to see how this data structure may be extended to n dimensions simply by adding layer upon layer of quad structures.

The quad-based data structure achieves the desired flexibility in the placement of grid points. However, there is a always cost associated with such gains. The major problem with the quad data structure is that is unwieldy for the programmer to use. Accessing the data on the grid for the purposes of debugging etc., can be very difficult as it will often involve many pointer accesses. Programming search routines to identify neighbouring grid points involves complex logic routines while accessing data inside a debugger "on the fly" can be time consuming and frustrating.

Ideally the programmer needs to build a transparent interface between himself and the data structure to eliminate these problems. One could build an array-type function that retrieves a grid point at the required location if it is present and returns `NULL` if it is not. The programmer could then use such a function as the basis for all the search routines. This is not implemented in MLAPM. The search functions of MLAPM are "hard-wired" as accessing each grid point individually is not the most efficient way of finding neighbours as in the majority of cases two of the six neighbours may be accessed simply by incrementing a single pointer. In the desire for raw speed the more complex option was chosen (at the expense of many extra programmer hours).

Does the complex nature of the data structure have an impact on the speed of the code? The answer to that question depends on how the code is used. Typically for N-body simulations the code will be operated with many more particles than grid points and particle operations will dominate. On the other hand in tests with more grid points than particles, the neighbour search routines clearly dominate the CPU use profile. A theoretical insight into the impact of such a data structure can be gained as follows. The user should realize that data within the computers' memory is stored in a linear fashion. Three dimensional arrays still have to be accessed through pointer accesses by the computers' operating system. In general such system calls will be faster than any access the programmer can use, but the difference will not be too great. Hence the impact such a data structure has on the speed of the code may not be as great as thought initially. I refer the reader to later chapters for actual speed tests of MLAPM.

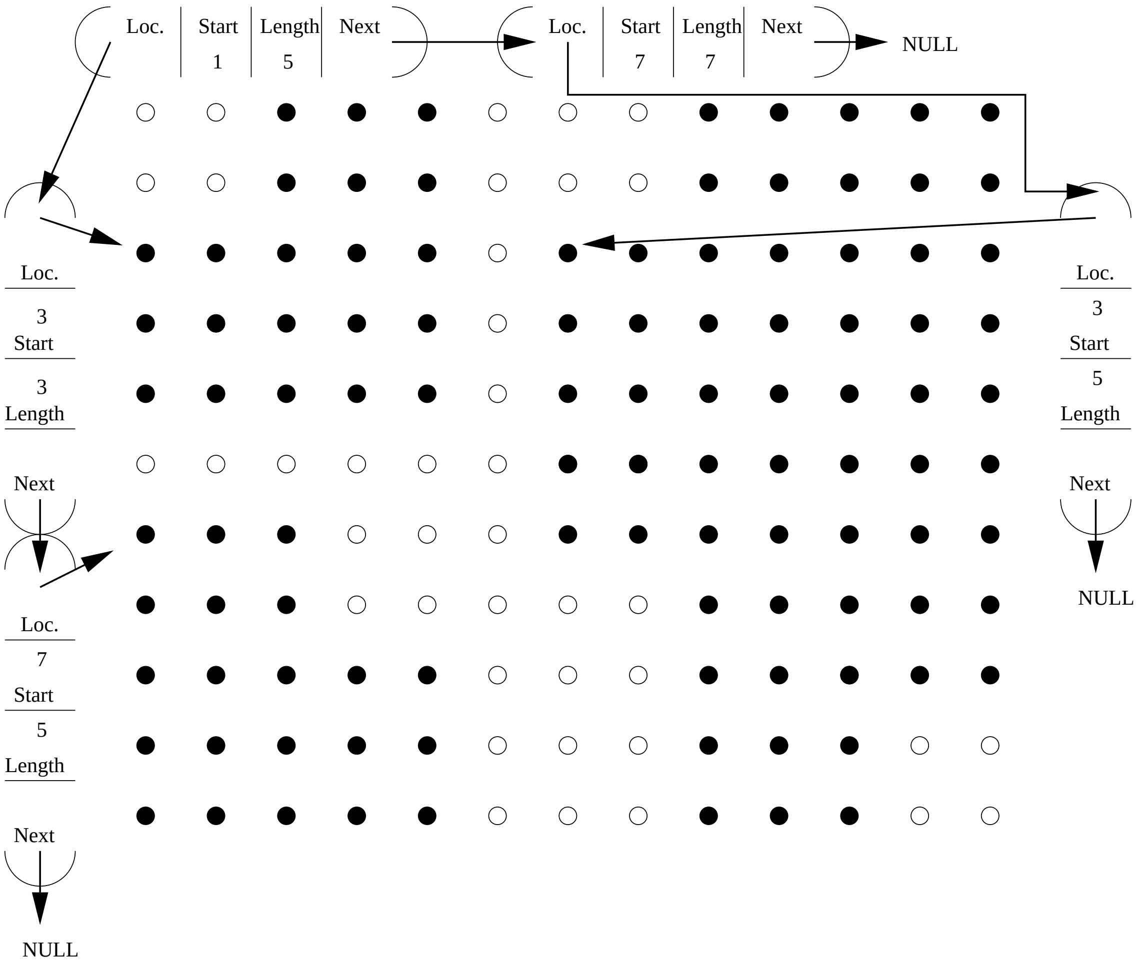


Figure 5.7: When the dimensionality is increased a second "level" of QUAD structures is added. Only the first *nquad* is shown in each block of nquads for clarity.

5.5 The QUAD structure in MLAPM

This section is intended to not only show how I have implemented the QUAD data structure in MLAPM but also to introduce the reader to the terminology I use to refer to various parts of the data structure. A good understanding of the various `typedef`'s will be vital when reading chapter 7 and the description of MLAPM. The code this section describes may be found in the C header file `tdef.h`.

5.5.1 QUAD definitions

All the key parts of the data structure use type definition statements to define their own keywords. This eliminates the need to use the `struct` keyword throughout the code, which aids readability. However this does mean that the reader must gain familiarity with the new types to understand any of the main body of the code. Pointer types are also type defined for most of the structures.

The most important structure is the "gridpoint", this is labeled a *node* in MLAPM. The use of node means identification is unambiguous. Pointers to node structures are `typedef`ed as `nptr`.

```
typedef struct node{
  float pot;                  /* gravitational potential */
  float dens;                /* mass density */
  partptr ll;                 /* head of particle linked-list */
#ifdef MULTISTEP
  /*
   * If MLAPM is being used in multiple timestep configuration then the
   * density must be stored. Mass assignment cannot occur in regions that
   * have been refined as the particles stored on the refined grid.
   * Hence the old density value must be replaced.
   */
  float olddens;             /* stored value of density */
#endif
  /*
   * The forces at the node are only required at the end of a timestep.
   * Hence the space allocated to the forces is used to store temporary
   * data at other times.
   */
  union force{
    float temp[3];           /* old potential, original density */
    float forces[NDIM];      /* grid force components */
    partptr new_ll;          /* new pointer for rebuilding linked list */
  }force;
}node, *nptr;
```

A separate QUAD is defined for each dimension. The use of three types of quad results in additional source code, however debugging benefits of having distinct types for the second and third dimension out weigh this cost. The three types are

- `nquad` or node quad - describes blocks of nodes.
- `cquad` or column quad - describes blocks of columns.
- `pquad` or plane quad - describes blocks of planes.

The three corresponding pointer types are `nqptr`, `cqptr` and `pqptr` respectively. The quad structure definitions are

```
typedef struct nquad *nqptr;

typedef struct nquad{          /* node-QUAD logical data structure */
  nptr loc;                     /* pointer to first node in block*/
  long x;                       /* x-coord of first node in block */
```

```
  long length;                    /* length of block of nodes */
  nqptr next;                     /* pointer to next nquad (or null) */
}nquad;

typedef struct cquad *cqptr;

typedef struct cquad{            /* column-QUAD logical data structure */
  nqptr loc;                      /* pointer to first nquad in block */
  long y;                         /* y-coord of first nquad in block */
  long length;                    /* length of block of nquads */
  cqptr next;                     /* pointer to next cquad (or null) */
}cquad;

typedef struct pquad *pqptr;

typedef struct pquad{            /* plane-QUAD logical data structure */
  cqptr loc;                      /* pointer to first cquad in block */
  long z;                         /* z-coord of first cquad in block */
  long length;                    /* length of block of cquads */
  pqptr next;                     /* pointer to next pquad (or null) */
}pquad;
```

Each grid (made up of QUAD structures and nodes) has an entry in an array of structures of type `gridls`. The array is re-sized as grids are generated and destroyed. The gridls structure contains a pointer to the first pquad as well as all the variables associated with the grid as a whole.

```
typedef struct gridlist{       /* grid variables & pointer to grid */
  long l1dim;                   /* virtual grid length */
  float grid_spac;              /* grid spacing */
  float l2dim;                  /* l1dim squared */
  float grid_spac_sqrd;         /* grid spacing squared */
  pqptr gridptr;                /* pointer to first pquad on grid */
  float critdens;               /* critical density on this grid */
  float old_resid;              /* old residual norm */
  float cur_resid;              /* current residual norm */
  float counter;                /* timestep counter */
  double a;                     /* scale factor (cosmic) */
  double time_level;            /* timelevel of this grid */
  double u;                     /* potential energy of particles */
  double old_t;                 /* ke of part on this grid at v(n - 1/2) */
  double new_t;                 /* ke of part on this grid at v(n + 1/2) */
}gridls;
```

Domain Grids

In MLAPM not all the grids are adaptive. The set of grids of length two up to the length set by the parameter `MAXDL1D` cover the whole computational domain. To distinguish these grids from the adaptive grids they are referred to as *domain* grids.

Particle Storage

The reader will note that the definition of the *node* above includes a *linked-list* pointer. There are several ways to arrange access to the particles. It is clear that the particles must be associated with the finest grid that covers the part of the computational domain that they lie within. Hence at the very least lists of particles within each grid should be maintained. It was deemed that it would be more appropriate to associate the particles with the actual cells within which they lie and to store them within linked lists. This decision was based both on the requirements for ease of access to particles within a given grid and to facilitate a P^3M version of the code.

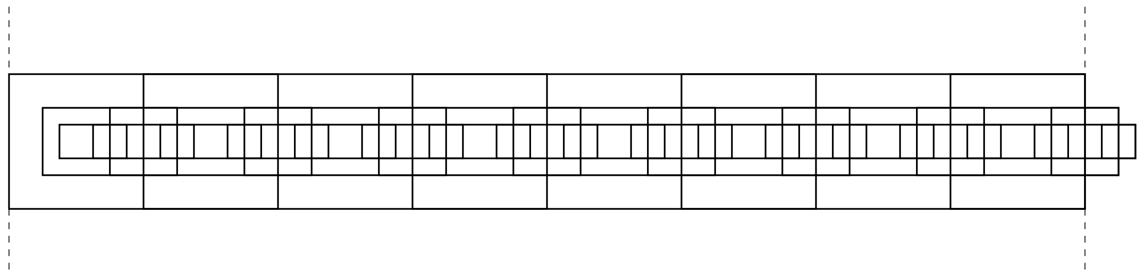


Figure 5.8: Periodicity leads to the non-alignment at the periodic boundary, indicated by the dashed line. The placement of particles with the grid hierarchy is complicated by this.

Periodicity

Cosmological simulations typically use triply-periodic boundary conditions on the edges of the computational box. MLAPM is no exception, but periodicity has serious implications for an adaptive grid and its supporting data structure. Firstly there is the problem that refined regions may extend across the periodic boundary. Secondly there is a problem over the actual definition of the boundary itself.

To make the grid periodic each grid must have a length in nodes that is a power of two[6]. Hence a typical sequence of grid lengths would be

$$2, 4, 8, 16, 32, \mathit{64}, \mathit{128}, \mathit{256}, \mathit{512}, \ldots$$

where the grid of length 32 nodes is the last domain grid. The grids are arranged so that the first node on every grid is co-spatial with a node on the next coarser grid. A boundary node may be identified as a node having any coordinate equal to $(gridlength - 1)$. The last quad *next* pointer in any direction is *not* set to point to the first quad but remains `NULL`. When processing nodes during a Gauss-Seidel sweep etc., when a quad is encountered with a null next pointer, the coordinate of the last node (or nquad or cquad) is calculated from the quad data and compared with the value of $(gridlength - 1)$ to see if it lies on the periodic boundary.

Consider figure 5.8, this illustrates several levels of a one dimensional grid. The first nodes are all co-spatial with each other. However the cells surrounding the nodes do not cover the same computational domain. Hence the periodic boundary is in a different location on every level of refinement. This has no impact on the multi-grid solver.

5.5.2 Other Structures and Type Definitions

There are a few other structures and type definitions that for completeness I will describe now. Again all these definitions are in the header file `tdef.h`.

There are two definitions which are useful tools. The first is a "vector" of `NDIM` floats, where `NDIM`[7] is a parameter defining the number of dimensions.

```
typedef float fvect[NDIM];
```

I also define a *boolean* data type for use with logic tests. C, unlike Pascal, has no separate boolean data type.

```
typedef char boolean;
```

A structure is used for the particles. The structure contains two `fvect`'s for the three components of the position and the three components of the velocity. As the particles are stored in linked lists the particle structure also contains a pointer to a particle structure. If MLAPM is to be used in multiple timestep mode, a further variable is required. The variable *stepsize* indicates

[6] The length may of course be the *virtual* length as if the grid were complete when it is not.

[7] Parameters are defined in the header file `param.h`, see §7.1.1 for further details

whether the particle should move a full or half timestep. Half timesteps are required for particles which, having done one timestep on a sub-grid, move out of that sub-grid into the next coarser grid. Owing to the sequence of timesteps[8], the coarse-grid particles move a timestep twice the size of the timestep on the next finer grid.

```
typedef struct particle *partptr;

typedef struct particle{       /* particle data */
  fvect pos;                    /* position vector */
  fvect mom;                    /* momentum vector */
  partptr ll;                   /* linked-list coord */
  float stepsize;               /* fraction of domain timestep done */
}part;
```

Some cosmological parameters are stored in the following structure which allows all the data to be kept together.

```
typedef struct cosprm{
  float h;           /* 100 h kms-1Mpc-1 */
  float omega;       /* density parameter */
  float box_length;  /* length of box in Mpch-1 */
  float force_coeff; /* "mass" - normalising force coefficient */
}cosprm;
```

The final new type is the "difference star" or `diff_star`. This type is used to return pointers to the six surrounding nodes in three dimensions.

```
typedef struct diff_star{
  nptr nfor;     /* node forward */
  nptr nback;    /* node backwards */
  nptr nleft;    /* node left */
  nptr nright;   /* node right */
  nptr nup;      /* node above */
  nptr ndown;    /* node below */
}diff_star, *diff_star_ptr;
```

[8]For a full explanation of the timestep sequence please refer to §6.5

Chapter 6

Integrating the Equations of Motion

The reader will recall from §3.2.1 that MLAPM (and indeed most cosmological simulation codes) use time centered leapfrog scheme as a discrete approximation to the equations of motion

$$\frac{d\mathbf{u}}{dp} + \frac{(1 + \alpha + \ddot{a}a/\dot{a}^2)}{\alpha a^\alpha}\mathbf{u} = -\frac{\nabla\phi}{\alpha^2\dot{a}^2a^{2\alpha+1}}, \tag{6.1}$$

where

$$\mathbf{u} = d\mathbf{r}'/dp, \tag{6.2}$$

and the time variable, $p = a^\alpha$. The discrete analogues of these equations are

$$\mathbf{u}_{n+1/2} = \frac{(1 - A_n\Delta p)\mathbf{u}_{n-1/2} + B_n\mathbf{F}_n\Delta p}{1 + A_n\Delta p}, \tag{6.3}$$

$$\mathbf{r}'_{n+1} = \mathbf{r}'_n + \mathbf{u}_{n+1/2}\Delta p, \tag{6.4}$$

where the coefficients A_n and B_n are given by the expressions

$$A_n = A(p_n) = \frac{(1 + \alpha + \ddot{a}a/\dot{a}^2)}{2\alpha a^\alpha}, \tag{6.5}$$

$$B_n = B(p_n) = \frac{1}{\alpha^2\dot{a}^2a^{2\alpha+1}}. \tag{6.6}$$

In this chapter I will justify the use of this approximation. Following Hockney and Eastwood (1988). I evaluate the performance of this approximation under four headings, consistency, accuracy, stability and efficiency. I will then go to consider the inclusion of multiple timesteps[1] within MLAPM.

6.1 Consistency

Are equations (6.3) and (6.4) consistent with the original continuous equations (6.1) and (6.2)? In the limit of a vanishingly small timestep the discrete approximations should tend to the continuous equations. Let $\Delta p \to 0$ in the manner of elementary calculus. Equation (6.4) is easy

$$\lim_{\Delta p\to 0}\frac{\mathbf{x}_{n+1} - \mathbf{x}_n}{\Delta p} = \lim_{\Delta p\to 0}\frac{\Delta\mathbf{x}}{\Delta p} \tag{6.7}$$

$$= \frac{d\mathbf{x}}{dp}. \tag{6.8}$$

[1] By multiple timesteps I refer to particles within the system not all moving with the same size timestep

Hence as required

$$\frac{d\mathbf{x}}{dp} = \mathbf{u}. \tag{6.9}$$

Equation (6.3) is a little more involved, first re-arrange it in the form

$$\frac{\mathbf{u}_{n+1/2}(1 + A_n\Delta p) - \mathbf{u}_{n-1/2}(1 - A_n\Delta p)}{\Delta p} = \frac{\mathbf{F}^i B_n}{m^i}, \tag{6.10}$$

where the superscripts refer to particle i. Next let $\Delta p \to 0$ as before

$$\begin{aligned}
& \lim_{\Delta p\to 0} \tfrac{\mathbf{u}_{n+1/2}(1+A_n\Delta p)-\mathbf{u}_{n-1/2}(1-A_n\Delta p)}{\Delta p} \\
= & \lim_{\Delta p\to 0} \left(\tfrac{\mathbf{u}_{n+1/2}-\mathbf{u}_{n-1/2}}{\Delta p} + A_n(\mathbf{u}_{n+1/2} + \mathbf{u}_{n-1/2})\right) \\
= & \lim_{\Delta p\to 0} \left(\tfrac{\Delta \mathbf{u}}{\Delta p} + A(2\mathbf{u} + O(\Delta p))\right) \\
= & \tfrac{d\mathbf{u}}{dp} + 2A\mathbf{u} \\
= & \tfrac{\mathbf{F}_i B}{m_i}.
\end{aligned} \tag{6.11}$$

Hence our discrete approximations (6.3) and (6.4) are consistent with the underlying continuous equations (6.1) and (6.2).

6.2 Accuracy: Sources of Error

The reader will recall the extensive discussion of errors in discrete approximations in chapter 4. The multi-grid method is focussed on the reduction of error to a level below the *truncation error*. Here we are also concerned with truncation error as we again have a discrete approximation to a set of differential equations. It is clear that the size of the timestep will play a role in determining the size of the truncation error. I will now proceed to show that the discrete approximation (6.3) is second-order accurate, ie. the truncation error $\epsilon \propto (\Delta p)^2$.

Firstly let us eliminate the velocity between the two equations of motion.

$$\begin{aligned}
\frac{B_n\mathbf{F}_n}{m_i} &= \tfrac{\mathbf{u}_{n+1/2}(1+A_n\Delta p)-\mathbf{u}_{n-1/2}(1-A_n\Delta p)}{\Delta p} \\
&= \tfrac{(\mathbf{x}_{n+1}-\mathbf{x}_n)(1+A_n\Delta p)-(\mathbf{x}_n-\mathbf{x}_{n+1})(1-A_n\Delta p)}{\Delta p^2} \\
&= \tfrac{1}{\Delta p^2}\left((\mathbf{x}_{n+1} - 2\mathbf{x}_n + \mathbf{x}_{n-1}) + A_n\Delta p(\mathbf{x}_{n+1} - \mathbf{x}_{n-1})\right) \\
&= \tfrac{\mathbf{x}_{n+1}-2\mathbf{x}_n+\mathbf{x}_{n-1}}{\Delta p^2} + A_n\tfrac{\mathbf{x}_{n+1}-\mathbf{x}_{n-1}}{\Delta p}.
\end{aligned} \tag{6.12}$$

Similarly with the continuous equations I obtain

$$\frac{d^2\mathbf{x}}{dp^2} + 2A(p)\frac{d\mathbf{x}}{dp} = \frac{B(p)\mathbf{F}_i}{m_i}. \tag{6.13}$$

Let $\mathbf{X}$ be a solution of equation (6.13). The truncation error at time-level n , $\epsilon_{\mathbf{n}}$, is given by

$$\frac{\mathbf{X}_{n+1} - 2\mathbf{X}_n + \mathbf{X}_{n-1}}{\Delta p^2} + A_n\frac{\mathbf{X}_{n+1} - \mathbf{X}_{n-1}}{\Delta p} = \frac{\mathbf{F}(\mathbf{X}_n)B_n}{m} - \epsilon_n. \tag{6.14}$$

Taylor expand $\mathbf{X}_{n+1}$ and $\mathbf{X}_{n-1}$ about $p = p_n$

$$\begin{aligned}
\mathbf{X}_{n+1} &= \mathbf{X}(p_n + \Delta p) \\
&= \mathbf{X}(p_n) + \Delta p\tfrac{d\mathbf{X}}{dp}\Big|_{p=p_n} + \Delta p^2\tfrac{d^2\mathbf{X}_n}{dp^2}\Big|_{p=p_n} + \cdots
\end{aligned} \tag{6.15}$$

$$\begin{aligned} \mathbf{X}_{n-1} = & \quad \mathbf{X}(p_n - \Delta p) \\ = & \quad \mathbf{X}(p_n) - \Delta p \tfrac{d\mathbf{X}}{dp}\big|_{p=p_n} + (\Delta p)^2 \tfrac{d^2\mathbf{X}_n}{dp^2}\big|_{p=p_n} - \cdots . \end{aligned} \tag{6.16}$$

Substituting into equation (6.14) gives

$$\frac{\mathbf{F}(\mathbf{X}_n)B}{m} - \epsilon_n = 2A\frac{d\mathbf{X}}{dp}\Big|_{p=p_n} + \frac{d^2\mathbf{X}}{dp^2}\Big|_{p=p_n} \frac{(\Delta p)^2}{3}\frac{d^3\mathbf{X}}{dp^3} + \frac{(\Delta p)^2}{12}\frac{d^4\mathbf{X}}{dp^4} + O((\Delta p)^3). \tag{6.17}$$

Subtracting the continuous equation I obtain an expression for the truncation error,

$$\begin{aligned} \epsilon_n = & \quad -\tfrac{(\Delta p)^2}{3}\left(\tfrac{d^3\mathbf{X}}{dp^3} + \tfrac{1}{4}\tfrac{d^4\mathbf{X}}{dp^4}\right), \\ |\epsilon_n| \propto & \quad (\Delta p)^2. \end{aligned} \tag{6.18}$$

Another source of errors is rounding error. Computers store numbers in finite blocks of memory leading to a limit in the precision of numbers. Usually roundoff errors are much smaller than truncation errors and can be ignored as long as the integration scheme is stable, i.e. errors do not grow with time.

6.3 Stability

Do the inevitable truncation and roundoff errors increase as the integration progresses? If the errors grow and not decay, no matter how small they are initially they will eventually dominate the real solution. Following the treatment of Hockney and Eastwood, (1988). I will now consider the propagation of errors. Consider the roundoff error $\mathbf{e}_n$ at time-level n, arising solely from an error at some earlier time-level k. Let $\tilde{\mathbf{X}}_n$ be a solution of the difference equations in the absence of roundoff error. Assuming that roundoff error occurs only once at time level k, we may write

$$\begin{aligned} & \tfrac{(\tilde{\mathbf{X}}_{n+1}+\mathbf{e}_{n+1})-2(\tilde{\mathbf{X}}_n+\mathbf{e}_n)+(\tilde{\mathbf{X}}_{n-1}+\mathbf{e}_{n-1})}{\Delta p^2} + A_n \tfrac{(\tilde{\mathbf{X}}_{n+1}+\mathbf{e}_{n+1}-\tilde{\mathbf{X}}_{n-1}-\mathbf{e}_{n-1})}{\Delta p} \\ = & \quad \tfrac{B_n \mathbf{F}_n(\tilde{\mathbf{X}}_n+\mathbf{e}_n)}{m}. \end{aligned} \tag{6.19}$$

Given that $\tilde{\mathbf{X}}$ is a solution of the difference equations I obtain

$$\frac{\tilde{\mathbf{X}}_{n+1} - 2\tilde{\mathbf{X}}_n + \tilde{\mathbf{X}}_{n-1}}{\Delta p^2} + A_n\frac{(\tilde{\mathbf{X}}_{n+1} - \tilde{\mathbf{X}}_{n-1})}{\Delta p} = \frac{B_n\mathbf{F}_n(\tilde{\mathbf{X}}_n)}{m}. \tag{6.20}$$

Subtracting equation (6.20) from equation(6.19) gives

$$\frac{\mathbf{e}_{n+1} - 2\mathbf{e}_n + \mathbf{e}_{n-1}}{\Delta p^2} + A_n\frac{(\mathbf{e}_{n+1} - \mathbf{e}_{n-1})}{\Delta p} = \frac{B_n\mathbf{F}_n(\tilde{\mathbf{X}}_n + \mathbf{e}_n)}{m} - \frac{B_n\mathbf{F}_n(\tilde{\mathbf{X}}_n)}{m}. \tag{6.21}$$

Next Taylor expand the force term $\mathbf{F}_n(\tilde{\mathbf{X}}_n + \mathbf{e}_n)$ about $\tilde{\mathbf{X}}_n$

$$\frac{\mathbf{e}_{n+1} - 2\mathbf{e}_n + \mathbf{e}_{n-1}}{\Delta p^2} + A_n\frac{(\mathbf{e}_{n+1} - \mathbf{e}_{n-1})}{\Delta p} = \frac{B_n}{m}\mathbf{e}_n(\nabla\cdot\mathbf{F}|_{x=\tilde{\mathbf{X}}_n}). \tag{6.22}$$

I consider only the worst case so I replace $\nabla\cdot\mathbf{F}|_{x=\tilde{\mathbf{X}}_n}$ with its maximum negative value $-|\nabla\cdot\mathbf{F}|_{max}$ where in doing so I have assumed that the solution of equation (6.22) is oscillatory. At this point I will simplify matters by considering each component of the error vector $\mathbf{e}$ separately. The *error propagation* equation for the ith component becomes (suppressing the index for clarity)

$$\frac{e_{n+1} - 2e_n + e_{n-1}}{\Delta p^2} + A_n\frac{e_{n+1} - e_{n-1}}{\Delta p} = -\frac{B_n|\nabla\cdot\mathbf{F}|_{max}}{m}. \tag{6.23}$$

As this equation is linear in e_n an appropriate trial solution is one of the form

$$e_n = \lambda^n = \exp(i\omega n\Delta p). \tag{6.24}$$

Substituting this in equation (6.23) and dividing throughout by λ^{n-1} gives

$$\frac{\lambda^2-2\lambda+1}{\Delta p^2}+A_n\frac{\lambda^2-1}{\Delta p}=-\frac{B_n\lambda|\nabla\cdot\mathbf{F}|_{max}}{m}. \tag{6.25}$$

To simplify further I will now drop the second term on the left hand side of equation (6.25) under the assumption that ${A_n}^2 \ll B_n|\nabla\cdot\mathbf{F}|_{max}$ which I will justify later. Equation (6.25) now becomes

$$\frac{\lambda^2-2\lambda+1}{\lambda}=-\frac{B_n|\nabla\cdot\mathbf{F}|_{max}}{m}\Delta p^2. \tag{6.26}$$

The two solutions of this equation can now be obtained

$$\lambda\pm=1-\frac{\Gamma}{2}\pm\frac{\Gamma}{2}\sqrt{1-\frac{4}{\Gamma^2}}, \tag{6.27}$$

where

$$\Gamma=\frac{B_n|\nabla\cdot\mathbf{F}|_{max}\Delta p^2}{m}. \tag{6.28}$$

With a general solution of the form

$$e_n=a{\lambda_+}^n+b{\lambda_-}^n, \tag{6.29}$$

where the both the coefficients and solutions are complex.

For the integration scheme to be stable the moduli of the two quantities λ_+ and λ_- must be less than or equal to one. To see under what conditions this is true consider the locus of the quantities $\lambda_\pm$ on the complex plane. Returning to equation (6.25) this may be written in the form

$$\frac{(\lambda-1)(\lambda-1)}{\lambda}=-\Gamma. \tag{6.30}$$

Consider the extrema of Γ when the timestep in infinite and zero, substituting $\Gamma=\infty$ and $\Gamma=0$ into equation (6.27) gives

$$\begin{array}{rcl} \lambda_+=\lambda_-=1 & \text{when} & \Gamma\ =0, \\ \lambda_+=0,\quad \lambda_-=-\infty & \text{when} & \Gamma\ =\infty. \end{array} \tag{6.31}$$

The path the roots λ take to go between these extrema will determine where the stability boundary lies. Take the real and imaginary parts of the logarithm of equation (6.30)

$$Re \qquad 2\ln|\lambda-1|-\ln|\lambda| \;=\; \ln\Gamma, \tag{6.32}$$

$$Im \qquad 2\arg(\lambda-1)-\arg\lambda \;=\; \pi. \tag{6.33}$$

The second equation gives the locus. The sum of the angles subtended by lines drawn to the zeros at $\lambda=1$ minus the angle subtended by the pole at the origin must equal π. From equation (6.24) we see that if λ is complex, then there are two complex conjugate roots with modulus one. The above expression tells us there can be no roots on the real positive axis apart from at $\lambda=1$. For $\lambda>1$ all the roots and poles of Γ lie to the left and hence subtends zero angle, while for $0<\lambda<1$ the two roots of Γ at $\lambda=1$ contribute 2π. Hence λ_+ and λ_- must travel round the opposite sides of the unit circle from $\lambda=1$ to $\lambda=-1$. The point $\lambda=-1$ is the only place where the roots can leave the unit circle. Indeed roots are possible along the whole of the negative real axis as the pole of Γ at the origin contributes π radians. Hence the stability boundary is to be found at the point $\lambda=-1$. Substituting this into equation (6.27) gives

$$\begin{array}{rcl} \Gamma & = & 4, \\ \left(\dfrac{B_n|\nabla\cdot\mathbf{F}|_{max}}{m}\right)^{\frac{1}{2}}\Delta p & = & 2. \end{array} \tag{6.34}$$

giving finally the stability limitation on the size of timestep

$$\Delta p\leq 2\left(\frac{m}{B_n|\nabla\cdot\mathbf{F}|_{max}}\right)^{\frac{1}{2}}. \tag{6.35}$$

Assumption

Recall the assumption I made earlier,

$$A_n{}^2 \ll B_n|\nabla \cdot \mathbf{F}|_{max}$$

I will now justify the use of the assumption and show how it allows the term in A_n in equation (6.25) to be dropped. Consider the relative sizes of the two terms

$$\frac{-B_n|\nabla \cdot \mathbf{F}|_{max}\lambda\Delta p^2}{m}. \tag{6.36}$$

and

$$-A_n(\lambda^2 - 1)\Delta p. \tag{6.37}$$

The latter term can be ignored if

$$\frac{B_n|\nabla \cdot \mathbf{F}|_{max}\lambda\Delta p^2}{m} \gg A_n\Delta p, \tag{6.38}$$

where I have taken both λ and $\lambda^2 - 1$ to be of order unity. Hence

$$A_n \ll \frac{B_n|\nabla \cdot \mathbf{F}|_{max}\lambda\Delta p}{m}. \tag{6.39}$$

From the stability criterion, equation (6.35),

$$\Delta p \sim \left(\frac{m}{B_n|\nabla \cdot \mathbf{F}|_{max}}\right)^{\frac{1}{2}}. \tag{6.40}$$

Substituting this into equation (6.39) gives

$$A_n{}^2 \ll B_n|\nabla \cdot \mathbf{F}|_{max}. \tag{6.41}$$

Hence if A_n satisfies this expression the term with A_n as coefficient may be neglected.

6.4 Efficiency

As always there is a trade off here between the accuracy of the integration and computer storage and time. To increase the accuracy of the integration a higher-order integration scheme could be used. A higher order scheme would enable the use of a larger timestep, at the expense of keeping data from more time-levels. It is generally accepted that the above second-order accurate leapfrog scheme is the most efficient for cosmological applications. It is also worth pointing out that when multiple timesteps are used, the book keeping required to hold more time-levels of data becomes prohibitive.

6.5 Multiple Timesteps

Recall the limit on the timestep size (6.35). The reader will note that the maximum size for the timestep is inversely proportional to the force gradient. This leads immediately to a problem. MLAPM is capable of high force resolution, which also means big force gradients and hence strict limits on the size of each timestep. How does one approach this problem?

One possibility would be for each particle to have a separate individually tailored timestep. This is not practical in a grid-based code, as a new potential calculation would be required for each particle move. A second possibility would be to have a single timestep for all particles based on the largest timestep admissible for all particles. This is not practical either as only a small percentage

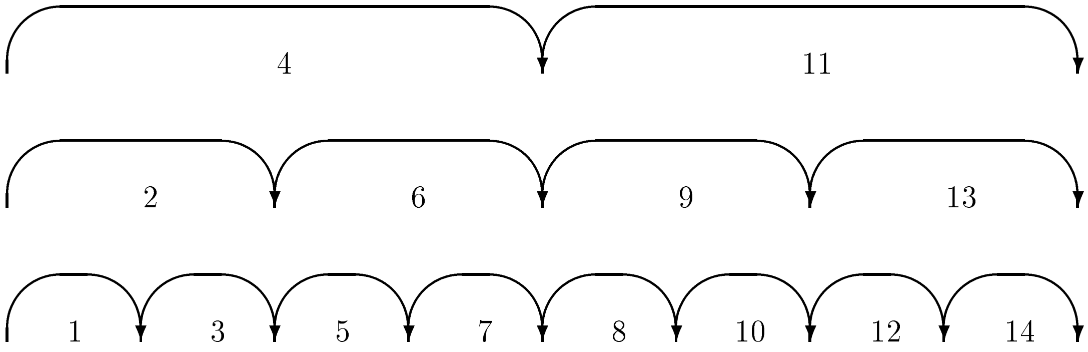


Figure 6.1: Ordering of timesteps in MLAPM, with three levels of grid.

of the particles will require such a small timestep for accurate integration. As I am aiming for a situation where operations on particles dominate the CPU time, adding timesteps to a run is not a good idea.

A third approach to time integration is available. The size of the force gradient is related to how fine a grid is being used. Only in refined regions will small timesteps be needed. Hence use a series of timesteps, one for each level of refinement. It will be highly desirable for the particles to be brought back into synchronization at frequent intervals, hence the timestep on each level should be an integer multiple of the timestep on the next finer level. This third approach offers both accurate integration and efficiency and has been implemented in MLAPM.

6.5.1 Organizing Multiple Timesteps

A multiple timestep scheme presents many challenges to the program designer. With multiple timesteps, the three previously distinct parts of the N-body code, potential solver, mass assignment and time integration must now be integrated. To facilitate the integration, a simple scheme must be chosen. Using the approach described in the previous section, this is achieved by using the same factor by which the grids are divided, two, as the factor by which a timestep is divided when going from a coarse grid to the next finer one.

How should the timesteps be ordered? The obvious approach would be to process each pair of timesteps on a fine grid before processing the larger timestep on the next coarser grid. This is not a good approach, as the fine grid particles are always being moved ahead of the coarse grid particles. This could lead to spurious forces on coarse-grid particles. For example consider particles falling into a dense region. The centre of the region is refined while the outer part is not. If the particles in the refined region are always moved in advance of the coarse grid particles, the coarse grid particles will "see" the over-density as being bigger than it is.

Minimizing Time-Asynchronization

What is needed is an ordering scheme that reduces any time asynchronization to a minimum. Such a scheme was proposed by Porter in his thesis (Porter, 1985) and has been adapted and implemented in MLAPM. In this scheme a coarse grid timestep may only be taken when all finer grid timesteps are synchronized at the halfway point of the coarse grid timestep. The scheme is illustrated in figure (6.1). In this way fine grid particles move once before and once after each timestep on the next coarser grid.

6.5.2 Time-Reversibility

Earlier I noted that time-reversibility was a desirable property of an integration scheme. Now I have moved to the use of multiple timesteps and multiple grids does the time-reversible nature of the leapfrog integrator still hold?

Consider the case of a set of grids of increasing refinement with associated particles. The particles may not move between grids and use the timestep scheme discussed above. In this case we have no problem, a particle timestep can be unwound in the same way that particles on a single grid could have their timestep unwound. In this simple situation the scheme is time-reversible.

Now let the particles move between grids. Now we see difficulties, a particle moving from a coarse grid to a fine grid will have a larger timestep than a particle moving from a fine grid to a coarse one. As it stands this situation is not time-reversible. Firstly let me point out that in general particles will not often leave high density refined regions, and thus the effects of not being time reversible will be limited. Secondly the problem could be eliminated by identifying particles that are about to leave a fine grid and move these particles with a coarse grid timestep. In practice this would be difficult to implement and of questionable benefit.

6.5.3 Velocity and Position Timesteps

The scheme described above in figure 6.1 refers only to the particle positions. If we use the simple leapfrog scheme the velocities will be offset in the usual way from the position timesteps (see equation (6.3)). There are no problems with this offset until one considers the rebuilding of the adaptive grid and the moving of particles from one grid level and hence from one timestep to another. It is easy to appreciate that unless the velocities and positions of all the particles at a given grid level are synchronized, errors in integration will result. Hence the leapfrog scheme is re-written slightly to ensure that the positions and velocities are synchronized properly (Quinn et al., 1997). The unavoidable cost of this is to replace the single position update with two updates.

Chapter 7

MLAPM: A Description of the Source Code

In this chapter I present a complete description of code that comprises the MLAPM package. The chapter begins with a description of the header files used by MLAPM. I next discuss general utility functions used through out the code. I then discuss the three key segments of MLAPM's operation, adaptive grid generation, the multi-grid potential solver and the integration of the particles' equations of motion. The theoretical aspects of each of these were discussed in chapters 5, 4, and 6 respectively. These three aspects of MLAPM are discussed in turn in §7.3, §7.4 and §7.5. The chapter concludes with a description of the controlling functions and input/output functions.

7.1 Header Files

This section contains a description of the header file `param.h`. The other main header file `tdef.h` was discussed in §5.5. These two headers are `#include`d by all source files. Each individual *.c* file also has an associated header file containing the declarations of functions used elsewhere in the code. Thus these header files provide the interface between different parts of the code. As these "interface" files contain no actual code they will not be discussed further.

7.1.1 `param.h`

`X`, `Y` and `Z`

At several points in the code the x, y and z directions are explicitly used (i.e. outside of a loop over an array). This directions are placed in the 0, 1 and 2 slots respectively, of all arrays in MLAPM. To aid readability the parameters `X`, `Y` and `Z` are defined to represent these numbers.

```
#define X 0     /* x-coord symbol */
#define Y 1     /* y-coord symbol */
#define Z 2     /* z-coord symbol */
```

Boolean parameters

To match up with the definition of a Boolean data type in `tdef.h` two boolean values are defined. The "positive" response is given the numerical value 1 and the "negative" response the value 0.

```
#define TRUE 1
#define FALSE 0
```

Grid Parameters

The grid hierarchy is formed into two blocks, a set of grids covering the whole computational box (domain grids) that are created at the start of the simulation, and a set of adaptive grids formed during the simulation. The size of the coarsest and the finest domain grids is set by two parameters

```
#define MINDL1D 2     /* minimum domain grid size (in 1-dim) */
#define MAXDL1D 32    /* maximum domain grid size (in 1-dim) */
```

An additional parameter `SHIFT` is also defined. This parameter is given by $1/(2*\texttt{MAXDL1D})$ but it is stated explicitly because of heavy usage. This parameter represents the coordinates of the first node on all adaptive grids. This parameter is used extensively by the particle relocation functions.

Refinement Parameters

The refinement signal is generated at any node if the density there satisfies

$$density > \rho_{crit} \times m_{crit}^{(n-n_f)} \tag{7.1}$$

where ρ_{crit} is the critical density defined by `CRITDENS`, m_{crit} is the density multiplier defined by `CRITMULTI`, n is the number of grid cells and n_f is the number of grid cells on the finest domain grid. The total number of grids is limited by a further parameter, `MAXNOGRIDS`. These two parameters are defined in `param.h`.

```
#define CRITDENS 1.0  /* refinement criterion on finest domain grid */
#define CRITMULTI 8.0 /* multiplying factor for CRITDENS = 2^NDIM */
#define MAXNOGRIDS 7  /* maximum number of grids */
```

Multi-grid Solver Parameters

As was noted in chapter 4, the multi-grid solver performs a number of Gauss-Seidel relaxation sweeps both prior to moving to a courser grid and immediately after returning from one. The number of sweeps in each case is controlled by the two parameters `PRE` and `POST`.

```
#define PRE 3  /* no. of sweeps prior to coarse grid approx. */
#define POST 3 /* no. of sweeps after coarse grid correction */
```

Other Parameters

The parameter `NDIM` is a reference to the number of spatial dimensions used by code.

```
#define NDIM 3 /* NB. This does not set the number of dimensions! */
```

This parameter does not control the number of dimensions MLAPM operates in, it is merely used to make the source more readable. It is however possible that future versions of the code will contain this feature.

In chapter 6 the parameter p was defined as the exponent of a in the definition of the time variable. In MLAPM the parameter p is given the name `P_EXPON`.

```
#define P_EXPON 1.5 /* Exponent of a in expression for p */
```

7.2 Utility Functions: `utility.c`

There are several small functions used throughout the code. These functions are to be found in the source file `utility.c`. The five utility functions are described below.

`modulo`

To account for the periodicity of the domain grids when searching for nodes an integer modulo function is provided. This function simply subtracts or adds the size of the grid to bring a grid coordinate index back within the computational box. Note that no support is provided for grid indices which after the addition or subtraction of the size of the grid still remain outside the grid. This would indicate an error.

```
long modulo(long a, long  b)
{
  if(a < 0)
    return a + b;
  if(a >= b)
    return a - b;
  else
    return a;
}
```

`set_opposite`

The function `set_opposite` is the boolean NOT operator for the boolean type.

```
boolean set_opposite(boolean tf)
{
  if(tf == TRUE)
    return FALSE;
  else
    return TRUE;
}
```

`is_even`

The reader will recall from chapter 4 that Gauss-Seidel iteration is usually performed in a chess-board pattern, with nodes being processed alternately. On adaptive grids exactly where the Gauss-Seidel sweep must start is determined by whether the first node is even or odd. This is checked by the `is_even` function.

```
boolean is_even(long a)
{
  int b = 2;

  if((fmod(a,b)) > 0.0)
    return FALSE;
  else
    return TRUE;
}
```

`test_tsc`

The function `test_tsc` test if any of the nodes in the supplied three dimensional node pointer array are missing (ie the pointer is `NULL`).

```
boolean test_tsc(nptr tsc_nodes[3][3][3])
{
  int i,j,k;

  for(i = 0; i < 3 ; i++)
    for(j = 0; j < 3 ; j++)
      for(k = 0; k < 3 ; k++)
        if(tsc_nodes[i][j][k] == NULL)
```

```
            return FALSE;
    return TRUE;
}
```

`freen`

The free function is a wrapper function for the standard library function `free`. The function ensures that the pointer argument is set to `NULL`.

```
void freen(void **ptr)
{
  free(*ptr);
  *ptr = NULL;
}
```

`del_squared`

The operator ∇^2 is required at several points within the code and is defined here for convenience.

```
float del_squared(gridls *cur_grid, nptr cur_node, diff_star_ptr dstar_ptr)
{
  float del_sqrd;         /* result */

  /* calculate del_sqrd */
  del_sqrd = (dstar_ptr->nleft->pot + dstar_ptr->nright->pot +
      dstar_ptr->nfor->pot + dstar_ptr->nback->pot +
      dstar_ptr->nup->pot + dstar_ptr->ndown->pot -
      6.0 * cur_node->pot) / cur_grid->grid_spac_sqrd;
  return del_sqrd;
}
```

`resid`

The residual is required by the multi-grid solver to determine convergence. The reader will recall that the residual is defined by

$$e = \rho - \nabla^2 \phi \tag{7.2}$$

```
float resid(gridls *cur_grid, nptr cur_node,diff_star_ptr dstar_ptr)
{
  float residual;

  residual = cur_node->dens - del_squared(cur_grid, cur_node, dstar_ptr);
  return residual;
}
```

`calc_a_p()` **and** `calc_b_p()`

These two functions calculate the coefficients A(p) and B(p) which are given by equations (6.5) and (6.6).

```
double calc_a_p(double a, double a_dot, double a_ddot)
{
  return (1.0 + P_EXPON + (a_ddot * a) / (a_dot * a_dot)) /
    (2 * P_EXPON * pow(a, P_EXPON));
}

double calc_b_p(double a, double a_dot)
{
  return 1. / (P_EXPON * P_EXPON * a_dot * a_dot * pow(a, (2 * P_EXPON + 1)));
}
```

7.3 Grid Generation and Upkeep

The data structure used by MLAPM was described fully in chapter 5. This section describes how that data structure is created and maintained during the evolution of a simulation. The code divides easily into two parts, the simple code that generates the domain grids (see §7.3.2), and the vastly more complex code that generates and maintains the adaptive grid structure. Adaptive grid maintenance is made more complex still by the use of multiple time-steps (see §6.5) which requires the grid to be updated in step with the time-step cycle.

This section deals first with the cover functions for calloc() which allocate blocks of memory for the standard `structs`. Next I deal with domain grid generation. I will then move on to discuss the creation and destruction of adaptive grids in both the single and multiple time-step modes of operation.

7.3.1 Memory Allocation: `alloc_struct.c`

`alloc_struct.c` contains a series of wrapper functions for the standard library memory allocation functions `calloc(), realloc()` and `free()`. For each of the major structure types some or all of the direct calls to the memory allocation routines are replaced by a separate function call. There are several reasons behind this. Firstly it is more convenient to to make a function call

```
pptr = c_part(no_part);
```

than to use a call like:

```
if((pptr = (partptr) calloc(no_part, sizeof(part))) == NULL){
  perror("c_part: error callocing particles \ n");
  exit(1);
}
```

on each occasion a block of structures must be created.

The wrapper functions give a greater opportunity to use systematic error handling procedures. The standard library functions `malloc()`, `realloc()` and `calloc()` all return a NULL pointer if the request for additional memory cannot be satisfied. If this is the case the code must deal with the situation. In the current version of MLAPM (2.0) this is simply dealt with by sending an error message to the user through the standard input/output library function `perror` and calling `exit()` which brings the operation of the code to a halt. Care must thus be taken to limit the number of grids so that the available memory on the heap is not exceeded. In future versions of MLAPM the behavior of the code when encountering this error could be enhanced.

The Dynamic Memory Handling functions

Definitions for all the following functions are to be found in `alloc_struct.c`.

```
extern partptr c_part(long block_size);

extern cqptr r_cquad(cqptr cquad_ptr, long old_size, long new_size);
extern nqptr r_nquad(nqptr nquad_ptr, long old_size, long new_size);
extern nptr r_node(nptr node_ptr, long new_size);

extern pqptr c_pquad(long block_size);
extern cqptr c_cquad(long block_size);
extern nqptr c_nquad(long block_size);
extern nptr c_node(long block_size);

extern void dest_node(nptr *ptr);
extern void dest_nquad(nqptr *ptr);
extern void dest_cquad(cqptr *ptr);
extern void dest_pquad(pqptr *ptr);
```

I will describe in detail the functions `c_nquad`, `r_nquad` and `dest_node`. All the other functions have very similar operation as can be seen by examining the source files.

`c_nquad`

```
partptr c_part(long block_size)
```

The argument block_size is the number of nquads to allocate.

```
{
  partptr pptr;     /* pointer to new particles */
  partptr i_pptr;   /* pointer to increment down block of particles */
  int i;            /* coordinate index (X,Y,Z) */

  /* generate block_size particles */

  if((pptr = (partptr) calloc(block_size, sizeof(part))) == NULL){
    perror("c_part: error callocing particles\n");
    exit(1);
  }
```

The `calloc(n, sizeof(object))` function is a C library function (from `stdlib.h`) that allocates storage space for n objects of size `sizeof(object)` or returns a `NULL` pointer if there is insufficient heap space available.

```
  /*
   * loop over nodes filling pot and dens with 0.0 and setting linked-list
   * pointer to NULL.
   */

  for(i_pptr = pptr; i_pptr < pptr + block_size;
      i_pptr++){
    for(i = X; i <= Z; i++){
      i_pptr->pos[i] = 0.0;
      i_pptr->mom[i] = 0.0;
    }
    i_pptr->ll = NULL;
        i_pptr->stepsize = 0.0;
  }
  return pptr;
}
```

The function `c_nquad` allocates a specified number of nquads that are consecutive in memory and hence can be individually accessed by incrementing a pointer. The function returns a pointer to the first nquad in the block. The function also ensures all data members of each individual nquad are initialized to zero. If the call to the memory allocation function `calloc` fails, `c_nquad` prints an error message and then terminates the program. All the `c_...` functions operate in the same way.

`r_nquad`

```
nqptr r_nquad(nqptr nquad_ptr, long old_size, long new_size)
{
  nqptr new_nquad_ptr;

  /* re-size block of nquads to new_size */

  if((nquad_ptr = (nqptr) realloc(nquad_ptr, new_size *
  sizeof(nquad)))
```

```
     == NULL){
    perror("r_nquad: error reallocing nquad\n");
    exit(1);
  }
```

The realloc(*pointer, newsize * sizeof(object)*) function is a standard C library function that changes the size of the dynamically allocated memory at the location pointed at by *pointer* to *newsize * sizeof(object)*. If memory cannot be allocated the function returns NULL. The contents of the memory is preserved up to the minimum of the old and new sizes. The function returns a pointer to the location of the allocated memory if successful.

```
  /* If there are new nquads initialize them */
  if(new_size > old_size){
    for(new_nquad_ptr = nquad_ptr + old_size; new_nquad_ptr < nquad_ptr
          + new_size; new_nquad_ptr++){
      new_nquad_ptr->loc = NULL;
      new_nquad_ptr->x = 0;
      new_nquad_ptr->length = 0;
      new_nquad_ptr->next = NULL;
    }
  }
  return nquad_ptr;
}
```

The r_nquad function changes the number of nquads in the block to new_size (typically increases the number in the block by one). Any new memory is initialized to zero. If the request for more nquads cannot be satisfied the function prints an error message and terminates the program.

dest_node

```
void dest_node(nptr *ptr)
{
  free(*ptr);
  *ptr = NULL;
}
```

The function dest_node frees the block of nodes at the location ptr. The function sets the pointer ptr to NULL, ensuring that the memory cannot be accessed after it has been de-allocated, a common mistake.

7.3.2 Domain grid generation

The first thing that must be said about domain grids is that once created they are never destroyed until the end of the program. Domain grids cover the whole computational box and hence none of the *QUAD* structures use their next pointers. Thus domain grids are simple to create. The function that creates each domain grid is gen_dom_grid(), which is found in the source file gen_dom_grid.c. The quads and nodes are allocated by the calloc() wrapper functions c_pquad, etc, described in the above section. This function is illustrated in full below with annotations.

```
long gen_dom_grid(gridls *dom_grid, long size)
{
  pqptr dom_pquad;          /* domain-grid pquad pointer */
  cqptr dom_cquad;          /* domain-grid cquad pointer */
  nqptr dom_nquad;          /* domain-grid nquad pointer */

  /*
   * fill in grid_list details
```

```
   */

  dom_grid->l1dim = size;           /* enter grid size */
  dom_grid->l2dim = size * size;    /* enter size^2 */
  dom_grid->old_resid = 0.0;        /* set all residuals to zero */
  dom_grid->cur_resid = 0.0;

  /* calculate the grid spacing */

  dom_grid->grid_spac =  1 / (float) dom_grid->l1dim;
  dom_grid->grid_spac_sqrd = dom_grid->grid_spac * dom_grid->grid_spac;
  /*
   * create the domain grid
   */

  dom_grid->gridptr = c_pquad(1);   /* generate pquad */
  dom_pquad = dom_grid->gridptr;    /* set dom_pquad */
  dom_pquad->length = size;         /* set length to size */
  dom_pquad->loc = c_cquad(size);   /* generate cquads */
```

As this is a domain grid only one pquad is required and then *size* cquads are needed with *size* nquads per cquad and so on. The coordinate of the start of each quad is always zero, as this is the default initialization provided by the "constructor" the assignment to zero is not repeated.

```
  /* loop over cquads */
  for(dom_cquad = dom_pquad->loc; dom_cquad < dom_pquad->loc + size;
      dom_cquad++){
    dom_cquad->length = size;       /* set length to size */
    dom_cquad->loc = c_nquad(size); /* generate nquads */

    /* loop over nquads */
    for(dom_nquad = dom_cquad->loc; dom_nquad < dom_cquad->loc + size;
        dom_nquad++){
      dom_nquad->length = size;         /* set length to size */

      /* generate size nodes for each nquad */
      dom_nquad->loc = c_node(size);
    }
  }
  return size;
}
```

7.3.3 Adaptive Grid Generation: gen_adaptive_grid.c

The generation of adaptive grids is handled by the function `gen_adaptive_grid()`. The whole process of creating a new grid is handled by this function. `gen_adaptive_grid.c` begins by creating a new entry in the grid_list, an array of structures of type `gridls`. The `gridls` structures, as the reader will recall from chapter 5, contain all the relevant information associated with each individual grid including a pointer to the first pquad that forms the grid. Next the main refinement routine, `ref_grid` is called. `ref_grid()` sweeps through the old, coarser grid identifying regions that satisfy the conditions required for refinement (i.e. the density is above the critical density for refinement). The refinement functions create the new grid as the sweep proceeds. The `ref_grid` hierarchy of functions is described in the next section, 7.3.4.

If the call to `ref_grid` has produced refinement, `gen_adaptive_grid` proceeds to set up the new grid ready for the multi-grid calculation (and prior to subsequent calls to `gen_adaptive_grid` from the controlling function `step()` which loops over the grid generation routine until refinement is no longer needed). Setting up the grid involves several steps. Firstly particles which are currently stored in linked-list on the coarse grid but which now lie within the boundaries of the new sub-grid must be moved to linked lists with the new grid. This process is controlled by the function `re_link` which can be found in the source file `re_link.c`. The *re-linking* of particles is one of the most complex operations in the code and stretches over several source files. This process is described in §7.3.6.

The next operation to be called from `gen_adaptive_grid` is `store_dens`. This function is only called in the multiple time-step mode of operation. The function stores the old values of the density which are required later. For a full discussion of the need for this see the section on the `step.c` control routine §7.6.1.

The remaining function calls in `gen_adaptive_grid` are common to both modes of operation. The first of these is the mass assignment routine `assign`. The function `assign` (found in `assign.c`) uses triangular-shaped cloud mass assignment to convert the particle masses to the grid density distribution. Alongside assign is the function `mass_to_dens` which completes the job assign starts by subtracting the average density in the box from every nodes' density. This is required by the Poisson equation in co-moving coordinates (see equation (3.13)). The mass assignment routines are described in §7.3.9.

Finally there are two sets of calls to the function `f_to_c_pquad` each enclosed in a loop over the pquads. The function `f_to_c_pquad` is part of the *fine-to-coarse* interpolation function hierarchy. Fine to coarse interpolation will formally be discussed as part of the multi-grid solver section of this chapter as it was originally developed for that process. However, density interpolation was found to be required to set up the densities correctly after new grids had been generated. The two functions passed to `f_to_c_pquad` actually do the processing while the *f_to_c_* functions are merely control routines. The first of the two, `f_to_c_node_reverse` corrects the fine grid densities to include density which spills into the fine grid region from particles in coarse-grid cells neighbouring the fine grid region. The second set of calls, passing the function `f_to_node_dens_only`, interpolates the fine grid density back to the coarse grid. A full explanation of the need for these operations can also be found in §7.3.9.

```
extern float idens;
extern gridls *dom_grid;
```

These two external variables are used in the mass assignment and routines respectively. They are set here but defined in the appropriate source file.

⋮

```
/*
 * gen_adaptive_grid: test current grid for refinement criteria,
 * generate adaptive sub-grid if required, re-link particles and adjust
 * densities as necessary.
 */
boolean gen_adaptive_grid(gridls **grid_list, int *no_grids)
{
  gridls *new_grid;     /* new-grid pointer                         */
  boolean refined;      /* refined YES/NO flag                      */
  pqptr t_fin_pquad;    /* to loop over ll of fine pquads           */
  static int counter;

  /* reallocate the grid list array */
  if(((*grid_list) = (gridls *) realloc(*grid_list, (*no_grids + 1) *
                                        sizeof(gridls))) == NULL){
    perror("gen_adaptive_grid:error reallocating grid list");
    exit(1);
  }
```

The first step is to create a new grid-list entry ready to accept the new grid data. If the `ref_grid()` produces no refinement the grid list is reduced to its original size.

```
  /* fill in new grid entry */
  new_grid = *grid_list + *no_grids;
  new_grid->critdens = (new_grid - 1)->critdens * CRITMULTI;
```

The critical density on the new grid is given by multiplying that of the previous grid by the factor CRITMULTI. The default setting for this value is 8 as the volume of cells on the new grid will be 1/8 of the volume of those on the current grid. The remaining settings are either derived from the settings of the current grid or are simply initialized to zero.

```
  new_grid->l1dim = (new_grid - 1)->l1dim * 2;
  new_grid->l2dim = new_grid->l1dim * new_grid->l1dim;
  new_grid->old_resid = 0.0;
  new_grid->cur_resid = 0.0;
  new_grid->grid_spac = 1 / (float) new_grid->l1dim;
  new_grid->grid_spac_sqrd = new_grid->grid_spac *
    new_grid->grid_spac;
  new_grid->counter = 0.0;
  new_grid->a = (new_grid - 1)->a;
  new_grid->time_level = 0;
  new_grid->u = 0.0;
  new_grid->old_t = 0.0;
  new_grid->new_t = 0.0;

  /* now call refinement routine */
  if((refined = ref_grid(new_grid, (new_grid - 1))) == YES){

    /* temp hack */
    /*store_dens(new_grid - 1);*/

    /* now relink particles to new grid */
    re_link((new_grid - 1), new_grid);

#ifdef MULTISTEP
    /* if in multiple timestep mode, store density on "old_grid" */
    store_dens(new_grid - 1);
#endif

    /* now assign mass */
    assign(new_grid);
    mass_to_dens(new_grid);
    for(t_fin_pquad = new_grid->gridptr; t_fin_pquad != NULL;
        t_fin_pquad = t_fin_pquad->next)
      f_to_c_pquad((new_grid - 1), (new_grid - 1)->gridptr, new_grid,
                   t_fin_pquad,
                   (void (*)(gridls *, pqptr, cqptr, nqptr, nptr, gridls
                             *, pqptr, cqptr, nqptr, nptr, long,
                             long, long))(f_to_c_node_reverse));
    idens = 0.0;
    /* temporary hack to take out density interpolation */
    for(t_fin_pquad = new_grid->gridptr; t_fin_pquad != NULL;
        t_fin_pquad = t_fin_pquad->next)
      f_to_c_pquad((new_grid - 1), (new_grid - 1)->gridptr, new_grid,
                   t_fin_pquad,
                   (void (*)(gridls *, pqptr, cqptr, nqptr, nptr, gridls
                             *, pqptr, cqptr, nqptr, nptr, long,
                             long, long))(f_to_c_node_dens_only));

    *no_grids += 1;    /* update number of grids */
```

⋮

```
  }else
    /* realloc gridls back to original size */
    if(((*grid_list) = (gridls *) realloc(*grid_list, (*no_grids) *
                                        sizeof(gridls))) == NULL){
      perror("gen_adaptive_grid:error reallocating grid list");
      exit(1);
    }
  dom_grid = find_dom_grid(*grid_list);
  return refined;
}
```

7.3.4 Refinement: ref_grid.c

Testing for refinement and building a new grid is response to the results of those tests is controlled by the functions `ref_grid`, `ref_pquad`, `ref_cquad` and `ref_nquad`. All these functions are defined in the source file `ref_grid.c` and of these functions only `ref_grid.c` is externally visible (via `ref_grid.h`). These functions loop through the structure of the "old" grid calling the density test routine on all the nodes. If the density test proves positive the grid structure is built. I will describe the operation of three of these functions, `ref_grid`, `ref_pquad` and `ref_nquad`. The function `ref_cquad` is nearly identical to `ref_pquad` and its operation can be inferred from a discussion of that function.

`ref_grid`

The function `ref_grid` is the main controlling function for the refinement process. The function creates the first pquad on the new grid and passes it to the next function in the hierarchy, `ref_pquad`. As all the subsequent parts of the structure must be referenced through this pquad it is necessary to create the pquad before any refinement tests are performed. This will turn out to be necessary throughout the grid hierarchy. If no refinement is needed the pquad is destroyed.

```
boolean ref_grid(gridls *new_grid, gridls *old_grid)
{
  pqptr old_pquad;        /* pquad pointer - old grid */
  pqptr new_pquad;        /* pquad pointer - new grid */
  pqptr rem_pquad;        /* pquad pointer - pointer to pquads that will be   */
                          /* destroyed */
  boolean refined;        /* refinement flag */

  /*
   * The new grid requires a pquad be created to other pquads and blocks of
   * cquads may be created and referenced should the refinement routine require
   * them.
   */

  new_grid->gridptr = c_pquad(1);  /* create a pquad */
  new_pquad = new_grid->gridptr;   /* define the old_pquad and new_pquad      */
  old_pquad = old_grid->gridptr;   /* to point to the first pquads on the old */
                                   /* and new grids respectively */
```

If refinement occurs the function `ref_pquad` returns the boolean value `TRUE`, otherwise `FALSE` is returned. If refinement of the grid has occurred, the last pquad in the linked-list of pquads is tested to see if it actually in use. If not it is destroyed. The last pquad (and in fact the last cquad etc) may be unused as the refinement functions ref_pquad are recursive. The function `ref_pquad` creates a pquad before entering the next recursive level. Hence if the last pquad created is not used by the refinement routine it must be removed. If refinement does not occur, and `FALSE` is returned, the pquad we initially created must be removed.

```
  /*
   * call ref_pquad. If refinement occurs ref_pquad returns the boolean value
   * TRUE or FALSE otherwise.
   */
  if((refined = ref_pquad(old_grid, old_pquad, new_pquad)) == TRUE){

    /*
     * Refinement has occurred, free potential excess pquad at end of pquad
     * linked list. Loop to end of linked list and test "loc" pointer to test if
     * any cquads are linked to the pquad.
     */
    if(new_pquad->next != NULL){
      for(rem_pquad = new_pquad; (rem_pquad->next)->next != NULL;
          rem_pquad = rem_pquad->next)
        ;
      if((rem_pquad->next)->loc == NULL){
        dest_pquad(&(rem_pquad->next));
      }
    }
  }else /* no refinement hence destroy new_pquad */
    dest_pquad(&(new_grid->gridptr));
  return (boolean)refined;  /* return refinement flag */
}
```

ref_pquad

The function `ref_pquad` is the controlling routine for the refinement process on the pquad level. Essentially its purpose is to loop over the block of cquads described by the supplied coarse pquad, testing for refinement by calling the `ref_cquad` function. However, there are several complications. The function is illustrated below with many annotations.

```
boolean ref_pquad(gridls *curgrid, pqptr coapquad, pqptr ifinpquad)
```

Three arguments are required a pointer to the coarse (or current) grid, a pointer to the coarse pquad we are testing and a pointer to the new fine pquad ready to accept the new grid data.

```
{
  cqptr icoacquad;     /* pointer to coarse cquad, loop pointer */
  long z;              /* z-coord of current coarse node */
  long zlen;           /* length of current refinement (z-direction) */
  boolean refined;     /* boolean refinement flag */
  boolean refever;     /* boolean refinement flag - recursive flag, have any  */
                       /* previous calls to ref_pquad produced refinement */
```

Refinement may not occur inside this coarse pquad but may occur on other recursive calls to this function. The flag `refever` keeps track of this and if `refined == FALSE` but `refever == TRUE` TRUE is returned by the function as refinement has occurred somewhere in the grid.

```
  cqptr i_fine_cq;     /* current fine cquad */
  boolean state;       /* on/off state flag - indicates whether current fine  */
                       /* cquad is cospatial or not */
```

As can be seen from the code below the two cases of the fine cquads being cospatial and non-cospatial with the coarse cquad are treated differently. I will postpone the discussion of this until I describe the functions which do the actual testing of nodes. This `state` flag keeps track of whether the fine cquad being tested is co-spatial or not.

```
  cqptr rem_cquad;     /* pointer to excess cquads - for removal loop */

  /* initilise state flags */
```

```
refined = FALSE;
refever = FALSE;
state = FALSE;

/* initialise zlen */
zlen = 0;
```

The long integer `zlen` keeps track of the length of the current refinement. This is what the length part of the fine pquad will be set equal to.

```
/*
 * The new pquad requires a cquad be created. Further cquads can be added and
 * blocks of nquads may be created and referenced as the refinement routine
 * requires them.
 */
ifinpquad->loc = c_cquad(1);   /* create first cquad */
i_fine_cq = ifinpquad->loc;    /* initialize i_fine_cq to point to new cquad*/
```

As noted above all the functions in the *ref_* hierarchy require a fine quad be created before entering the function.

```
/* loop over the coarse planes */
for(z = coapquad->z, icoacquad = coapquad->loc; icoacquad <
    coapquad->loc + (coapquad->length - 1); z++, icoacquad++){

  /* test cospatial planes */

  /* call ref_cquad, if refinement occurs free excess cquads */
  if(ref_cquad(curgrid, coapquad, icoacquad, i_fine_cq,
               (boolean (*) (gridls*, pqptr, cqptr, nqptr, nptr, long, long,
                             long))(test31_pl6),
               (boolean (*) (gridls*, pqptr, cqptr, nqptr, nptr, long, long,
                             long))(test31_pl4), z) == TRUE){
    if(i_fine_cq->next != NULL){  /* check "next" pointer not NULL */

      /* find the last cquad along this plane */
      for(rem_cquad = i_fine_cq; (rem_cquad->next)->next != NULL ;
          rem_cquad = rem_cquad->next)
        ;
      if((rem_cquad->next)->loc == NULL){/* if loc pointer is NULL, destroy */
        dest_cquad(&(rem_cquad->next));
        rem_cquad->next = NULL;
      }
    }
    if(state == FALSE){
      refever = TRUE;            /* refinement has occured on this plane */
      ifinpquad->z = ( z * 2);   /* fill in start of refinement */
      state = TRUE;              /* refinement state now TRUE - indicates    */
      /* refinement occurring */
    }
    zlen++;                      /* increase length by one */
```

If the test produces a `TRUE` result but the state is `FALSE`, this indicates that a refinement block has just been started. Hence the initial details such a starting coordinate must be entered and the state flags reset.

```
    /* create another cquad in this block */
    ifinpquad->loc = r_cquad(ifinpquad->loc, zlen, (zlen + 1));
    i_fine_cq = ifinpquad->loc + zlen; /* set i_fine_pq to point to new cquad */
```

Again here we create a new cquad ready to be passed to the function ref_cquad. However, here that cquads must be allocated as a block (i.e. we only have a pointer to the first in the block). Hence we use the cover function for realloc, r_cquad.

```
      /* test intermediate plane */

      /* call ref_cquad, if refinement occurs free excess cquads */
      if(ref_cquad(curgrid, coapquad, icoacquad, i_fine_cq,
                   (boolean (*) (gridls*, pqptr, cqptr, nqptr, nptr, long, long,
                                 long))(test2l_pl6),
                   (boolean (*) (gridls*, pqptr, cqptr, nqptr, nptr, long, long,
                                 long))(test2l_pl4), z) == TRUE){
        if(i_fine_cq->next != NULL){  /* check "next" pointer not NULL */

          /* find the last cquad along this plane */
          for(rem_cquad = i_fine_cq; (rem_cquad->next)->next != NULL ;
              rem_cquad = rem_cquad->next)
            ;
          if((rem_cquad->next)->loc == NULL){/* if loc pointer is NULL, destroy */
            dest_cquad(&(rem_cquad->next));
            rem_cquad->next = NULL;
          }
        }
        zlen++;                     /* increase length by one */

        /* create another cquad */
        ifinpquad->loc = r_cquad(ifinpquad->loc, zlen, (zlen + 1));
        i_fine_cq = ifinpquad->loc + zlen;
      }else{  /* no refinement, current block has finished, set state to FALSE */
        state = FALSE;     /* state is FALSE */

        /* remove the last cquad in block - unused */
        ifinpquad->loc = r_cquad(ifinpquad->loc, zlen + 1, zlen);
        ifinpquad->length = zlen;     /* fill in length of block */
        ifinpquad->next = c_pquad(1); /* add a linked pquad */
        ifinpquad = ifinpquad->next;  /* move to linked pquad */
        ifinpquad->loc = c_cquad(1);  /* create first cquad */
        i_fine_cq = ifinpquad->loc;   /* reinitialize i_fine_cq */
        zlen = 0;                     /* reset zlen */
```

Here we see the conclusion a refinement block. When the test on the intermediate plane fails the end cquad is removed by another call to r_cquad. The remaining details of the fine pquad are filled in and a new pquad is created and linked via the next pointer.

Now consider the loop over the cquads again. The loop does not include the last coarse cquad as this must be treated differently. There are four possible situations, dealt with by an `if - else` block. In turn these are:

1. The current pquad is not the last covering the computational domain, and the refinement flag indicates a refined region is being generated.
2. The current pquad is the last pquad in the computational domain with ref_cquad producing refinement.
3. Neither of the above are true but the state flag indicates we are in the middle of a refinement block and it must be ended.
4. All other possibilities, here we delete the unused cquad.

```
      }
    }
  }
```

```
/*
 * Deal with last coarse cquad on plane. The following If-Else statement deals
 * with the complicated problem of the last coarse cquad on the plane. There
 * are four possibilities:
 * 1. The current pquad is not the last covering the computational domain, and
 *    the refinement flag indicates a refined region is being generated.
 * 2. The current cquad is the last cquad in the computational domain with
 *    ref_cquad producing refinement.
 * 3. Neither of the above are true but the state vector is TRUE and so a
 *    refined region must be ended.
 * 4. All other possibilities - delete unused cquad
 */
if(coapquad->next != NULL){
  if(state == TRUE){  /* call ref_cquad to generate refinement */
    ref_cquad(curgrid, coapquad, icoacquad, i_fine_cq,
              (boolean (*) (gridls*, pqptr, cqptr, nqptr, nptr, long, long,
                            long))(test3l_pl6),
              (boolean (*) (gridls*, pqptr, cqptr, nqptr, nptr, long, long,
                            long))(test3l_pl4), z);
    if(i_fine_cq->next != NULL){ /* check "next" pointer not NULL */

      /* find the last cquad along this plane */
      for(rem_cquad = i_fine_cq; (rem_cquad->next)->next != NULL ;
          rem_cquad = rem_cquad->next)
        ;
      if((rem_cquad->next)->loc == NULL){/* if loc pointer is NULL, destroy */
        dest_cquad(&(rem_cquad->next));
        rem_cquad->next = NULL;
      }
    }
    zlen++;                        /* increase zlen by one*/
    ifinpquad->length = zlen;      /* fill in length of refinement */
    ifinpquad->next = c_pquad(1); /* create a linked fine pquad */
    ifinpquad = ifinpquad->next;
  }else
    dest_cquad(&(ifinpquad->loc));/* destroy unused cquad */

  /* call ref_pquad recursively */
  refined = ref_pquad(curgrid, coapquad->next, ifinpquad);
```

The above `if` statement handles the fist of the cases listed above. The coarse pquad being handled by `ref_pquad` is not the last pquad in the computational box as is indicated by the "next" pointer not being NULL. In this situation we must recursively call `ref_pquad` on that next pquad but first we must end the refinement block. This is done by calling the function `ref_cquad` on the last cquad of the block. The return value is thrown away as it is irrelevant, if refinement occurred on the intermediate plane between the two last coarse cquads then refinement will occur on this plane. After the refinement has been done the pquad data members are filled in and a new pquad allocated for the recursive function call.

```
}
else if((coapquad->z + coapquad->length == curgrid->l1dim) &&
        (ref_cquad(curgrid, coapquad, icoacquad, i_fine_cq,
                   (boolean (*) (gridls*, pqptr, cqptr, nqptr, nptr, long,
                                 long, long))(test3l_pl6),
                   (boolean (*) (gridls*, pqptr, cqptr, nqptr, nptr, long,
                                 long, long))(test3l_pl4), z) == TRUE)){

  /* we are actually at the boundary of the box */
  if(i_fine_cq->next != NULL){ /* check "next" pointer not NULL */
```

```
      /* find the last cquad along this plane */
      for(rem_cquad = i_fine_cq; (rem_cquad->next)->next != NULL ;
          rem_cquad = rem_cquad->next)
        ;
      if((rem_cquad->next)->loc == NULL){/* if loc pointer is NULL, destroy */
        dest_cquad(&(rem_cquad->next));
        rem_cquad->next = NULL;
      }
    }
    if(state == FALSE){    /* refined region starts here (possible because of */
      /* triply-periodic boundary conditions */
      refever = TRUE;      /* set refever flag */
      ifinpquad->z = (z * 2); /* enter z coordinate into pquad */
    }
    zlen++;

    /* create another cquad */
    ifinpquad->loc = r_cquad(ifinpquad->loc, zlen, (zlen + 1));
    i_fine_cq = ifinpquad->loc + zlen; /* set i_fine_cq to point to new cquad */
    if(ref_cquad(curgrid, coapquad, icoacquad, i_fine_cq,
                 (boolean (*) (gridls*, pqptr, cqptr, nqptr, nptr, long, long,
                               long))(test2l_pl6),
                 (boolean (*) (gridls*, pqptr, cqptr, nqptr, nptr, long, long,
                               long))(test2l_pl4), z) == TRUE){
      /* test plane between "edges" */

      if(i_fine_cq->next != NULL){ /* check "next" pointer not NULL */

        /* find the last cquad along this plane */
        for(rem_cquad = i_fine_cq; (rem_cquad->next)->next != NULL ;
            rem_cquad = rem_cquad->next)
          ;
        if((rem_cquad->next)->loc == NULL){/* if loc pointer is NULL, destroy */
          dest_cquad(&(rem_cquad->next));
          rem_cquad->next = NULL;
        }
      }
      zlen++;
    }else /* an extra cquad has been created - destroy it */
      ifinpquad->loc = r_cquad(ifinpquad->loc, zlen +1, zlen);
    ifinpquad->length = zlen;   /* fill in the length */
```

This part of the if block deals with the periodic case, i.e. the coarse cquad runs up to the edge of the grid. The test for refinement is performed in the usual way. Next we see the same `if(state == FALSE)...` block as was found in the main cquad loop. This deals with the possibility that a refined region can occur starting at the "far" edge of the grid and straddle across to the "front" edge of the grid. On the new fine grid a cquad could exist between the coarse cquad at the boundary of the box and the first cquad in the box, hence a test for refinement is performed.

```
  }else if(state == TRUE){ /* conclude the current refinement */
    ref_cquad(curgrid, coapquad, icoacquad, i_fine_cq,
              (boolean (*) (gridls*, pqptr, cqptr, nqptr, nptr, long, long,
                            long))(test3l_pl6),
              (boolean (*) (gridls*, pqptr, cqptr, nqptr, nptr, long, long,
                            long))(test3l_pl4), z);
    if(i_fine_cq->next != NULL){ /* check "next" pointer not NULL */

      /* find the last cquad along this plane */
      for(rem_cquad = i_fine_cq; (rem_cquad->next)->next != NULL ;
          rem_cquad = rem_cquad->next)
```

```
          ;
        if((rem_cquad->next)->loc == NULL){/* if loc pointer is NULL, destroy */
          dest_cquad(&(rem_cquad->next));
          rem_cquad->next = NULL;
        }
      }
      ifinpquad->length = ++zlen;   /* fill in the length */
```

Given the above two parts of the if block, this case resolves to the coarse pquad being the last but not running up to the boundary and the state being TRUE indicating that the loop was creating a new refinement block when it exited. We must generate any remaining refinement on this last cospatial cquad by calling ref_cquad (again the return value is ignored). The refinement block is then concluded in the usual way.

```
    }
    else
      dest_cquad(&(ifinpquad->loc));      /* destroy unused cquad */
```

Here we have the last catch-all case. It resolves to the coarse pquad being the last on the grid but not running up to the boundary and the new cquad is not part of an existing refinement block (the state is FALSE). In this situation all we need do is clean up the unused fine cquad.

```
  return (refever || refined);
}
```

`ref_nquad`

The function `ref_nquad` operates in a similar way to `ref_pquad`, however there are important differences and hence I now discuss it separately. This function is the last in the *ref_* hierarchy, as such it no longer calls a separate function on each of the nodes it creates. The grid nodes are fundamental in a sense, as they do not contain pointers to further elements of the grid structure. As a result of this each block of nodes is allocated together at the end of the function and not in situ via the use of `realloc` as we have seen in `ref_nquad`.

Another feature which the reader will have notices is the passing to the function of a test function. In `ref_pquad` the four functions `test3l_pl6`, `test3l_pl4`, test2l_pl6 and `test2l_pl4` appeared being passed to `ref_cquad`. These are the functions which test the nodes. The explanation for why four are required can be found in the next section §7.3.5. The correct function is passed to `ref_nquad` which it then uses to test for refinement on each of the nodes.

```
boolean ref_nquad(gridls *curgrid, pqptr coapquad, cqptr coacquad, nqptr
                  coanquad, nqptr ifinnquad,
                  boolean (*test)(gridls*, pqptr, cqptr, nqptr, nptr, long, long,
                                  long), long z, long y)
{
  nptr icoanode;               /* pointer to coarse node */
  long x;                      /* x-coord of current coarse node */
  long xlen = 0;               /* length of current refinement (x-direction) */
  boolean state = FALSE;       /* on/off refinement state indicator */
  boolean refined = FALSE;     /* test result variable */
  boolean refever = FALSE;     /* have we ever refined along this column */
  nqptr icoanquad;             /* pointer to coarse cquad */

  x = coanquad->x;      /* set x coord to first node */
  if(x == 0){           /* first node is on edge - account for periodicity */

    /* find last coarse nquad */
    for(icoanquad = coanquad; icoanquad->next != NULL ; icoanquad  =
        icoanquad->next)
      ;
```

```
    /* if the last node in the last coarse cquad is on the edge of the grid */
    if(icoanquad->x + icoanquad->length == curgrid->l1dim){
      icoanode = icoanquad->loc + (icoanquad->length - 1);
      x = curgrid->l1dim - 1;
      /* and if test indicates refinement needed */
      if(test(curgrid, coapquad, coacquad,icoanquad, icoanode , z, y, x)
         == TRUE){
        /* start a refinement */
        state = TRUE;
        refever = TRUE;
        (ifinnquad)->x = 0;
      }
      x = 0;
```

The above section of code deals explicitly with the first coarse node. If this coarse node lies on the boundary, a fine node will need to be created if refinement is generated by the corresponding boundary node at the other edge of the grid. This is what is being tested for here. Firstly we test if the coarse node lies on the lower x-boundary. Next we test if a coarse node lies on the upper x-boundary, if this node exits we test it with the `test` function. If the test is positive the refinement block is started and a count of the nodes needed kept.

This feature was not seen in the code above for `ref_pquad`. The node test are arranged slightly differently. In the functions `ref_pquad` and `ref_cquad`, refinement “tests” are performed for all the new quad structure locations. As we shall see in the code below only the cospatial node locations are tested, if the test results in refinement two nodes are generated. One of these new nodes lies is co-spatial with the coarse node and one lies beyond it. If the node test proves negative but the state is TRUE, one additional node is added and the refinement concluded. On the first node we to know if the state is TRUE or FALSE and this is what is being tested for in this section.

```
    }
  }
  /* loop over coarse nodes */

  for(icoanode = coanquad->loc; icoanode < coanquad->loc +
      (coanquad->length - 1); x++, icoanode++){

    /* test the coarse node for refinement */
    if(test(curgrid, coapquad, coacquad, coanquad, icoanode, z, y,
            x) == TRUE){
      if(state == FALSE){
        refever = TRUE;
        state = TRUE;
        (ifinnquad)->x = (x * 2); /* fill in start of refinement */
      }
      xlen += 2;              /* add the next two nodes to length */
    }
    /* only state indicates refine - we must be concluding a finnquad */

    else if(state == TRUE){
      (ifinnquad)->length = (++xlen); /* fill in length of refinement */
      ifinnquad->loc = c_node(xlen);  /* create the correct no nodes */
      state = FALSE;
      (ifinnquad)->next = c_nquad(1); /* gen. new linked nquad */
      (ifinnquad) = (ifinnquad)->next;/* move to linked nquad */
      xlen = 0;      /* reset xlen */
```

As described above we see that the main loop contains three main possibilities. The coarse node can generate refinement which results in two more nodes being add to the count of nodes to be created. The coarse node may not generate refinement but the state may be TRUE, in this case one additional node is requested and the refinement block is concluded. The nodes are created in a single call to c_node. Finally if refinement is not indicated and the state is FALSE we do nothing but move on to the next coarse node in the block described by the coarse nquad.

```
    }
  }
  /* last node described by coarse cquad */

  if(coanquad->next != NULL){           /* there is a linked coarse cquad */
    if(state == TRUE){                 /* conclude current finnquad */
      xlen++;
      (ifinnquad)->length = (xlen);      /* fill in length of refinement */
      ifinnquad->loc = c_node(xlen);     /* generate nodes */
      (ifinnquad)->next = c_nquad(1);    /* gen. new linked nquad */
      (ifinnquad) = (ifinnquad)->next;   /* move to linked nquad */
    }
    /* call recursively */

    refined = ref_nquad(curgrid, coapquad, coacquad, coanquad->next, ifinnquad,
                        (boolean (*)(gridls*, pqptr, cqptr, nqptr, nptr, long,
                                     long, long))(test), z, y);
```

Here we see the start of an if block dealing with the last coarse node. The if block is very similar to that found at the end of ref_pquad and the three cases may be translated directly across. No default catch-all case is included as there is no cleaning up of unused structures to be done.

```
  }
  else if((coanquad->x + coanquad->length == curgrid->l1dim) &&
          (test(curgrid, coapquad, coacquad, coanquad, icoanode, z, y, x)
           == TRUE)){
    if(state == FALSE){               /* new refinement */
      refever = TRUE;                /* we have refined */
      (ifinnquad)->x = (x * 2);/* fill in start of refinement */
    }
    xlen += 2;

    /* now conclude the finnquad */
    ifinnquad->loc = c_node(xlen);
    (ifinnquad)->length = xlen;
    ifinnquad->next = NULL;
  }
  else if(state == TRUE){      /* conclude refinement */
    (ifinnquad)->length = ++xlen;
    ifinnquad->loc = c_node(xlen);
    ifinnquad->next = NULL;
  }
  return (refever || refined);
}
```

7.3.5 Node Density Tests - denstest.c

Testing the nodes to see if they are over the critical density for the current grid is central to how a new refined grid is created. All we need do is compare the density with the critical density that is found in the gridlist structure [1]. In practice this is complicated by the requirement for a *buffer* zone. By buffer zone I refer to a set of nodes where refinement would not be generated by virtue of the mass density they have but by the fact that neighbouring nodes have densities over the critical density, this is illustrated in the figure 7.1.

Why do we need to have this buffer zone? Consider what happens to the range of densities on a new grid as compared to the same region on the old, coarser grid. In general the densities

[1] The critdens member of the C struct, see §5.5

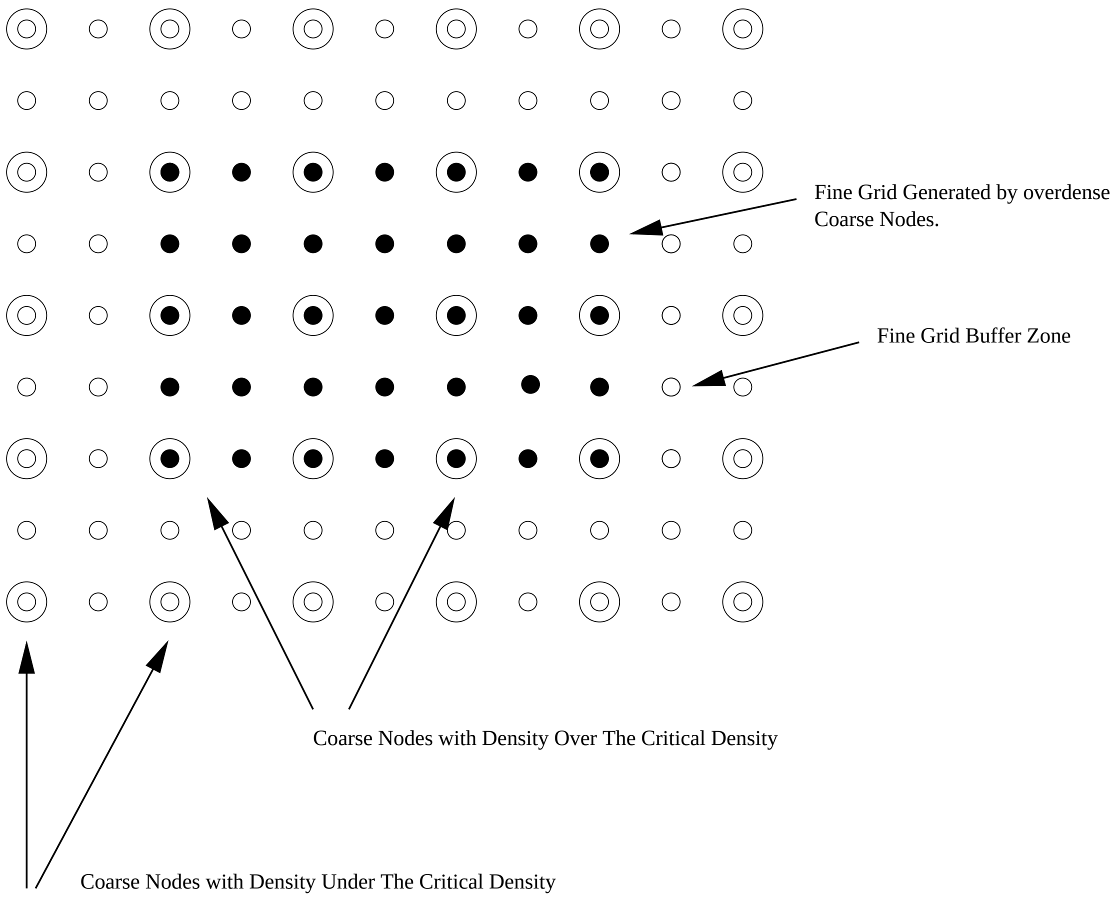


Figure 7.1: *The Buffer Zone.* A buffer zone of extra fine nodes is created around those otherwise generated to smooth the transition between grids

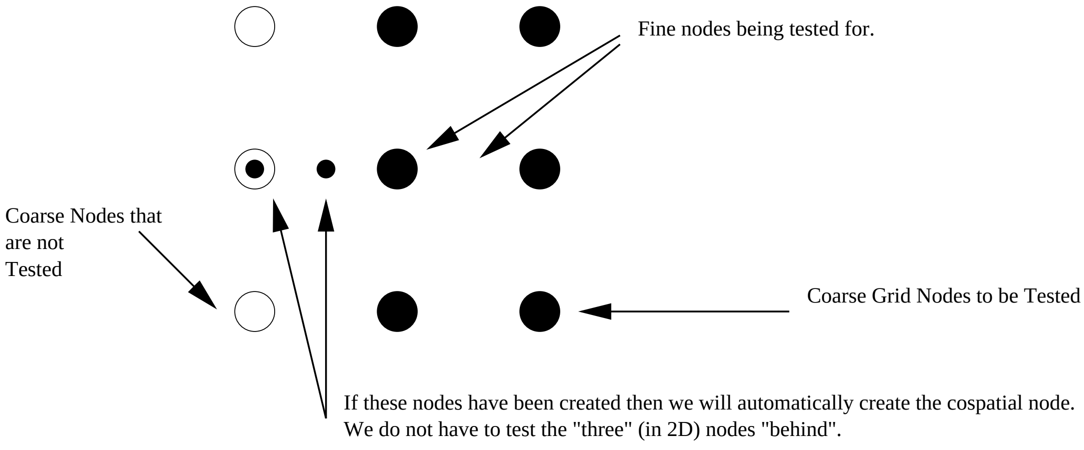


Figure 7.2: *2D Representation of Tested Nodes.* In the cospatial case 9 nodes out of 27 can be omitted as a result of the way `ref_nquad` operates.

found on the fine grid will show a wider range as the mass assignment algorithm will have smoothed them less. As a result of this we could be faced with a sharp change in density and hence a sharp change in forces as we move across the boundary from the old to the new grid. The inclusion of a buffer zone smoothes this transition.

So in practical terms how does this buffer zone affect the density test algorithm? Consider first the simplest case, of testing whether to create a fine grid node that is co-spatial with a coarse grid node. Instead of just testing the coarse grid node the fine grid node will be co-spatial with, we now have to test all the 26 surrounding coarse grid nodes as well. This adds considerable complexity and a large number of operations. However, careful programming can avoid most of the additional computational burden.

9 of the 27 total tests can be eliminated. We do not have to test the plane of nine nodes *behind* the current coarse grid node. We can eliminate these tests as if any of these nodes were over the critical density they would have generated a node co-spatial with the last coarse node along the current column and one in between that and the location we are testing. If the *state*[2] is TRUE then we will generate a node at the co-spatial location whether or not the test for the critical density is true or not. If the state is TRUE the test at the co-spatial node location is in fact to see if we should generate the next intermediate fine node along this column. So what happens if the state is FALSE? In this case we know that none of the "back-plane" nodes have produced any refinement anyway so there is no point in testing to see if we need to add "buffer" nodes. This simplification does not extend to any more dimensions, so we are still left with 18 tests. The above argument is illustrated in figure 7.2.

We have so far discussed the case of testing the node cospatial with a coarse grid nodes. This is implemented in the function `test3l_pl6`. There are three additional cases which I will now outline, these are to be found in the functions `test3l_pl4`, `test2l_pl6` and `test2l_pl4` (the naming convention test*a*l_pl*b* stands for *a* levels and *b* nodes per plane). The locations in the grid relative to a coarse grid node are illustrated in the figure 7.3. In the cases of `test3l_pl4` and `test2l_pl4` the fine node location we are testing lies between two coarse grid columns. As the buffer zones only extend to the next coarse grid node we need not test the entire plane to the right in the 27 node cube. Similarly the cases treated by `test2l_pl6` and `test2l_pl4` lie between two coarse grid planes and no tests need be performed except on those two planes. Hence the "bottom" plane of the original 27 node cube is eliminated.

`test3l_pl6`

```
boolean test3l_pl6(gridls *curgrid, pqptr curpquad, cqptr curcquad, nqptr
```

[2]The state is the refinement on/off state described in the previous section §7.3.4

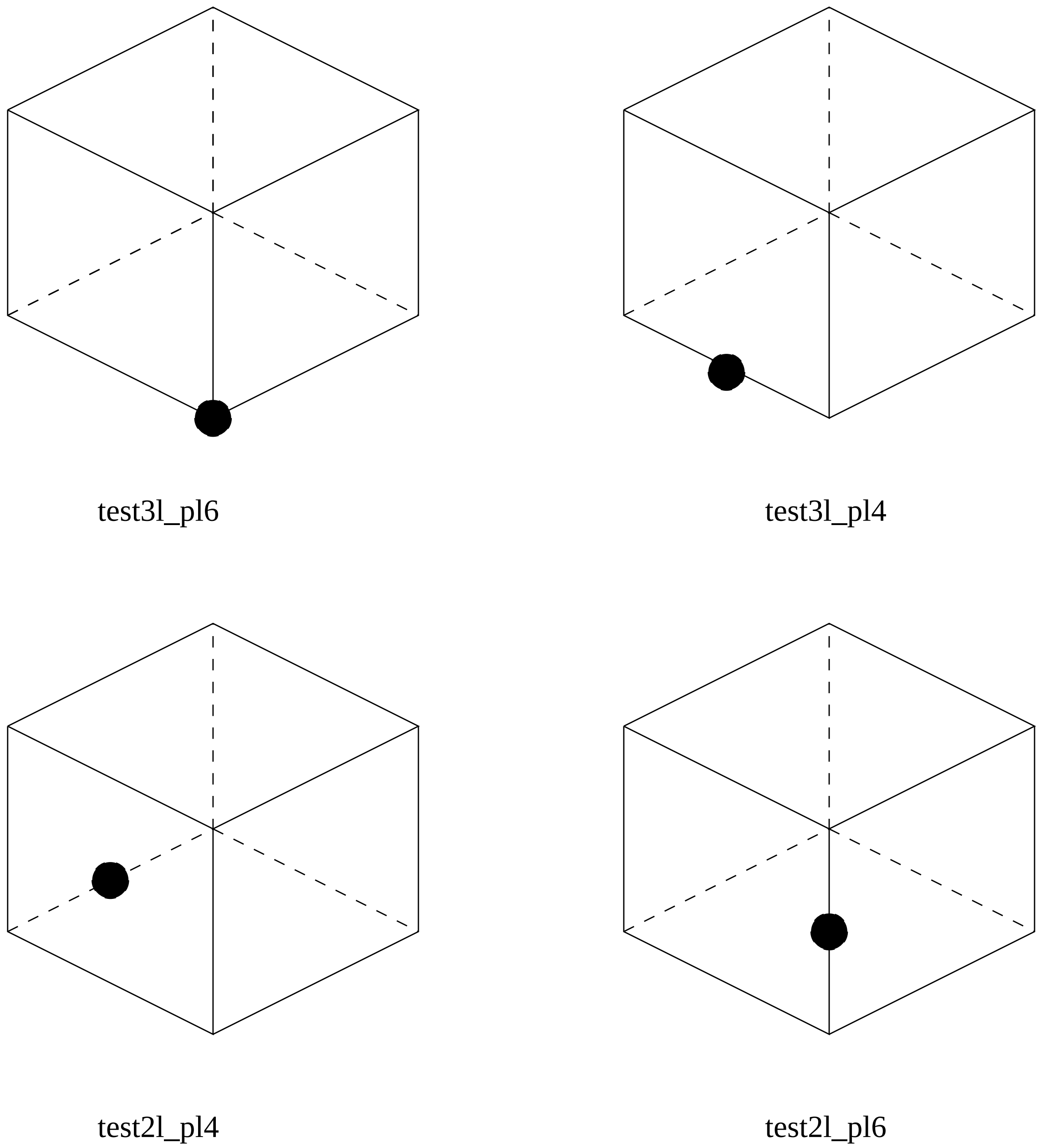


Figure 7.3: The location of the test site for the fine grid node relative to the coarse grid nodes located at the corners of the cube

```
                  curnquad, nptr curnode, long z, long y, long x)
{
  pqptr icurpquad;          /* current coarse pquad */
  cqptr icurcquad;          /* current coarse cquad */
  nqptr icurnquad;          /* current coarse nquad */
  nptr icurnode;            /* current coarse node */
  long endx;        /* end coord of icurnquad */
  long endy;        /* end coord of icurcquad */
  long endz;        /* end coord of icurcquad */

  /* test plane with curnode */
  if(test_pl6(curgrid, curpquad, curcquad, curnquad, curnode, z, y, x) == TRUE)
    return TRUE;

  /* test plane below curnode */
  /* is plane there ? */
  if(z > curpquad->z){

    /* now look for correct cquad */
    /* increment down cquad ll */
    for(icurcquad = (curpquad->loc + (z - 1 - curpquad->z));
        (y > (endy = (icurcquad->y + icurcquad->length)))
          && (icurcquad->next != NULL); icurcquad = icurcquad->next)
      ;
    if((y >=  icurcquad->y) && (y < endy)){ /* is nquad in this cquad */

      /* now look for correct nquad */
      /* increment down nquad ll */
      for(icurnquad = icurcquad->loc + (y - icurcquad->y);
          (x > (endx = (icurnquad->x + icurnquad->length)))
            && (icurnquad->next != NULL); icurnquad = icurnquad->next)
        ;
      if((x >=  icurnquad->x) && (x < endx)){/* is node in this nquad */
        icurnode = icurnquad->loc + (x - icurnquad->x);
        if(test_pl6(curgrid, curpquad, icurcquad, icurnquad, icurnode,
                    (z - 1), y, x) == TRUE)
          return TRUE;
      }
    }
  }
```

In the above fragment we search for the coarse plane below the plane co-spatial with `curnode` in the standard non-periodic case. The code first test that the z coordinate of `curnode` lies within the bounds of the current coarse pquad `curpquad`. If true the code sets the value of the temporary cquad pointer `icurcquad` to point to the correct plane and then moves on to look for the correct nquad. Finding the nquad is a three step process.

1. Find the cquad in the linked-list that has a starting index lower than the y value of `curnode`. Test that the nquad with the correct y value lies within the block described by this cquad. Set `icurcquad` to point to this cquad and set `icurnquad` to point to the nquad.
2. Loop through the nquad linked-list until the x value of `curnode` is greater than that of the starting index of the nquad. Test that the node with coordinate x lies within the block described by this nquad. Set `icurnquad` to point to this nquad and set `icurnode` to point to the correct node.
3. Call the six-points-per-plane function `test_pl6` passing the pquad pointers that point to the *Local* quads (I use *Local* to refer to the quads which contain the node. In some cases it is necessary to pass the pointers to the first quad in the linked list of quads. It is important to remember exactly what is being passed!).

```
  /* periodic case */
```

```
else if(z == 0){

  /* search for correct nquad */
  for(icurpquad = curgrid->gridptr;
      (( curgrid->l1dim - 1) > (endz = (icurpquad->z + icurpquad->length))) &&
        (icurpquad->next != NULL); icurpquad = icurpquad->next)
    ;
  if(((curgrid->l1dim - 1) >= icurpquad->z) &&
     ((curgrid->l1dim - 1) < endz)){   /* is cquad in this pquad */
    for(icurcquad = icurpquad->loc + ((curgrid->l1dim - 1) - icurpquad->z);
        (y > (endy = (icurcquad->y + icurcquad->length )))
          && (icurcquad->next != NULL); icurcquad = icurcquad->next)
      ;
    if((y >= icurcquad->y) && (y < endy)){ /* is nquad in this cquad */
      for(icurnquad = icurcquad->loc + (y - icurcquad->y);
          (x > (endx = (icurnquad->x + icurnquad->length)))
            && (icurnquad->next != NULL); icurnquad = icurnquad->next)
        ;
      if((x >=  icurnquad->x) && (x < endx)){/* is node in this nquad */
        icurnode = icurnquad->loc + (x - icurnquad->x);
        if(test_pl6(curgrid, icurpquad, icurcquad, icurnquad, icurnode,
                    (curgrid->l1dim - 1), y, x) == TRUE)
          return TRUE;
      }
    }
  }
}
```

This fragment deals with the periodic case in the z direction. If z is 0 then the plane below is in fact the last plane at the opposite end of the computational box. Hence the z part of the non-periodic case is replaced by a search for the edge plane while the remainder of the search is identical to the normal case.

```
/* test plane above */
if(z < (curpquad->z + curpquad->length - 1)){
  ⋮
```

Here we deal with the plane above the plane cospatial with `curnode`. Again only the z direction search parts are different. The first test is to see if the plane above is within the bounds of the current coarse pquad.

```
}
/* deal with periodic case */

else if(z == (curgrid->l1dim - 1)){     /* at upper z boundary */

  /* search for correct pquad */
  /* move to first pquad  */
  icurpquad = curgrid->gridptr;
  if(icurpquad->z == 0){  /* if 1st cquad is on boundary */
    for(icurcquad = icurpquad->loc;
        (y > (endy = (icurcquad->y + icurcquad->length )))
          && (icurcquad->next != NULL); icurcquad = icurcquad->next)
      ;
    ⋮
```

In this last part of `test2l_pl6` we deal with the periodic case for the plane above `curnode`. If `curnode` has a z coordinate equal to that of the upper z boundary of this grid then the plane above will be the plane with z = 0 if it exists. The z-search code is modified to look for this plane. The remainder of the code remains the same.

```
  }
  return FALSE;
```

If none of the tests are successful FALSE is returned.

```
}
```

test_pl6

This function starts with the location of the "centre" node on a plane. It finds the neighbouring columns (accounting for the periodicity) and passes control to the column test routine `test_col`.

```
boolean test_pl6(gridls *curgrid, pqptr curpquad, cqptr curcquad, nqptr
                 curnquad, nptr curnode, long z, long y, long x)
{
  cqptr icurcquad;  /* current coarse cquad */
  nqptr icurnquad;  /* current coarse nquad */
  nptr icurnode;    /* current coarse node */
  long endx;        /* xcoord of end of icurnquad */
  long endy;        /* ycoord of end of icurcquad */

  /* Test column with curnode */
  if(test_col(curgrid, curcquad, curnquad, curnode, y, x) == TRUE)
    return TRUE;

  /* test column to left */
  if(y < ((curcquad->y) + (curcquad->length) - 1)){/* next column there? */

    /* search for correct node */
    for(icurnquad = curcquad->loc + (y - curcquad->y) + 1;
        (x > (endx = (icurnquad->x + icurnquad->length)))
          && (icurnquad->next != NULL); icurnquad = icurnquad->next)
      ;
    if((x >=  icurnquad->x) && (x < endx)){   /* is node in this nquad */
      icurnode = icurnquad->loc + (x - icurnquad->x);
      if(test_col(curgrid, curcquad, icurnquad, icurnode, (y + 1), x) == TRUE)
        return TRUE;
```

In a similar fashion to that of the previous function, test3l_pl6, we next treat the non-periodic case of the column to the left of the central column. The format of the search follows.

1. Test if the column is within the current block of nquads by checking the central column is not at the upper boundary of the cquad.
2. Find the enclosing nquad by searching down the linked-list of nquads.
3. If an nquad enclosing the first node on the column for testing is found, pass the node and local quad pointers to `test_col`.

```
    }
    /* deal with periodic case to left */
  }
  else if(y == (curgrid->l1dim - 1)){     /* at y upper boundary */

    /* search for correct node */
    /* move to first cquad along plane */
    icurcquad = curpquad->loc + (z - curpquad->z);
    if(icurcquad->y == 0){  /* if 1st nquad is on boundary */
      for(icurnquad = icurcquad->loc;
```

```
          (x > (endx = (icurnquad->x + icurnquad->length )))
            && (icurnquad->next != NULL); icurnquad = icurnquad->next)
        ;
      ⋮
```

Here we test the periodic case. If the central column is on the upper y boundary the column to the left of it is the nquad at the y index = 0 (if it exists). The search is modified to look for this column, the remainder of the code remains the same.

```
    }

    /* test column to right */
    /* is the next column there ? */
  }
  if(y > curcquad->y){
    ⋮
```

Here we deal with the non-periodic column to the right. If the central column is not the first nquad described by the current cquad then we move on to find the correct node in an identical way to that found in the search for the left column above.

```
  }
  else if(y == 0){    /* at lower y boundary */

    /* search for correct node */
    for(icurcquad = curpquad->loc + (z - curpquad->z);
        (( curgrid->l1dim - 1) > (endy = (icurcquad->y + icurcquad->length))) &&
          (icurcquad->next != NULL); icurcquad = icurcquad->next)
      ;
    if(((curgrid->l1dim - 1) >= icurcquad->y) &&
       ((curgrid->l1dim - 1) < endy)){     /* is nquad in this cquad */
      ⋮
```

This last segment deals with the periodic case to the right. If the central column has the y index 0, then the column to the right is actually the column at the upper y boundary of the computational box (if it exists). The search is modified to look for this column directly but the nquad linked-list loop etc., remains the same.

```
  }
  return FALSE;
}
```

If none of the tests are positive, FALSE is returned.

test_col

This small function is the bottom level in the density test function hierarchy and as such actually does the testing. It tests the two nodes in the supplied column. The function first looks to see is the second node in the column exists. If this node is not present no additional nodes beyond the first node should be created and the function returns FALSE.

```
boolean test_col(gridls *curgrid, cqptr curcquad, nqptr
                 curnquad, nptr curnode, long y, long x)
{
  nqptr fstnquad;             /* first nquad on a row */
  nptr icurnode = curnode;    /* current node */

  /* test if curnode++ is really there */
```

```
    if(x < (curnquad->x + curnquad->length - 1)){
      if((++icurnode)->dens > curgrid->critdens)     /* now test the density */
        return TRUE;
    }
    /* deal with periodicity */

    else if(x == (curgrid->l1dim - 1)){
      fstnquad = curcquad->loc + (y - curcquad->y);
      if(fstnquad->x == 0){
        icurnode = fstnquad->loc;            /* set node to 1st one on column */
        if(icurnode->dens > curgrid->critdens)
          return TRUE;
     }else
        return FALSE;
    }else
      return FALSE;

    /* test curnode */

    if(curnode->dens  > curgrid->critdens)
      return TRUE;

    return FALSE;
  }
```

A Note on Computational Overheads

Previously, in the initial discussion of the node density tests I noted that the addition of the buffer zones added a great many more tests, and hence a significant overhead. In practice this is not a bad as earlier implied. The reader will have noted from the code fragments above that if a density test is `TRUE` anywhere in the hierarchy the testing routine is exited immediately. In general less than the maximum number of tests will be performed per node. The ordering of the tests is such that the tests involving less searching are performed first. Hence the amount of CPU time involved in the density test is reduced to a minimum.

7.3.6 Moving the Particles from Coarse to Fine grid: `relink.c search.c part_search.c lltools.c unassign.c`

The process called "re-linking" is the procedure of taking coarse grid particles stored in linked lists in each coarse grid node and placing them within the linked lists of the next finer grid[3]. This process is complex and involves a substantial amount of code and operations to achieve efficiently as is hinted at by the number of source files in the title. Before I discuss the implementation it is worth returning to the problem of particle storage and explain why it is necessary to constantly re-link particles in the first place.

Storing the Particles

Where to store the particles so they may be quickly accessed is a problem for all N-body codes to overcome. MLAPM has taken an approach similar to that of P^3M codes although for different reasons. MLAPM stores the particles in linked-lists (although they are in fact allocated as a block). Each particle is placed, x-sorted, in a linked-list attached to the cell within which the particle is currently located. In regions with multiple grids a particle is placed in a linked-list within the finest cell in which it is physically located. If, however, that cell is on the boundary of that grid the particle is placed within the finest cell that is not a boundary cell.

[3]The reader will recall the storage arrangements for particles from the description of chapter 5.

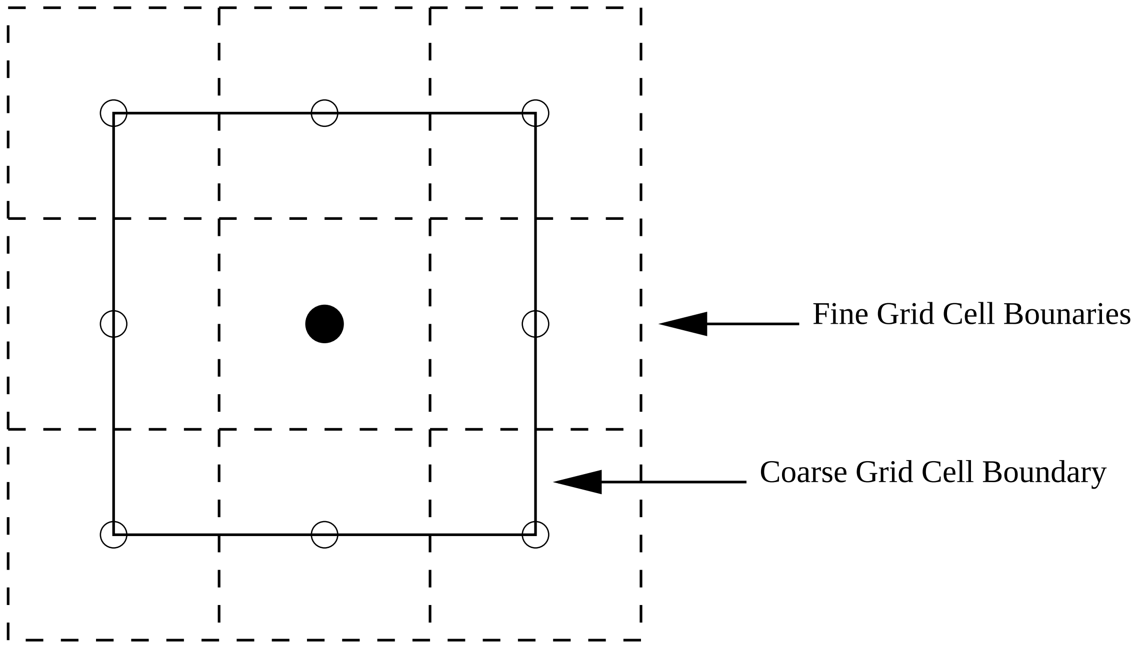


Figure 7.4: Particles located in the linked-list of the coarse grid cell will be dispersed among nine cells on the next finer grid

Why store the particles locally to the cells? Firstly to make mass assignment efficient; if all the particles located within a cell are easy to find, the additional burden of sorting particles is not added to the routine where the majority of CPU time is spent[4]. Secondly, to enable the use of multiple time-steps. To make the code work with a different time-step on each level of refinement, the particles that fall inside each level of refinement must be known and be readily accessible.

Why Linked Lists?

There are many ways of storing a list of data, why use singly-linked-lists? Firstly, altering linked lists does not involve any reallocation of memory [5] which is quite a slow process. If we had to continually search down the linked-list for the particles required, this would cause a problem. However, the particle lists are x-sorted and no sorting or search for particular particles takes place. De-referencing a pointer, I might add, is a very fast operation on modern computers.

The Overlap Problem

The major obstacle here is what I have termed the *Overlap Problem*. Recall figure 5.8 in chapter 5 and the accompanying discussion. In this figure I showed that because fine grid nodes and coarse grid nodes must be aligned. The definition of a "cell" as having the node at its centre leads to fine grid cells overlapping the boundaries of coarse grid cells. Figure 7.4 illustrates the two dimensional case.

As a result of the overlap of cell boundaries, when searching for particles to add to a new fine-grid cell, more than one coarse cell will have to be searched. In fact in three dimensions, up to eight coarse nodes may have to be searched. There are eight cases corresponding to the eight different ways a fine grid node may be aligned with the coarse grid. Figure 7.5 shows each of the eight cases.

[4]see the notes on the time profile in §8.3.8
[5]As say an array of pointers to particles would

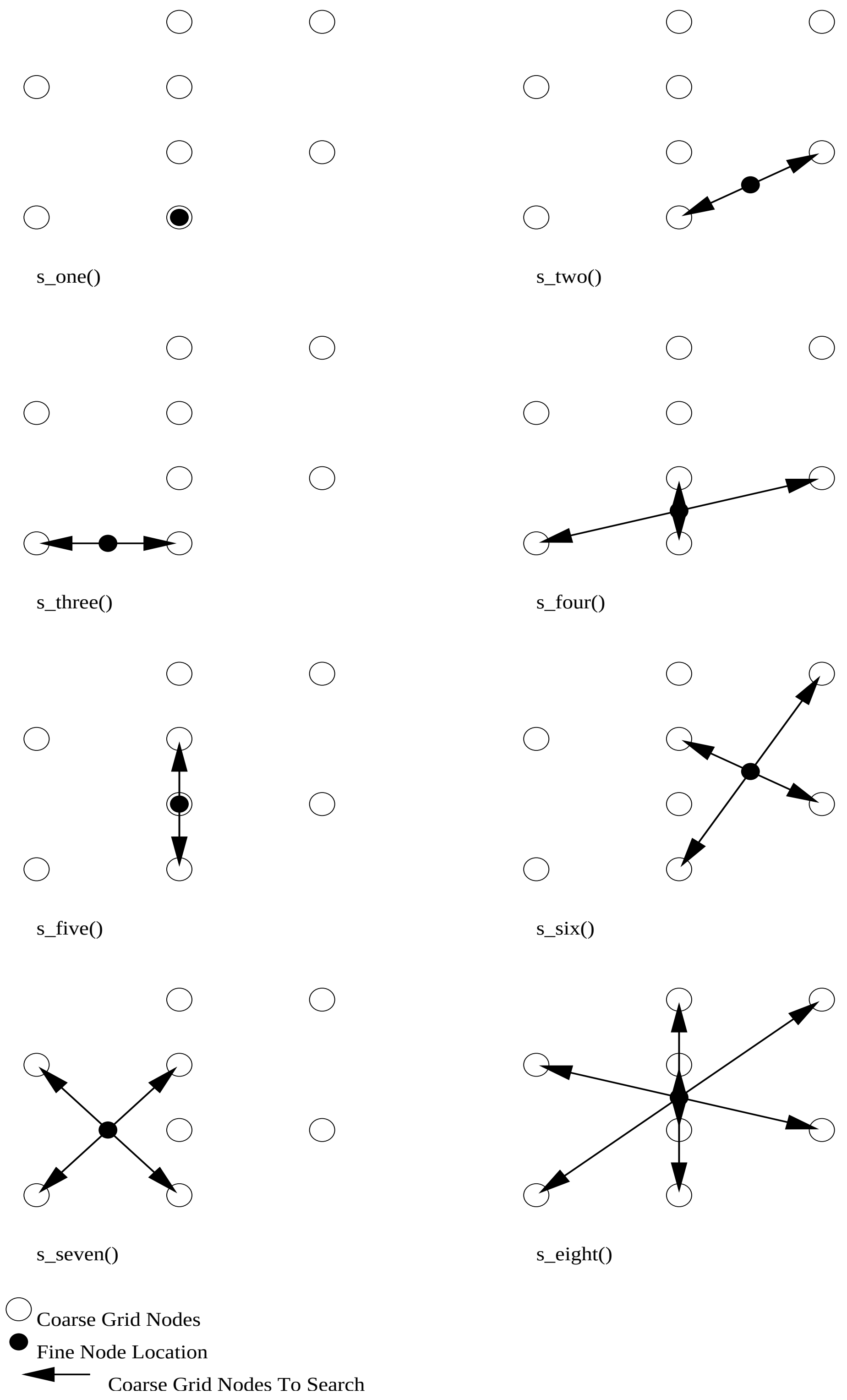


Figure 7.5: The eight particle search functions Corresponding to the eight different relative locations of a fine node to a coarse node are eight different combinations of coarse nodes that need to be searched for particles to re-link

Code Organization

The *relinking* code can be roughly divided into three parts, code responsible for organizing the looping process, code which operates on the linked-lists and particles, and support code. The bulk of the code in the four source files is organizational code found in the files `relink.c` and `part_search.c`. Operations on the linked-lists themselves may be found in the file `lltools.c`, while the particle unassign procedure (see later discussion for more details) is found in the file `unassign.c`. Support code can be found in the file `search.c`, which contains the code to find the twenty-six surrounding nodes[6] and also in the file `lltools.c`. In the following sections I will discuss the main functions in more detail.

Selecting the particle-search function: `relink.c`

The purpose of the relink functions `re_link()` through `re_link_node` is to determine which of the eight particle search functions should be called. A secondary role of these functions is to calculate the coordinates of the boundaries of the fine grid cell to which particles are being re-linked. In this section I will discuss the functions `re_link_pquad()` and `re_link_node()` in detail. Of the remaining functions in the source file `relink.c`, the functions `re_link_cquad()` and `re_link_nquad()` operate in a similar fashion to `re_link_pquad()`. The last remaining function is `re_link()` which performs a simple loop over pquads calling `re_link_pquad()`.

Recall the eight different relative locations of a fine and coarse grid node in figure 7.5. The relative location can be described by three flags, one per spatial dimension, indicating if the fine node lies the same plane as the coarse node. The re-link functions make use of this by successively splitting between two cases on each level. The function `ref_link_pquad()` differentiates between the case of the fine node being in the same z-plane as the coarse grid node and the case where the fine node is not on the same plane as the corresponding coarse grid node. Hence `re_link_pquad()` differentiates between two sets of search functions, `s_one()` through `s_four()` and `s_five()` through `s_eight()`. Similarly the functions `re_link_cquad()` and `re_link_nquad()` refine the "choice" of particle search function down to two of four and one of two respectively. `re_link_pquad` is now presented in full with annotations.

```
void re_link_pquad(gridls *coa_grid, gridls *fin_grid, pqptr
                   coa_pquad, pqptr fin_pquad)
{
  cqptr fin_cquad;     /* pointer to fine-grid cquad */
  cqptr t_fin_cquad;   /* temporary fine-grid cquad pointer */
  cqptr coa_cquad;     /* pointer to coarse-grid cquad */
  pqptr t_coa_pquad;   /* temporary coarse cquad pointer */
  long fin_z;                          /* fine-grid y index */
  long coa_z;                          /* coarse-grid y index */
  int nstate;          /* nquad state variable - determines whether */
  /* the cquad is cospatial with the coarse grid or not */

  npts = 0;                             /* assign no. of nodes to zero */
  massconv = p_mass*pow(coa_grid->l1dim, 3);     /* set the massconv factor */

  /* initialise nquad pointers */

  t_coa_pquad = coa_pquad;
  fin_cquad = fin_pquad->loc;
  fin_z = fin_pquad->z;
  for(coa_z = fin_z / 2; coa_z > t_coa_pquad->z +
      t_coa_pquad->length; t_coa_pquad = t_coa_pquad->next)
    ;     /* find correct coa_pquad */
  coa_cquad = t_coa_pquad->loc + (coa_z - t_coa_pquad->z);
  nstate = TRUE;
```

[6] The *search* code is used is several places within the code but is discussed here as this is the first place it is used

```
/* loop over fin_cquads - alternately looping over coa_cquad */
for(; fin_cquad < fin_pquad->loc + fin_pquad->length; fin_cquad++,
    fin_z++){
  for(t_fin_cquad = fin_cquad; t_fin_cquad != NULL; t_fin_cquad =
      t_fin_cquad->next){
    /* pass control to re_link_cquad */

    switch(nstate){
    case TRUE:
      re_link_cquad(coa_grid, coa_pquad, coa_cquad, fin_grid, fin_pquad,
                    t_fin_cquad, coa_z, fin_z,
                    (boolean (*)(gridls*, pqptr, cqptr, nqptr, nptr, nptr,
                                 long, long, long, float, float, float,
                                 float, float, float, float, float, float))
                                 (s_one),
                    (boolean (*)(gridls*, pqptr, cqptr, nqptr, nptr, nptr,
                                 long, long, long, float, float, float,
                                 float, float, float, float, float, float))
                                 (s_two),
                    (boolean (*)(gridls*, pqptr, cqptr, nqptr, nptr, nptr,
                                 long, long, long, float, float, float,
                                 float, float, float, float, float, float))
                                 (s_three),
                    (boolean (*)(gridls*, pqptr, cqptr, nqptr, nptr, nptr,
                                 long, long, long, float, float, float,
                                 float, float, float, float, float, float))
                                 (s_four));
      break;
    case FALSE:
      re_link_cquad(coa_grid, coa_pquad, coa_cquad, fin_grid, fin_pquad,
                    t_fin_cquad, coa_z, fin_z,
                    (boolean (*)(gridls*, pqptr, cqptr, nqptr, nptr, nptr,
                                 long, long, long, float, float, float,
                                 float, float, float, float, float, float))
                                 (s_five),
                    (boolean (*)(gridls*, pqptr, cqptr, nqptr, nptr, nptr,
                                 long, long, long, float, float, float,
                                 float, float, float, float, float, float))
                                 (s_six),
                    (boolean (*)(gridls*, pqptr, cqptr, nqptr, nptr, nptr,
                                 long, long, long, float, float, float,
                                 float, float, float, float, float, float))
                                 (s_seven),
                    (boolean (*)(gridls*, pqptr, cqptr, nqptr, nptr, nptr,
                                 long, long, long, float, float, float,
                                 float, float, float, float, float, float))
                                 (s_eight));
      break;
    default:
      perror("re_link_pquad: error in nstate switch");
      exit(1);
      break;
    }
  }
  if(is_even(fin_z) == FALSE){
    coa_cquad++;
    coa_z++;
    nstate = TRUE;
```

```
    }else
      nstate = FALSE;
```

The above loop over the fine pquads alternates between odd and even states. The even state corresponds to the "co-spatial" case. The appropriate set of four particle search functions is passed to `re_link_cquad()`, the next function down in the hierarchy. `re_link_cquad()` and subsequent functions work in a similar fashion.

```
  }
  npts = 0;
}
```

The function `re_link_node()`, presented below, is a wrapper for the particle search function. All the particle search functions need to be able to access all the fine grid nodes surrounding the fine node upon which we are operating. Hence the function `search3l_pl9()` is called to obtain pointers to all the nodes surrounding the node `fin_node`[7]. The upper and lower x-boundaries of the fine grid cell are also calculated here. A fragment from `re_link_node()` is illustrated below, I have left out the long argument list and initial declarations.

```
  tsc_nodes[1][1][1] = fin_node;   /* current node in centre */

  /* get pointers to tsc nodes */
  npts++;
  search3l_pl9(fin_grid, fin_pquad, fin_cquad, fin_nquad, tsc_nodes,
               &fin_z, &fin_y, &fin_x);

  if(test_tsc(tsc_nodes) == TRUE){
```

If any of the returned node pointers are `NULL` this indicates that the fin-grid node under consideration, `fin_node`, is on the edge of the fine grid. As explained elsewhere, forces cannot be obtained for sub-grid edge cells and particles must not be placed in them. The function `test_tsc()` returns `FALSE` if any of the pointers in the three dimensional array `tsc_nodes` is `NULL`, and hence the particle search function is not called in this case.

```
    /* calculate ranges */
    temp_x = (((float)fin_x - 0.5) / (float)fin_grid->l1dim) + (float)SHIFT;
    min_x = modulo_one(temp_x);
    temp_x = ((((float)fin_x + 0.5) / (float)fin_grid->l1dim) + (float)SHIFT);
    max_x = modulo_one(temp_x);
    temp_x = (((float)fin_x / (float)fin_grid->l1dim) + (float)SHIFT);
    mid_x = modulo_one(temp_x);

    /* call particle search function */
    search(coa_grid, coa_pquad, coa_cquad, coa_nquad, coa_node, fin_node,
           coa_x, coa_y, coa_z, mid_x, min_x, max_x, min_y, max_y, mid_y, min_z,
           max_z, mid_z);
  }
}
```

Controlling Operations on the Particle Linked-lists: `part_search.c`

The particle search functions control the removal of particles from coarse-grid linked-lists and placing them in fine-grid linked lists. As explained above there are eight of these functions, of which I propose to present one in detail. As a result of the *overlap* problem these functions vary in complexity from `s_one` which need only access particles in one coarse-grid linked-list to `s_eight`

[7] The references to "tsc" in the function are referring to *triangular-shaped-cloud* the mass assignment scheme used by MLAPM. The function `search3l_pl9` was initially designed to obtain pointers to all the nodes surrounding the central node for the purpose of mass assignment

which accesses eight coarse-grid linked-lists. Essentially these functions loop through the coarse-grid linked lists testing if each particle should be placed in the fine-grid cell that is being considered. As the particles are sorted in the x-coordinate direction, the tests are simplified. The operations on the linked-list are carried out by the two functions `inc_to_a_x()` and `tran_to_a_x()` both in the `lltools.c` source file. The first of the functions, `inc_to_a_x()`, increments down the linked list upto the supplied x coordinate. The function `tran_to_a_x()` transfers particles down to the supplied x-coordinate. These two functions are described in greater detail below.

I will now present the function `s_one()` with annotations. This is the simplest of the functions but does illustrates how all eight functions work.

```
boolean s_one(gridls *coa_grid, pqptr coa_pquad, cqptr coa_cquad, nqptr
              coa_nquad, nptr coa_node, nptr fin_node, long coa_x, long coa_y,
              long coa_z, float mid_x, float min_x, float max_x, float min_y,
              float max_y, float mid_y, float min_z, float max_z, float mid_z)
{
  partptr *cur_part;      /* pointer to pointer to current particle */
  partptr tran_ls;        /* pointer to particle transfer list */
  partptr fin_end;        /* end of linked list */

  /* initialize cur_part */
  cur_part = &(coa_node->ll);

  /* malloc tranls particle ie a temporary head of chain */
  tran_ls = c_part(1);
```

A temporary particle is created to act as head of chain during the transfer process

```
  /* test to see if we are in middle of grid */
  if(max_x > min_x){                             /* normal case */
```

This tests if the fine cell straddles the periodic boundary. If it does the logic for the x-tests must be altered

```
    /* increment down ll to min_x */
    if((cur_part = inc_to_a_x(cur_part, min_x)) == NULL)
      return FALSE;
```

The fine-grid cell is in the centre of this coarse-grid cell. Hence no particles should be tested with x-coordinates between that of the coarse-grid cell boundary and the fine-grid cell boundary at `min_x`. The pointer `cur_part` is incremented down the linked-list until particles in the list have x-coordinates greater than `min_x`. If no particles were in the list `FALSE` is returned.

```
    /* transfer particles to tran_ls */
    cur_part = tran_to_a_x(coa_grid, coa_pquad, coa_cquad, coa_nquad, coa_node,
                           coa_x, coa_y, coa_z, cur_part, tran_ls, &fin_end,
                           max_x, min_y, max_y, mid_y, min_z, max_z, mid_z,
                           (boolean (*)(float, float, float, float, float, float,
                                        float, float,
                                        boolean (*z_test)(float, float, float,
                                                          float)))(y_test_c),
                           (boolean (*)(float, float, float, float))(z_test_c));
    fin_node->ll = tran_ls->ll;
    return TRUE;
```

Here the particles are transfered from the temporary head of chain to the node particle pointer `ll`. Particles were in the list so true is returned.

```
  }else{     /* edge case */
```

Now the edge case is considered. If the cell lies over the periodic boundary the upper boundary of the cell will have a lower x coordinate than the lower boundary of the cell. The particle linked-lists will also reflect this as they are ordered by increasing x coordinate. Hence in this situation the roles of min_x and max_x are reversed.

```
    /* transfer particles to tran_ls until we reach max_x */
    if((cur_part = tran_to_a_x(coa_grid, coa_pquad, coa_cquad, coa_nquad,
                              coa_node, coa_x, coa_y, coa_z, cur_part, tran_ls,
                              &fin_end, max_x, min_y, max_y, mid_y, min_z,
                              max_z, mid_z,
                              (boolean (*)(float, float, float, float, float,
                                           float, float, float,
                                           boolean (*z_test)
                                             (float, float, float, float)))
                                             (y_test_c),
                              (boolean (*)(float, float, float, float))
                              (z_test_c))) == NULL){
      fin_node->ll = tran_ls->ll;
      return  TRUE;
```

Particles are transferred between linked-lists upto x-coordinate max_x. If there are no more particles beyond this point NULL will be returned by tran_to_a_x() and the function is exited.

```
    }
    /* increment down ll to min_x */
    if((cur_part = inc_to_a_x(cur_part, min_x)) == NULL){
      fin_node->ll = tran_ls->ll;
      return TRUE;
    }
    /* transfer particles until end of ll */
    cur_part = tran_to_a_x(coa_grid, coa_pquad, coa_cquad, coa_nquad, coa_node,
                           coa_x, coa_y, coa_z, cur_part, fin_end, &fin_end,
                           END, min_y, max_y, mid_y, min_z, max_z, mid_z,
                           (boolean (*)(float, float, float, float, float, float,
                                        float, float,
                                        boolean (*z_test)(float, float, float,
                                                          float)))(y_test_c),
                           (boolean (*)(float, float, float, float))(z_test_c));
    fin_node->ll = tran_ls->ll;
    tran_ls->ll = NULL;
    freen((void *)&tran_ls);
    return TRUE;
```

Here the pointer is first increment through the middle of the list up to the point min_x. Beyond this point all particles should be tested for transfer and hence tran_to_a_x() is called up to the coord END[8].

```
  }
}
```

The reader will have noted that two functions, y_test_c and z_test_c, are passed to the function tran_to_a_x(). These are the y and z test functions which test if each particle is within the new fine-grid cell boundaries. The z test function is in fact also passed as an argument of the y test function which is why the function argument list is so complex. A detailed explanation of these functions is given later.

The remaining seven particle search functions follow the same pattern as s_one(), with the addition of a loop over more coarse-grid linked-lists. I refer the reader to the source code file part_search.c. There are two main complications in that there are more coarse-grid nodes to find

[8] END is #defined to be 2.0

and in the need to use different y and z test functions for each of the coarse-grid cell linked-lists. The former is achieved in a manner similar to that used by the function `search3l_pl9()` which is described below, while the later is achieved through the use an array of pointers to functions.

Operating on Particle Linked-Lists: `lltools.c`

The two functions which operate on the linked lists are, as noted above, `inc_to_a_x()` and `tran_to_a_x()`. Of these the former is the simpler. `inc_to_a_x()` simply increments down the particle list until it finds a particle with an x coordinate greater than or equal to `a_x` at which point it returns a pointer to it. If the linked-list ends without finding such a particle, `NULL` is returned. The function is illustrated below.

```
partptr *inc_to_a_x(partptr *cur_part, float a_x)
{
  /* empty particle list return NULL */
  if((*cur_part) == NULL)
    return NULL;

  /* loop over ll until *cur_part->pos[X] => a_x */

  for(; (*cur_part)->pos[X] < a_x; cur_part = &((*cur_part)->ll)){
    if((*cur_part)->ll == NULL)
      return NULL;
  }
  return cur_part;
}
```

The transfer function `tran_to_a_x()` also increments down the particle linked-list until a particle with an x coordinate greater than or equal to `a_x` is found. In addition all particles are tested by calling the supplied `y_test()` function (which in turn calls the `z_test()` function passed to the `y_test()` function). If a particle is transferred, its mass must be "unassigned" from the coarse grid. This is achieved through the function `unassign()`.

The requirement for particles moving from a coarse grid to a finer one to have their mass unassigned comes from the use of multiple timesteps. Mass assignment will only be carried out on a coarse grid at the start of a coarse-grid timestep. At intervening sub-grid timesteps, the density field must reflect the current fine grid density field restricted to the coarse grid. Hence when a particle is "re-linked", its mass is removed from the coarse grid. The particle mass is then assigned on the fine grid and then the whole sub-grid mass distribution is restricted back to the coarse grid. After a fine grid timestep the density field will again be restricted back to the coarse grid.

```
partptr *tran_to_a_x(gridls *cur_grid, pqptr cur_pquad, cqptr
                     cur_cquad, nqptr cur_nquad, nptr cur_node,
                     long x, long y, long z, partptr *cur_part,
                     partptr tran_ls, partptr *fin_end,  float a_x,
                     float min_y, float max_y, float mid_y, float min_z,
                     float max_z, float mid_z,
                     boolean (*y_test)(float, float, float, float, float,
                                       float, float, float,
                                       boolean (*z_test)(float, float, float,
                                                         float)),
                     boolean (*z_test)(float, float, float,
                                       float))
{
  partptr fin_part;    /* pointer to current last particle in tran_ls */
  fin_part = tran_ls;

  /* empty particle list hence return NULL */
  if((*cur_part) == NULL)
    return NULL;
```

```
  /* while the current particle is within x range */

  while((*cur_part)->pos[X] < a_x){
    if((*cur_part)->ll == NULL){    /* if this is the last particle */
      if(((*y_test)(min_y, max_y, mid_y, ((*cur_part)->pos[Y]), min_z,
                    max_z, mid_z, ((*cur_part)->pos[Z]),
                    (boolean (*)(float, float, float, float))
                      (z_test))) == TRUE){
        unassign(cur_grid, cur_pqaud, cur_cquad, cur_nquad, cur_node,
                 z, y, x, *cur_part);
        fin_part->ll = *cur_part;   /* connect cur_part to tranls */
        fin_part = fin_part->ll;    /* move down tranls */
        *cur_part = (*cur_part)->ll;/* move down cur_part ll */
        fin_part->ll = NULL;        /* new tranls particle is not */
        /* followed */
      }
      *fin_end = fin_part;          /* set end-of-list pointer */
      return NULL;
```

If the particle fin_part is the last, finish off by setting the end of list pointer (fin_end and setting fin_part's ll pointer to be NULL.

```
    }
    /* not the last but needs to be transferred */
    else if(((*y_test)(min_y, max_y, mid_y, ((*cur_part)->pos[Y]),
                       min_z, max_z, mid_z, ((*cur_part)->pos[Z]),
                       (boolean (*)(float, float, float, float))(z_test))) == TRUE){
      unassign(cur_grid, cur_pquad, cur_cquad, cur_nquad, cur_node, z, y, x,
               *cur_part);
      fin_part->ll = *cur_part;     /* connect cur_part to tranls */
      fin_part = fin_part->ll;      /* move down tranls */
      *cur_part = (*cur_part)->ll;  /* move down cur_part ll */
      fin_part->ll = NULL;          /* new tranls particle is not followed */
```

This copes with the standard case of a particle in the middle of the linked-list. If the particle passes the y_test() it is added to the transfer list.

```
    }/* not to be transferred, not the last */
    else
      cur_part = &((*cur_part)->ll);/* move down cur_part ll */
  }
  *fin_end = fin_part;              /* set end-of-list pointer */
  return cur_part;
}
```

Operations on particles: unassign.c

The function unassign() is almost identical (barring a sign) to the mass assignment routine assign(). Hence I refer the reader to §7.3.9 for details.

Support Code: y_test_?() & z_test_?

There are six of these test functions, y_test_c, y_test_l, y_test_u, z_test_c, z_test_l and z_test_u. The labels c, l and u refer to "centre", "lower" and "upper" respectively. The *overlap problem* is again the complicating issue and it is this that is responsible for the need for six test functions. Consider the y case (the z case is identical), if the fine grid node lies in the same y-plane that the base coarse grid node lies in, then only the "centre" test, y_test_c is required. If, however, the opposite is true, then the "lower" test will be required for particles attached to the base coarse

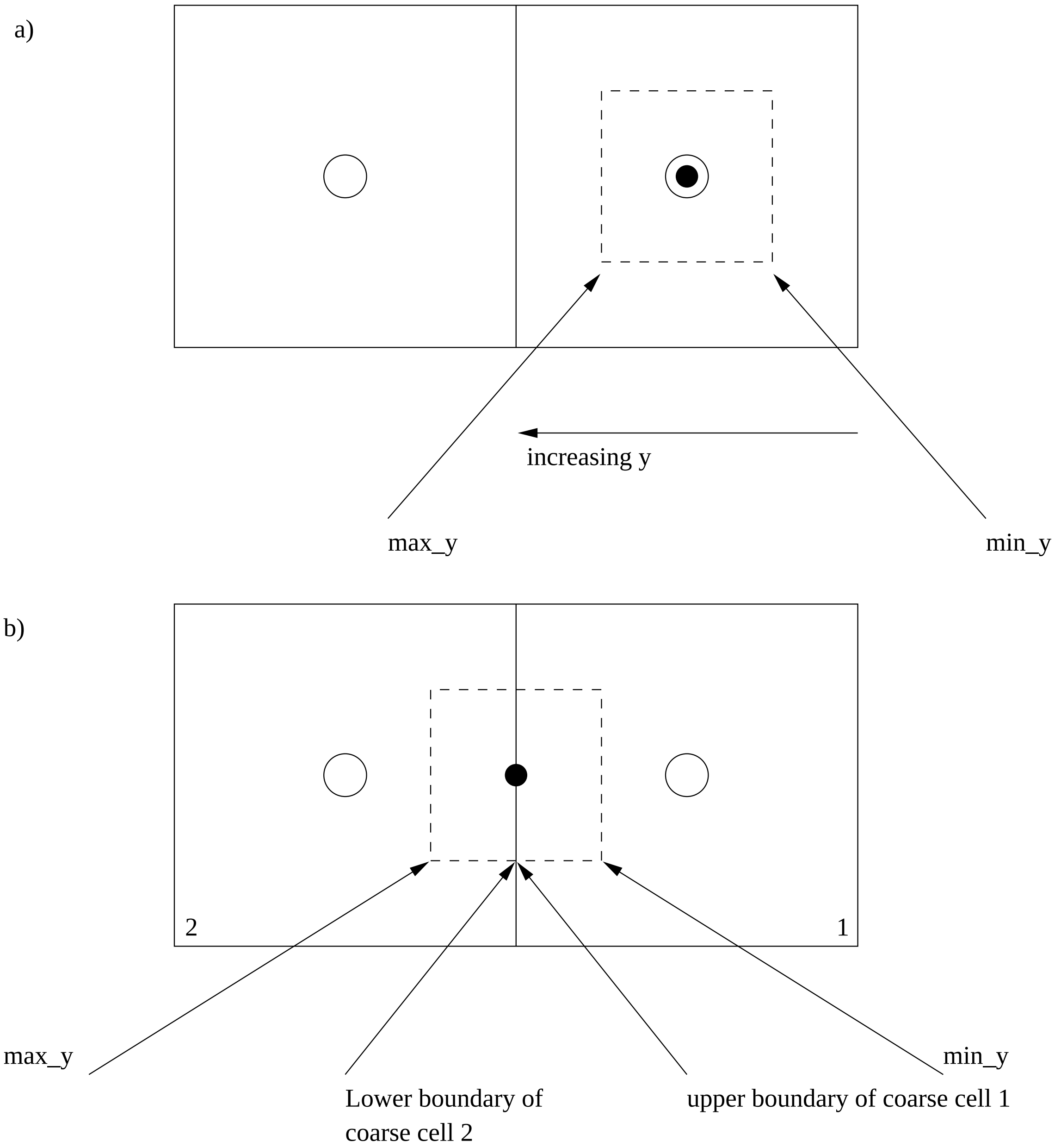


Figure 7.6: y_tests Here we see the two cases quite clearly (in 2D). In case (a) we need to test the particles in the linked-list in cell 1 to see if they have y-coordinates between `min_y` and `max_y`. In case (b) we need to test the particles in cell 1 to see if they have y coordinate between `min_y` and the upper boundary of the cell (`y_test_l`) and the particles of cell 2 to see if they have y coordinate between the lower y boundary of the cell and `max_y`.

grid node and the "upper" test will be required for particles attached to the coarse grid node one above the base coarse grid node in the y direction. This is illustrated in figure 7.6.

I will now show the test functions `y_test_c()` and `z_test_c()` indicating where the "l" and "u" variants differ.

```
boolean y_test_c(float min_y, float max_y, float mid_y, float test_y,
                 float min_z, float max_z, float mid_z, float test_z,
                 boolean (*z_test)(float, float, float, float))
{
  float mid_upper;

  if(mid_y == 0.0)
    mid_upper = 1.0;
        else
          mid_upper = mid_y;
```

This section deals with the periodic case. If `mid_y` lies on the periodic boundary then the test must be split into `min_y` to 1.0 and 0.0 to `max_y`.

```
  if(((test_y >= min_y) && (test_y < mid_upper)) ||
     ((test_y >= mid_y) && (test_y < max_y))){
```

In the case of `y_test_l` only the first line of this `if` test is performed, while in the case of `y_test_u` only the second line of the test is performed.

```
    if((z_test(min_z, max_z, mid_z, test_z)) == TRUE)
      return TRUE;
    else
      return FALSE;
  }else
    return FALSE;
}

boolean z_test_c(float min_z, float max_z, float mid_z, float test_z)
{
  float mid_upper;

  if(mid_z == 0.0)
    mid_upper = 1.0;
  else
    mid_upper = mid_z;

  if(((test_z >= min_z) && (test_z < mid_upper)) ||
     ((test_z >= mid_z) && (test_z < max_z)))
```

In the case of `z_test_l` only the first line of this `if` test is performed, while in the case of `z_test_u`, only the second line of the test is performed.

```
    return YES;
  else
    return NO;
}
```

Support Code: `merge_ll()`

The function `merge_ll()` is in the source file `lltools.c` and is used by most of the particle search functions to merge two x-sorted linked-lists into one x-sorted linked-list. I refer the interested reader to the source file for further details.

7.3.7 Finding the 26 neighbouring Nodes: search.c

In several places within the code it is necessary to find the 26 neighbouring nodes. The function search3l_pl9() searches for these nodes and fills a three dimensional array of pointers to nodes with either a pointer to the correct node or NULL if the node is not present. The search function algorithm is hierarchical in nature. The top level function search3l_pl9() calls a "plane" search function searchpl9() on the z-plane cospatial with the original node and on the planes above and below (accounting for the periodicity). In turn the function searchpl9() calls a "column" search function searchcol() three times. searchcol() finds the three nodes required from a particular column. The main function search3l_pl9() is illustrated below. The remaining functions operate in a similar fashion and I refer the reader to the source code.

```
void search3l_pl9(gridls *curgrid, pqptr curpquad, cqptr curcquad,
                  nqptr curnquad, nptr tsc_box[3][3][3], long *z, long *y,
                  long *x)
```

The array tsc_box[3][3][3] is used to return the pointers to nodes.

```
{
  pqptr icurpquad;         /* current coarse pquad */
  cqptr icurcquad;         /* current coarse cquad */
  nqptr icurnquad;         /* current coarse nquad */
  nptr icurnode;           /* current coarse node */
  long endx;                                     /* end coord of icurnquad */
  long endy;                                     /* end coord of icurcquad */
  long endz;               /* end coord of icurcquad */

  /* search plane with curnode */
  searchpl9(curgrid, curpquad, curcquad, curnquad, tsc_box[1], *z, *y, *x);
```

The first call to searchpl9() searches the plane that contains the original node cur_node. Using the way arrays are organized in C means only part of the array tsc_box need be passed to the function.

```
  /* search plane below curnode */
  /* is plane there ? */
  if(*z > curpquad->z){
```

The periodic case is dealt with separately.

```
    /* now look for correct cquad */
    /* increment down cquad ll */
    for(icurcquad = (curpquad->loc + (*z - 1 - curpquad->z));
        (*y > (endy = (icurcquad->y + icurcquad->length)))
          && (icurcquad->next != NULL); icurcquad = icurcquad->next)
      ;
```

The familiar pattern of searching through the cquads to find one containing an nquad at the correct y coordinate is repeated here.

```
    if((*y >=  icurcquad->y) && (*y < endy)){ /* is nquad in this cquad */
      /* now look for correct nquad */
      /* increment down nquad ll */
      for(icurnquad = icurcquad->loc + (*y - icurcquad->y);
          (*x > (endx = (icurnquad->x + icurnquad->length)))
            && (icurnquad->next != NULL); icurnquad = icurnquad->next)
        ;
      if((*x >=  icurnquad->x) && (*x < endx)){/* is node in this nquad */
        icurnode = icurnquad->loc + (*x - icurnquad->x);
        tsc_box[0][1][1] = icurnode;      /* assign middle node */
        searchpl9(curgrid, curpquad, icurcquad, icurnquad, tsc_box[0],
                  modulo(*z - 1, curgrid->l1dim), *y, *x);
```

Once the node in the centre of 3x3 grid immediately below the original node is found, `searchpl9()` is called with the correct cquad and nquad as arguments. This time the lower part of the three-dimensional array `tsc_box` is passed to the function.

```
    }else{
      null_plane(tsc_box[0]);
```

If the central node on the 3x3 plane below `cur_node` is not present, the corresponding plane of pointers in `tsc_box` is set to `NULL` by the function `nul_plane()` (The function `nul_col()` performs a similar role for the function `searchpl9()`).

```
    }
  }
  /* periodic case */
  else if(*z == 0){

    /* search for correct nquad */
    for(icurpquad = curgrid->gridptr;
        (( curgrid->l1dim - 1) > (endz = (icurpquad->z + icurpquad->length)))
          && (icurpquad->next != NULL); icurpquad = icurpquad->next)
      ;
    if(((curgrid->l1dim - 1) >= icurpquad->z) &&
       ((curgrid->l1dim - 1) < endz)){   /* is cquad in this pquad */
      ⋮
```

In the periodic case the search for the correct cquad is modified. The cquad is no longer guaranteed to be physically next to `cur_cquad` in memory. The pquads must be search to find the last one covering the box. One the last pquad has been found the last cquad in it has it's z coordinate tested to see if it is on the boundary. Once a cquad has been found the remainder of the section is identical to that above.

```
  /* search plane above */
  ⋮
```

The search for the plane above the original node follows a similar pattern.

```
}
```

7.3.8 Storing and Restoring the Density: store_dens.c

In the multiple timestep mode of operation the density must be stored. This reflects the need to restore the density to all grids, once a timestep has been taken. For a full explanation I refer the reader to the discussion of the timestep cycle `step()` in §7.6.1. I will briefly run through the functions `store_dens()` and `restore_dens()` at this point in the test as this is where they first appear.

I will not show any annotated code here as both the functions are very simple. They both use a hierarchy of functions (`stdens_pquad()` through `stdens_nquad()` to loop over a grid. Depending on whether the density is being stored or re-stored the function `stdens_node()` or `rstdens_node()` is passed through the grid-loop hierarchy. The density is stored in a member of the node structure `olddens`[9] (see chapter 5).

7.3.9 Mass Assignment: `assign.c` & `zero_mean_dens.c`

Mass assignment in MLAPM is performed using the *triangular-shaped cloud* scheme (see chapter 2). The mass assignment is done by the functions of `assign.c`. A second step is also required, recall

[9] Only present if `MULTISTEP` is defined.

equation (3.13), the mean density of the universe must be subtracted from the actual density. This is natural given that MLAPM uses periodic boundary conditions. No solution of Poisson's equation exist if periodic boundary conditions are used and the average density in the computational box is non-zero. This second step is performed by the functions in the source file `zero_mean_dens.c`.

`assign.c`

`assign.c` contains the mass assignment control routines `assign()`, `ass_pquad()`, `ass_cquad()` and `ass_nquad()`. Mass assignment is performed by the function `ass_node`. The control routines loop over all the nodes on a given grid in the now familiar fashion and hence will not be presented here. The function `ass_node()` is presented below with annotations.

```
static float masstodens;    /* global - mass to density conversion factor */
```

This global variable (declared `static` so it's only visible within this source file) is the mass to density conversion factor. It is equal to the number of nodes per side on the grid, cubed. The value is calculated in the function `assign()`.

```
void ass_node(gridls *grid_ptr, pqptr cur_pqud, cqptr cur_cquad,
              nqptr cur_nquad, nptr cur_node, long z_coord, long y_coord,
              long x_coord)
{
  nptr tsc_nodes[3][3][3];  /* nodes to assign to */
  partptr cur_part;         /* current particle */
  fvect xyz_coords;         /* actual float coords of node */
  fvect pn_sep;             /* particle-node separation / grid_spac */
  int i_coord;              /* coord changing index */
  int i,j,k;                /* indices for 3D arrays */
  fvect weights[3];         /* weights in each dimension */
  float pnarg_a;            /* pn sep temp arg */
  float pnarg_b;            /* pn sep temp arg */
  float tpnarg;             /* temp to calc pn args */
  fvect temp_coords;        /* float vector - calc un_mod coords */

  tsc_nodes[1][1][1] = cur_node;   /* current node in centre */
```

Triangular-shaped cloud mass assignment scheme assigns mass to the twenty-seven nodes surrounding the particle's position. Pointers to these nodes are obtained using the `search3l_pl9()` function described in §7.3.7. The central node of the 3x3x3 block is initialized to point to the "parent" node of the particle whose mass is being assigned.

```
  /* calculate realspace coords of cur_node */
  temp_coords[X] = (((float) x_coord) /
                    (float)grid_ptr->l1dim) + (float)SHIFT;
  xyz_coords[X] = modulo_one(temp_coords[X]);
  temp_coords[Y] = (((float) y_coord) /
                    (float)grid_ptr->l1dim) + (float)SHIFT;
  xyz_coords[Y] = modulo_one(temp_coords[Y]);
  temp_coords[Z] = (((float) z_coord) /
                    (float)grid_ptr->l1dim) + (float)SHIFT;
  xyz_coords[Z] = modulo_one(temp_coords[Z]);
```

The first step in calculating the weights (see equation (3.34)) is to calculate the actual physical coordinates of the central node accounting for periodicity. This is done in the above section and stored in the vector of doubles `xyz_coords[]`.

```
  /* get pointers to tsc nodes */

  search3l_pl9(grid_ptr, cur_pquad, cur_cquad, cur_nquad, tsc_nodes,
               &z_coord, &y_coord, &x_coord);
```

```
  /* loop over ll */

  for(cur_part = cur_node->ll; cur_part != NULL; cur_part =
        cur_part->ll){

    /* calc fraction to be assigned to each tsc node */

    for(i_coord = 0; i_coord < NDIM; i_coord++){
      pn_sep[i_coord] = (cur_part->pos[i_coord] - xyz_coords[i_coord]) *
        grid_ptr->l1dim;
```

Once the coords of the node have been obtained, the next step is to calculate the separation of the particle from the parent node.

```
      if(fabs(pn_sep[i_coord]) > 0.5){
        tpnarg = cur_part->pos[i_coord] + 0.5;
        pnarg_a = modulo_one(tpnarg);
        tpnarg = xyz_coords[i_coord] + 0.5;
        pnarg_b = modulo_one(tpnarg);
        pn_sep[i_coord] = (pnarg_a - pnarg_b) * grid_ptr->l1dim;
```

Periodicity can produce components of the separation vector pn_sep[] greater than 0.5 when the parent cell lies across the periodic boundary. To correct this particle and node have their coordinates shifted by 0.5 and then fmod applied again to give new coordinates. The true separation is then obtained.

```
      }
    }
    for(i_coord = 0; i_coord < NDIM; i_coord++){
      weights[1][i_coord] = 0.75 - pow(pn_sep[i_coord], 2);
      weights[0][i_coord] = 0.5 * pow((0.5 - pn_sep[i_coord]), 2);
      weights[2][i_coord] = 0.5 * pow((0.5 + pn_sep[i_coord]), 2);
    }

    /* assign mass */

    for(k = 0; k < 3; k++){
      for(j = 0; j < 3; j++){
        for(i = 0; i < 3; i++){
          if(tsc_nodes[k][j][i] != NULL){
            tsc_nodes[k][j][i]->dens += weights[k][Z] * weights[j][Y] *
              weights[i][X] * masstodens;
```

Finally the weights are combined to give the total fraction of each particle mass which must be assigned to the surrounding nodes. The function unassign() differs only in the sign of the assignment.

```
          }
        }
      }
    }
  }
}
```

zero_mean_dens.c

This source file contains the functions zmd_pquad() through zmd_nquad(). These functions loop through all the nodes on the grid subtracting the mean density from the dens member of each node structure. The value of mean_dens is set in the startup function startrun(). The external variable mean_dens is defined in the source file mlapm_main.c.

7.4 The Multi-grid Potential Solver

The multi-grid algorithm was discussed in detail in chapter 4. In this section I will discuss the implementation of the multi-grid algorithm as the potential solver at the heart of MLAPM. The code for the solver is contained in the source files `mg_control.c`, `solve_cg.c`, `gs.c`, `go_up.c`, `go_down.c`, `resid_norm.c`, `trunc_err.c`. The multigrid solver is controlled by the function `mg_control()` found in the source file `mg_control.c`. The exact solution required for the coarsest grid is performed by the function `solve_cg()` in the file `solve_cg.c`. `gs.c` contains functions to perform a Gauss-Seidel sweep on a grid. The file `go_down.c` contains the prolongation operator, while `trunc_err.c` also deals with restriction. The file `resid_norm.c` contains the routine to calculate the norm of the residual. Finally the truncation error calculation is performed by the routines in the source file `trunc_err.c`

7.4.1 Multigrid control: `mg_control.c`

The multigrid solver is controlled by the function `mg_control()` which controls the implementation of Brandt's cycle C from the start of iterations until convergence has been achieved on the finest grid. Convergence is tested by the function `convergence()` while slow convergence is tested for by the function `slow_conv()`. The function `find_domgrid_no()` can also be found in the source file `mg_control.c`, this function is used to find the index number of the finest grid which covers the whole computational box.

`mg_control()`

The multigrid cycle implemented in MLAPM is Brandt cycle C. Although this is an early approach to multigrid, it has shown itself to be effective in this context. There are some difference between the cycle as implemented here and Brandt's cycle C [10].

The cycle begins on the *domain*[11] grid rather than the finest grid. Early tests showed that reliable convergence could be best achieved by starting on the domain grid. To understand why this should be so, consider the boundary conditions. On the domain grid the boundary conditions are triply-periodic while on sub-grids the boundary conditions are fixed. The sub-grid boundary conditions are fixed by the potential on the boundary, found by interpolating from the next coarser grid. Hence it makes sense not to iterate over sub-grids until a solution has been obtained on the domain grid, although of course that domain grid solution make be further refined in the same multigrid cycle.

The second key difference is that once the multigrid cycle has progressed on to sub-grids, it is prevented from returning to the set of domain grids. This is done purely because of the periodic boundary conditions. In testing convergence occasionally slowed dramatically. This slow convergence arose because the zero total mass condition on domain grids was violated when residuals from sub-grids were transfered back onto the domain grids. This corresponds to the system of linear equations we are solving becoming singular. Suisalu and Saar (1995a) solve this problem by using the Galerkin condition to modify the difference approximation on coarse grids. The ART (see for example (Knebe et al., 1999)) code simply uses iterative refinement, which would be achieved in MLAPM by only allowing the cycle to progress downwards. The practical solution suggested here works well, and MLAPM could be viewed as using *two* multi-grid solutions, one for the domain grid problem and one for the adaptive sub-grid problem.

```
/* mg_control: control routine for the multigrid Poisson Solver */
void mg_control(gridls *grid_list, int no_grids)
{
  int grid_no;             /* the current grid number */
  int no_grids_minus_one;  /* no_grids minus one */
  int i;                   /* index for GS sweep loop */
```

[10] see chapter 4 and (Brandt, 1977a)

[11] Finest grid covering the whole computational domain.

```
int domgrid_no;          /* grid no. of domain grid */
gridls *cur_grid, *for1_grid;        /* pointer to the current grid */
boolean cg_conv;         /* has current grid converged? */
boolean nondomain;       /* passed through onto non-domain grids */

/* zero resids */
for(cur_grid = grid_list; cur_grid < grid_list + no_grids; cur_grid++)
  cur_grid->cur_resid = 0.;

/* set the cur_grid to the domain grid & cg_conv to FALSE */
no_grids_minus_one = no_grids - 1;
domgrid_no = find_domgrid_no();
grid_no = domgrid_no;
cur_grid = dom_grid;
cg_conv = FALSE;
nondomain = FALSE;

/* loop over the grid until converged on finest grid */

while(!((cg_conv == TRUE) && (grid_no == no_grids_minus_one))){

  if(grid_no == 0){
    solve_cg(cur_grid);
    cur_grid = go_down(cur_grid);
    grid_no++;
  }else{
    cur_grid->old_resid = cur_grid->cur_resid;
    for(i = 0; i < NSWEEP; i++)
      gs(cur_grid); /* do GS relaxation sweep */

    cur_grid->cur_resid = resid_norm(cur_grid);   /* calc the residual */
```

On any grid other than the coarsest NSWEEP iterations of Gauss-Seidel relaxation are performed.

```
    if((cg_conv = converged(cur_grid, grid_no, domgrid_no)) == TRUE)
      {
        if(grid_no != no_grids_minus_one){   /* Are we on the finest grid ?*/
          cur_grid = go_down(cur_grid);
          grid_no++;
```

Test for convergence and if true and not on the finest grid continue iterations on the next finer grid.

```
          if(grid_no > domgrid_no)
            nondomain = TRUE;
          cg_conv = FALSE;
        }
      }
    else if(slow_conv(cur_grid) == TRUE){
```

Test for slow convergence and if true continue iterating on the next coarser grid. If, however, we have passed off the domain grid onto sub-grids, prevent the cycle from crossing back onto the domain grids.

```
      if(nondomain == FALSE){
        if((grid_no != 0)){   /* Are we on the coarsest grid ?*/
          cur_grid = go_up(cur_grid);
          grid_no--;
        }
      }
```

```
          else{
            /* don't go back up to domain grids */
            if(grid_no >  (domgrid_no + 1)){
              cur_grid = go_up(cur_grid);
              grid_no--;
            }
            /* stay on this grid */
          }
        }
      }
    }
  }
```

`converged()`

The function `converged()` determines if the solution on a particular grid has converged. The residuals and the truncation error are calculated and compared to determine if convergence to within the limits of accuracy (i.e. to within the truncation error) has been achieved.

`slow_conv()`

This function tests for slow convergence. It does so by comparing the ratio of the latest set of residuals to the previous set with the parameter `ETA`. `ETA` has the default value of 0.625 defined at the top of `mg_control.c`.

`find_domain_grid()`

This function finds the index number of the domain grid. This is given by

$$n_i = \log_2(gridsize) - 1. \tag{7.3}$$

7.4.2 Solving for the Potential on the Coarsest Grid: `solve_cg.c`

This file contains the function `solve_cg()` which provides a direct rather than iterative solution on the coarsest grid. The solution is obtained by the use of a FFT potential solver[12]. I have also investigated the use of a direct matrix inversion approach, which is simple to implement for the specific case of a coarsest grid of $2 \times 2 \times 2$. However, for the general case of a coarsest grid of $2^n \times 2^n \times 2^n$ (with n positive integer) the FFT approach offers greater flexibility. The function `solve_cg()` acts to convert between MLAPM's data structure and that of the FFT potential solver. For details of the FFT approach I refer the reader to (Hockney and Eastwood, 1988).

7.4.3 The Gauss-Siedel Sweep: `gs.c`

The routines contained within the file `gs.c` perform the Gauss-Siedel sweeps on the grids. Red-Black ordering of the sweep is used. The hierarchy of functions takes the now familiar form of an externally visible grid-control function, `gs()`, a series of *quad* control functions, `gs_pquad()` through `gs_nquad()`, and a node operator function, `gs_node()`. The operator ∇^2 is approximated by a seven-point difference operator. Pointers to the six surrounding nodes are obtained by the function `dstar_search()` which is described in §7.4.4. I will now present the externally visible function `gs()`, the function `gs_pquad()` as an example of the control routines, and the function `gs_node()`.

[12]The FFT solver itself was kindly provided by Dr S. N. Dutta

`gs()`

```
void gs(gridls *cur_grid)
{
  /* call gs_pquad red then black */

  gs_pquad(cur_grid, cur_grid->gridptr, TRUE);
  gs_pquad(cur_grid, cur_grid->gridptr, FALSE);
```

This functions runs the two halves of the sweep in turn, indicated by the passing of the boolean values `TRUE` and `FALSE`.

```
}
```

`gs_pquad()`

This function controls the Gauss-Siedel sweep through the linked list of pquads and calls the cquad control routine `gs_cquad()` for each cquad in a block described by a pquad. In performing the sweep, `gs_pquad()` takes account of both the grid's periodicity and the red-black ordering of the sweep. Nodes which lie on the physical edges of a sub grid should not be swept over. However, nodes on the periodic edges of the box need to be swept over. To deal with the periodicity the cquad with coordinate $z = 0$ (if present) and the cquad with coordinate $z = l1dim$ are treated separately. To deal with the red-black ordering a boolean state variable, `cur_type` is used and alternately switched on and off.

```
void gs_pquad(gridls *cur_grid, pqptr fst_pquad, boolean sweep_type)
{
  pqptr cur_pquad;      /* current pquad */
  pqptr old_pquad;      /* old pquad */
  cqptr cur_cquad;      /* current cquad */
  boolean cur_type;     /* current boolean value of sweep_type */
  boolean o_e;          /* odd or even? - result of nquad test */
  long z;                /* z-coord of cur_cquad */

  /* deal with periodic case */

#ifdef PERIODIC
```

The functions in this source file are programmed with non-periodic boundary conditions in mind. All sections relating to periodic boundary conditions are surrounded by `#ifdef PERIODIC` blocks which can easily be "switched off".

```
  if(fst_pquad->z == 0){
    /* test if last cquad exists */
    for(cur_pquad = fst_pquad; cur_pquad->next != NULL; cur_pquad =
        cur_pquad->next)
      ;
    if((cur_grid->l1dim == (cur_pquad->z + cur_pquad->length)) &&
       (fst_pquad->length > 1))
      gs_cquad(cur_grid, fst_pquad, fst_pquad->loc, sweep_type, 0);
```

The algorithm for the case that the first pquad starts with coordinate $z = 0$ goes as follows. Test the first pquad in the box to see if it starts with coordinate $z = 0$, if it does, test to see if the last pquad in the computational box runs to include the coordinate $z = l1dim - 1$. If both edges of the computational box are present then treat as periodic. The algorithm is reversed for the opposite case.

```
  }
#endif
```

```
  /* loop over pquads */
  for(cur_pquad = fst_pquad; cur_pquad != NULL; old_pquad = cur_pquad,
      cur_pquad = cur_pquad->next){
    /* test first cquad in pquad for odd/even */

    if((o_e = is_even(cur_pquad->z)) == TRUE)
      cur_type = set_opposite(sweep_type);
    else
      cur_type = sweep_type;
```

At first sight this may appear incorrect as cur_type is set to equal the opposite of the input parameter sweep_type if the z coordinate of the start of the cquad block described by cur_pquad is even. This is because the end cquad is not swept over if the grid is not periodic and has already been swept over by the preceding section if it is.

```
    /* loop over inner cquads */

    for(cur_cquad = cur_pquad->loc + 1, z = cur_pquad->z + 1;
        cur_cquad < (cur_pquad->loc + cur_pquad->length - 1);
        cur_cquad++, z++){
      gs_cquad(cur_grid, cur_pquad, cur_cquad, cur_type, z);
      cur_type = set_opposite(cur_type);
```

Here the interior cquads are looped over with the function set_opposite alternating the ordering parameter cur_type.

```
    }
  }
  /* deal with periodic case */

#ifdef PERIODIC
  if(old_pquad->z + old_pquad->length == cur_grid->l1dim){
    /* test if first cquad exists */
    if(fst_pquad->z == 0){
      cur_cquad = old_pquad->loc + (old_pquad->length - 1);
      gs_cquad(cur_grid, old_pquad, cur_cquad, cur_type, z);
    }
  }
#endif
}
```

gs_node()

gs_node() updates the potential on the node cur_node using the values of the potential at the six surrounding nodes forward, back, left, right, above and below) and the density at cur_node (recall equation (4.19)) . A diff_star structure is used to hold pointers to the set of six nodes and the function dstar_search() is used to locate them.

```
void gs_node(gridls *cur_grid, pqptr cur_pquad, cqptr cur_cquad,
             nqptr cur_nquad, nptr cur_node, long z, long y, long x)
{
  struct diff_star dstar;  /* structure containing six surrounding nodes */
  struct diff_star *dstar_ptr;   /* ptr to ... */

  /* search for pointers if any are null return NULL */
  dstar_ptr = dstar_search(cur_grid, cur_pquad, cur_cquad, cur_nquad,
                           cur_node, z, y, x, &dstar);

  /* difference */
  if(dstar_ptr != NULL)
```

```
    cur_node->pot = 0.1666666666667 *
      (dstar_ptr->nfor->pot + dstar_ptr->nback->pot  +
       dstar_ptr->nleft->pot + dstar_ptr->nright->pot +
       dstar_ptr->nup->pot + dstar_ptr->ndown->pot -
       (cur_node->dens * cur_grid->grid_spac_sqrd));
}
```

7.4.4 7 - point operator node search: dstar_search.c

The file `dstar_search.c` contains the source for the function `dstar_search()` which finds pointers to the six surrounding nodes required to complete a seven-point difference operation. As noted above `dstar_search()` returns the pointers in a special structure `diff_star`. Both the periodic and non-periodic cases are coded with the usual `#ifdef PERIODIC` block. The code is presented below with annotations, only the periodic case is shown.

```
diff_star_ptr dstar_search(gridls *cur_grid, pqptr cur_pquad,
                           cqptr cur_cquad, nqptr cur_nquad, nptr
                           cur_node, int z, int y, int x,
                           diff_star_ptr dstar_ptr)
{
  pqptr s_pquad;      /* pquad to search for nodes */
  cqptr s_cquad;      /* cquad to search for nodes */
  nqptr s_nquad;      /* nquad to search for nodes */
  float endx;
  float endy;

#ifdef PERIODIC
  /* get nup */
  /* periodic case */

  if(z == cur_grid->l1dim - 1){
    s_pquad = cur_grid->gridptr;
    if(s_pquad->z != 0)     /* first plane is not present */
      return NULL;
```

If any of the nodes cannot be found then `NULL` is returned. To save CPU time the function is exited immediately if a node is missing.

```
    /* find correct cquad */

    for(s_cquad = s_pquad->loc;
        (y > (endy = (s_cquad->y + s_cquad->length )))
          && (s_cquad->next != NULL); s_cquad = s_cquad->next)
      ;
    if((y >= s_cquad->y) && (y < endy)){/* is nquad in this cquad */
```

Loop through the cquads looking for one which contains the an nquad with the correct y coordinate.

```
      for(s_nquad = s_cquad->loc + (y - s_cquad->y);
          (x > (endx = (s_nquad->x + s_nquad->length )))
            && (s_nquad->next != NULL); s_nquad = s_nquad->next)
        ;
      if((x >= s_nquad->x) && (x < endx))/* is node in this nquad */
        dstar_ptr->nup = s_nquad->loc + (x - s_nquad->x);
```

Having found the node set the `nup` member of the `diff_star` structure to point to it.

```
      else
        return NULL;
    }
```

```
    else
      return NULL;
  }/* non-periodic case */
  else if(z < cur_pquad->z + cur_pquad->length - 1){
```

In the non-periodic case the initial test now checks if a cquad with the correct z coordinate lies within the current pquad. The remainder of the code is identical to that above.

```
    ⋮
  }else
    return NULL;

  /* get ndown */
  /* periodic case */
  if(z == 0){    /* we are at min boundary */

    /* search for correct node */
    for(s_pquad = cur_pquad; s_pquad->next != NULL; s_pquad =
          s_pquad->next)
      ;
    if(s_pquad->z + s_pquad->length == cur_grid->l1dim){
```

Here we are looking for the node immediately below the original node. In the periodic case this node will located on the plane with z coordinate `l1dim`. Hence the `pquad` list is looped through until the last `pquad` is located. The last pquad is then tested to see if it contains a `cquad` lying on the boundary.

```
      ⋮
  }/* non-periodic case */
  else if(z > cur_pquad->z){
```

Here in the non-periodic case I test if the cquad containing the original node is the first described by `cur_pquad`. If it is then the plane below z is not present and `NULL` should be returned.

```
     ⋮
  }else
    return NULL;

  /* get nleft */
  /* search for correct node */
  /* move to first cquad along plane */

  if(y == cur_grid->l1dim - 1){
    s_cquad = cur_pquad->loc + (z - cur_pquad->z);
```

Here I deal with the periodic case for `nleft`. If the y coordinate of the original node lies on the upper boundary of the grid then the node to it's left will lie on the lower y-boundary of the grid. `s_cquad` is then reset.

```
    for(s_nquad = s_cquad->loc;
        (x > (endx = (s_nquad->x + s_nquad->length )))
          && (s_nquad->next != NULL); s_nquad = s_nquad->next)
      ;
    if((x >= s_nquad->x) && (x < endx))/* is node in this nquad */
      dstar_ptr->nleft = s_nquad->loc + (x - s_nquad->x);
    else
      return NULL;
  }else if(y < cur_cquad->y + cur_cquad->length - 1){
```

Here I deal with the non-periodic case.

```
    for(s_nquad = cur_cquad->loc + (y - cur_cquad->y + 1);
        (x > (endx = (s_nquad->x + s_nquad->length )))
          && (s_nquad->next != NULL); s_nquad = s_nquad->next)
      ;
    if((x >= s_nquad->x) && (x < endx))/* is node in this nquad */
      dstar_ptr->nleft = s_nquad->loc + (x - s_nquad->x);
    else
      return NULL;
  }else
    return NULL;

  /* get nright */
  if(y == 0){    /* we are at min boundary */
```

The remaining nodes are found by the remainder of the function, which is not illustrated here.

```
  ⋮
  /* return dstar */
  return dstar_ptr;
}
```

7.4.5 Restriction of Fine Grids: trunc_error.c; fine_to_coarse.c

The restriction process is the process of transferring the potential and density distributions from a fine grid to the next coarser grid. After the information is transferred Gauss-Siedel iteration sweeps continue on the coarse grid. For a full exposition of this process I refer the reader to §4.2. The functions in the file `fine_to_coarse.c` perform the restriction operation. The restriction operation is controlled from the function `trunc_err()`.

An estimate of the relative truncation error between to grids can be obtained by calculating

$$\tau^k = L^{k-1}(\mathcal{R}\phi^k) - \mathcal{R}L^k\phi^k \tag{7.4}$$

To calculate this an appropriate algorithm would be

1. Restrict the fine grid potential to the coarse grid.
2. Calculate ∇^2 of this new coarse grid potential.
3. Calculate ∇^2 of the fine grid potential.
4. Restrict $\nabla^2\phi^k$ to the $k-1$ grid and sum the absolute value of τ^k

It is clear from the first step that fine to coarse interpolation of the potential must be performed before the truncation error can be calculated.

`fine_to_coarse.c`

The file `fine_to_coarse.c` contains the functions `f_to_c_pquad()`, `f_to_c_cquad()`, `f_to_c_nquad()`, `f_to_c_node_dens_only()`, `f_to_c_node_reverse()`, `f_to_c_node_pot()` and `f_to_c_node_dens()`. The ... `pquad` through ... `cquad` function hierarchy performs a loop over the fine grid nodes and a loop over the corresponding coarse grid nodes. The appropriate node function is passed to the looping hierarchy via a function pointer.

As an example of the looping mechanism I now reproduce the function `f_to_c_pquad()` with annotations.

```
void f_to_c_pquad(gridls *coa_grid, pqptr coa_pquad, gridls *fin_grid,
                  pqptr fin_pquad, void (*f_to_c_node)
                  (gridls *, pqptr, cqptr, nqptr, nptr, gridls *,
                   pqptr, cqptr, nqptr, nptr, long, long,
                  long))
```

The last argument is a pointer to a function which takes the listed arguments.

```
{
  cqptr fin_cquad;     /* pointer to fine-grid cquad */
  cqptr coa_cquad;     /* pointer to coarse-grid cquad */
  pqptr t_coa_pquad;   /* temporary coarse pquad pointer */
  cqptr t_fin_cquad;   /* temporary fine cquad pointer */
  long coa_z;  /* coarse-grid z index */
  long fin_z;  /* fine-grid z index */

  /* initialise nquad pointers */

  t_coa_pquad = coa_pquad;

  fin_cquad = fin_pquad->loc;
  fin_z = fin_pquad->z;

  /* find correct coa_cquad */

  for(coa_z = (fin_z / 2); coa_z > t_coa_pquad->z +
      t_coa_pquad->length; t_coa_pquad = t_coa_pquad->next)
    ;

  coa_cquad = t_coa_pquad->loc + (coa_z - t_coa_pquad->z);
```

The first step is to find the coarse cquad with z coordinate coa_z

```
  /* loop over fin_cquads - alternately looping over coa_cquad */

  for(; fin_cquad < (fin_pquad->loc + fin_pquad->length); fin_cquad +=
      2, fin_z += 2, coa_cquad++){
    for(t_fin_cquad = fin_cquad; t_fin_cquad != NULL; t_fin_cquad =
        t_fin_cquad->next)
      /* pass control to f_to_c_cquad */
```

This double loop runs over the block of fine cquads described by the fine pquad fin_pquad and the "linked-list" of cquads attached to each through the next pointer

```
      f_to_c_cquad(coa_grid, t_coa_pquad, coa_cquad, fin_grid,
                   fin_pquad, t_fin_cquad, fin_z,
                   (void (*)(gridls *, pqptr, cqptr, nqptr, nptr, gridls
                            *, pqptr, cqptr, nqptr, nptr, long, long, long))
                   (f_to_c_node));
  }
}
```

The node functions are passed through the above hierarchy and then called by the function `f_to_c_nquad()`. I will now discuss each of these node functions in turn.

`f_to_c_node_dens()` This function performs fine to coarse *density* interpolation. Recall from equation (4.48), that in full approximation storage mode the source term on coarse grids is not the density but

$$\rho^k = \nabla^2(\mathcal{R}\phi^{k+1}) + \mathcal{R}(\rho^{k+1} - \nabla^2\phi^{k+1}) \tag{7.5}$$

This density restriction operator calculates the value of this expression at each coarse grid location. The restriction operator is *full-weighting*, i.e. the three-dimensional version of equation (4.35). The term

$$\rho^{k+1} - \nabla^2 \phi^{k+1} \tag{7.6}$$

is the residual, this is previously calculated by the function `resid()` called by the `resid_norm()` function hierarchy. The residual is stored in the '0'th element of the `force.temp` part of the node `struct`. A code fragment is illustrated below.

```
void f_to_c_node_dens(gridls *coa_grid, pqptr coa_pquad, cqptr coa_cquad,
                      nqptr coa_nquad, nptr coa_node, gridls *fin_grid,
                      pqptr fin_pquad, cqptr fin_cquad, nqptr fin_nquad, nptr
                      fin_node, long fin_z, long fin_y, long fin_x)
{
  float residual;                 /* fine grid residuals */
  nptr tsc_nodes[3][3][3];        /* nodes to interpolate from */
  struct diff_star dstar;         /* 7pt structure */
  struct diff_star *dstar_ptr;    /* pointer to 7pt structure */

  tsc_nodes[1][1][1] = fin_node;   /* current node in centre */

  /* get pointers to tsc nodes */

  search31_pl9(fin_grid, fin_pquad, fin_cquad, fin_nquad, tsc_nodes,
               &fin_z, &fin_y, &fin_x);
```

For full-weighting pointers to the 26 nearest neighbours to the fine node are required.

```
  /* If no pointers are NULL - construct weighted density */

  if(test_tsc(tsc_nodes) == TRUE){

    /* construct full-weighted residual */

    residual = 0.125 * fin_node->force.temp[0] + 0.0625 *
      (tsc_nodes[0][1][1]->force.temp[0]  +
       tsc_nodes[1][0][1]->force.temp[0]  +
       tsc_nodes[1][1][0]->force.temp[0]  +
       tsc_nodes[2][1][1]->force.temp[0]  +
       tsc_nodes[1][2][1]->force.temp[0]  +
       tsc_nodes[1][1][2]->force.temp[0])
```

The midpoints of the faces of the 3x3x3 cube.

```
+ 0.03125
      * (tsc_nodes[0][0][1]->force.temp[0]  +
         tsc_nodes[0][2][1]->force.temp[0]  +
         tsc_nodes[2][0][1]->force.temp[0]  +
         tsc_nodes[2][2][1]->force.temp[0]  +
         tsc_nodes[0][1][0]->force.temp[0]  +
         tsc_nodes[0][1][2]->force.temp[0]  +
         tsc_nodes[2][1][0]->force.temp[0]  +
         tsc_nodes[2][1][2]->force.temp[0]  +
         tsc_nodes[1][0][0]->force.temp[0]  +
         tsc_nodes[1][0][2]->force.temp[0]  +
         tsc_nodes[1][2][0]->force.temp[0]  +
         tsc_nodes[1][2][2]->force.temp[0])
```

The midpoints of all the edges of the cube.

```
      + 0.015625 * (tsc_nodes[0][0][0]->force.temp[0] +
                    tsc_nodes[0][0][2]->force.temp[0] +
                    tsc_nodes[0][2][0]->force.temp[0] +
                    tsc_nodes[2][0][0]->force.temp[0] +
                    tsc_nodes[0][2][2]->force.temp[0] +
                    tsc_nodes[2][0][2]->force.temp[0] +
                    tsc_nodes[2][2][0]->force.temp[0] +
                    tsc_nodes[2][2][2]->force.temp[0]);
```

The corners of the cube.

```
    /* call dstar search on coarse grid */

    dstar_ptr = dstar_search(coa_grid, coa_pquad, coa_cquad,
                             coa_nquad, coa_node, (fin_z / 2),
                             (fin_y / 2), (fin_x / 2), &dstar);
```

Find the nodes for the seven point ∇^2 operator on the coarse grid.

```
    /* interpolate rhs */
    coa_node->dens = residual + del_squared(coa_grid, coa_node,
                                            dstar_ptr);
  }
}
```

`f_to_c_node_pot()` The fine grid potential has to be restricted to the coarse grid. This is done by the function `f_to_c_node_pot()`. In this case $\phi^k = \mathcal{R}\phi^{k+1}$ and the function simply constructs the full-weighted potential. The potential is stored in the second element of the `force.temp` part of the node `struct`.

```
void f_to_c_node_pot(gridls *coa_grid, pqptr coa_pquad, cqptr coa_cquad,
                     nqptr coa_nquad, nptr coa_node, gridls *fin_grid,
                     pqptr fin_pquad, cqptr fin_cquad, nqptr fin_nquad, nptr
                     fin_node, long fin_z, long fin_y, long fin_x)
{
  nptr tsc_nodes[3][3][3];  /* nodes to interpolate from */
  tsc_nodes[1][1][1] = fin_node;   /* current node in centre */

  /* get pointers to tsc nodes */

  search3l_pl9(fin_grid, fin_pquad, fin_cquad, fin_nquad, tsc_nodes,
               &fin_z, &fin_y, &fin_x);

  /* If no pointers are NULL - construct weighted density */

  if(test_tsc(tsc_nodes) == TRUE){

    /* construct full weighted restricted density */

    coa_node->pot = 0.125 * fin_node->pot + 0.0625 *
      ⋮
    coa_node->force.temp[1] = coa_node->pot;
  }else
    coa_node->force.temp[1] = coa_node->pot;
```

If any of the `tsc_nodes` pointers are NULL, the fine grid-node potential is simply copied to the coarse grid. The potentials of nodes on the edges of fine grids are defined by interpolation from the coarse grid and hence are not iterated over. The potential is copied for completeness only.

```
}
```

`fine_to_c_node_dens_only()` As noted above, a density-only fine to coarse interpolation function is required to give the density on coarse grids in regions where refinement has occured. This function performs this task. The weights for each node in the cube are calculated by the `startup()` function and stored in the global variable `tsc_weights()`.

```
void f_to_c_node_dens_only(gridls *coa_grid, pqptr coa_pquad, cqptr coa_cquad,
                           nqptr coa_nquad, nptr coa_node, gridls *fin_grid,
                           pqptr fin_pquad, cqptr fin_cquad, nqptr
                           fin_nquad, nptr fin_node, long fin_z,
                           long fin_y, long fin_x)
{
  nptr tsc_nodes[3][3][3];  /* nodes to interpolate from */
  int i,j,k;                /* loop indices */
  float sumweights;         /* sum of used weights */

  tsc_nodes[1][1][1] = fin_node;   /* current node in centre */
  sumweights = 0.0;

  no_pts++;
  /* get pointers to tsc nodes */

  search3l_pl9(fin_grid, fin_pquad, fin_cquad, fin_nquad, tsc_nodes,
               &fin_z, &fin_y, &fin_x);

  /* construct full weighted restricted density */

  for(i = 0; i <= Z; i++)
    for(j = 0; j <= Z; j++)
      for(k = 0; k <= Z; k++)
        if(tsc_nodes[i][j][k] != NULL){
          coa_node->dens += tsc_nodes[i][j][k]->dens *
            tsc_weights[i][j][k];
          sumweights += tsc_weights[i][j][k];
        }

  coa_node->dens += mean_dens * sumweights;
  idens += sumweights;
}
```

`f_to_c_node_reverse()` Recall from section 7.3.3 the use of `f_to_c_node_reverse()` to add to the *fine* grid the density of coarse grid particles whose "cloud" spills over into the refined region. In this context the function sweeps over the fine grid nodes adding the coarse grid density that was not removed when particles were originally transferred to the fine grid. This density comes from particles outside the refined region. Only interior grid points are assigned to and the function `test_tsc()` is called to test for this condition. The function is illustrated below.

```
void f_to_c_node_reverse(gridls *coa_grid, pqptr coa_pquad, cqptr coa_cquad,
                         nqptr coa_nquad, nptr coa_node, gridls *fin_grid,
                         pqptr fin_pquad, cqptr fin_cquad, nqptr
                         fin_nquad, nptr fin_node, long fin_z, long fin_y,
                         long fin_x)
{
  nptr tsc_nodes[3][3][3];  /* nodes to interpolate to */
  float adj_dens;           /* density adjusted for mean dens */

  tsc_nodes[1][1][1] = fin_node;   /* current node in centre */

  /* get pointers to tsc nodes */
```

```
    search3l_pl9(fin_grid, fin_pquad, fin_cquad, fin_nquad, tsc_nodes,
                 &fin_z, &fin_y, &fin_x);

    /* If no pointers are NULL - interpolate */

    adj_dens = 8.0 * (coa_node->dens + mean_dens);

    if(test_tsc(tsc_nodes) == TRUE){

      /* if this is an interior point add coarse grid density to fine */
      /* grid */

      fin_node->dens += 0.125 * adj_dens;
      tsc_nodes[0][1][1]->dens += 0.0625 * adj_dens;
      tsc_nodes[1][0][1]->dens += 0.0625 * adj_dens;
      ⋮
      tsc_nodes[2][2][2]->dens += 0.015625 * adj_dens;

      /* set coarse density to -mean_dens */

      coa_node->dens = -mean_dens;
    }
  }
```

`trunc_err.c`

As I noted above the function `trunc_err()` in fact controls the whole process of moving the cycle from a fine to a coarse grid. Once the operations of `f_to_c_ ...` are completed the truncation error is calculated. I will illustrate the process by describing the control routine `trunc_err()` and the function which performs the calculation, `trunc_node()`. The remaining functions in this source file simply loop over the grid in a similar way to the `f_to_c_ ... quad` hierarchy.

`trunc_err()` The routine first calls the fine-to-coarse function hierarchy twice, to transfer the potential and then the density to the coarse grid. Secondly the `trunc_ ...` hierarchy is called to calculate the truncation error at each point.

```
float trunc_err(gridls *cur_grid)
{
  gridls *coa_grid;   /* coarse grid */
  gridls *fin_grid;   /* fine grid */
  pqptr coa_pquad;    /* coarse pquad */
  pqptr fin_pquad;    /* fine pquad */
  pqptr t_fin_pquad;  /* temporary fine pquad */
  float trunc_error;  /* sum of truncation error squared */
  long n_pts;     /* no points */

  /* initialize fine and coarse pointers */

  fin_grid = cur_grid;
  coa_grid = cur_grid - 1;
  fin_pquad = fin_grid->gridptr;
  coa_pquad = coa_grid->gridptr;
  trunc_error = 0.0;
  n_pts = 0;

  /* call fine to coarse */
  for(t_fin_pquad = fin_grid->gridptr; t_fin_pquad != NULL;
      t_fin_pquad = t_fin_pquad->next)
```

```
  f_to_c_pquad(coa_grid, coa_pquad, fin_grid, t_fin_pquad,
               (void (*)(gridls *, pqptr, cqptr, nqptr, nptr, gridls
                         *, pqptr, cqptr, nqptr, nptr, long,
                         long, long))(f_to_c_node_pot));
for(t_fin_pquad = fin_grid->gridptr; t_fin_pquad != NULL;
    t_fin_pquad = t_fin_pquad->next)
  f_to_c_pquad(coa_grid, coa_pquad, fin_grid, t_fin_pquad,
               (void (*)(gridls *, pqptr, cqptr, nqptr, nptr, gridls
                         *, pqptr, cqptr, nqptr, nptr, long,
                         long, long))(f_to_c_node_dens));

/* call trunc_error */

trunc_error = trunc_pquad(coa_grid, coa_pquad, fin_grid, fin_grid->gridptr,
                          trunc_error, &n_pts);

/* return trunc_error norm */
return (trunc_error /(float) n_pts);
}
```

`trunc_node()` Recall equation (4.58), an estimate of the truncation error is given by

$$\tau^{k+1} = \|\nabla^2(\mathcal{R}\phi^{k+1}) - \mathcal{R}(\nabla^2\phi^{k+1})\| \tag{7.7}$$

The reader should now compare this with equation (7.5), this shows that immediately after transferring the density to the coarse grid the truncation error on the fine grid is given by

$$\tau^{k+1} = \|\rho^k - \mathcal{R}\rho^{k+1}\| \tag{7.8}$$

```
float trunc_node(gridls *coa_grid, pqptr coa_pquad, cqptr coa_cquad,
                 nqptr coa_nquad, nptr coa_node, gridls *fin_grid,
                 pqptr fin_pquad, cqptr fin_cquad, nqptr fin_nquad, nptr
                 fin_node, long fin_z, long fin_y,
                 long fin_x, float trunc_err, long *n_pts)
{
  struct diff_star dstar;    /* dstar structure - pointers to nodes */
  struct diff_star *dstar_ptr; /* ptr to ... */
  float tau;                   /* truncation error estimate */
  float wdens;                 /* weighted density */

  /* call dstar search on fine grid */

  if((dstar_ptr = dstar_search(fin_grid, fin_pquad, fin_cquad,
                               fin_nquad, fin_node, fin_z, fin_y,
                               fin_x, &dstar)) != NULL){
    wdens = 0.5 * fin_node->dens + 0.125 *
      (dstar_ptr->nfor->dens + dstar_ptr->nback->dens +
       dstar_ptr->nleft->dens + dstar_ptr->nright->dens +
       dstar_ptr->nup->dens + dstar_ptr->ndown->dens);
    tau = wdens  - coa_node->dens;
    trunc_err += fabs(tau);
    (*n_pts)++;
  }
  return trunc_err;
}
```

7.4.6 Prolongation of Coarse Grids: coarse_to_fine.c, go_down.c, i_func.c

The process of prolongation of a coarse grid potential is controlled by the function `go_down()` and the hierarchy of functions `c_to_f_ ... quad()`. The prolongation operator is bi-linear interpolation, controlled by the eight interpolation functions `i_one()` through `i_eight()`. These eight reflect how the interpolation changes with the relative position of fine and coarse grid nodes. For fine grid nodes which are cospatial with a coarse grid node, bi-linear interpolation reduces to the potential of the fine grid being copied directly from the coarse grid. In the opposite extreme of the fine-grid node not being cospatial with any of the planes of coarse-grid nodes, the fine grid potential is given by the average of the potentials on the eight surrounding coarse grid nodes. The reader will recall a similar discussion in section 7.3.6.

`go_down.c`

The source file `go_down.c` contains the main control function `go_down()`. This function calls the coarse-to-fine function hierarchy.

```
gridls *go_down(gridls *cur_grid)
{
  gridls *coa_grid;       /* coarse grid */
  gridls *fin_grid;       /* fine grid */
  pqptr coa_pquad;        /* coarse pquad */
  pqptr fin_pquad;        /* fine pquad */

  /* initialize fine and coarse pointers */
  coa_grid = cur_grid;
  fin_grid = cur_grid + 1;
  fin_pquad = fin_grid->gridptr;
  coa_pquad = coa_grid->gridptr;

  /* call coarse to fine */
  c_to_f_pquad(coa_grid, fin_grid, coa_pquad, fin_pquad, fin_pquad);

  return fin_grid;
}
```

`coarse_to_fine.c`

The coarse-to-fine hierarchy of functions, `c_to_f_pquad()` through `c_to_f_nquad()`, control the prolongation process on the pquad to nquad levels respectively. The functions determine the relative positions of coarse and fine grid nodes and hence the appropriate interpolation function to call in a manner very similar to the way in which the `re_link` hierarchy calls the appropriate `part_search` function (see section 7.3.6). I present the function `c_to_f_nquad()` as an example.

```
void c_to_f_nquad(gridls *coa_grid, gridls *fin_grid, pqptr coa_pquad,
                  pqptr fin_pquad, cqptr coa_cquad, cqptr fin_cquad,
                  nqptr coa_nquad, nqptr fin_nquad, nqptr cur_fin_nquad,
                  long coa_z, long coa_y, long fin_z, long fin_y,
                  void (*interp_A)(gridls*, pqptr, cqptr, nqptr, nptr,
                                   gridls*, pqptr, cqptr, nqptr, nptr,
                                   long, long, long, long, long, long),
                  void (*interp_B)(gridls*, pqptr, cqptr, nqptr, nptr,
                                   gridls*, pqptr, cqptr, nqptr, nptr,
                                   long, long, long, long, long, long))
```

The two interpolate functions that will be alternated between as the code iterates down a column of nodes, are passed to `c_to_f_nquad()` as function pointers from `c_to_f_cquad()`.

```
{
  nptr fin_node;      /* pointer to fine-grid node */
  nptr coa_node;      /* pointer to coarse-grid node */
  nqptr t_coa_nquad;  /* temporary pointer to coarse-grid nquad */
  long fin_x;                          /* fine-grid x index */
  long coa_x;                          /* coarse-grid x index */
  int no_state;       /* node state - determines whether node in */
                      /* "cospatial" position or not */

  /* initialize node pointers */
  fin_node = cur_fin_nquad->loc;
  fin_x = cur_fin_nquad->x;
  coa_x = fin_x / 2;

  /* find correct coarse nquad */
  for(t_coa_nquad = coa_nquad ;coa_x > t_coa_nquad->x +
        t_coa_nquad->length; t_coa_nquad = t_coa_nquad->next)
    ;
  coa_node = t_coa_nquad->loc + (coa_x - t_coa_nquad->x);
  no_state = TRUE;

  /* loop over fin_nodes - alternately looping over coa_node */
  for(; fin_node < cur_fin_nquad->loc + cur_fin_nquad->length;
      fin_node++, fin_x++){

    /* pass control to interp functions */
    switch(no_state){
    case TRUE:
      interp_A(coa_grid, coa_pquad, coa_cquad, coa_nquad, coa_node,
               fin_grid, fin_pquad, fin_cquad, fin_nquad, fin_node,
               coa_x, coa_y, coa_z, fin_x, fin_y, fin_z);
      no_state = FALSE;
      break;
    case FALSE:
      interp_B(coa_grid, coa_pquad, coa_cquad, coa_nquad, coa_node,
               fin_grid, fin_pquad, fin_cquad, fin_nquad, fin_node,
               coa_x, coa_y, coa_z, fin_x, fin_y, fin_z);
      no_state = TRUE;
      coa_node++;
      coa_x++;
      break;
    default:
      perror("c_to_f_nquad: error in no_state switch");
      exit(1);
      break;
```

The function loops over the block of nodes described by `cur_fin_nquad`, alternately looping over the corresponding block of coarse nodes. If `fin_node` is cospatial with the plane containing `coa_node` `interp_A()` is called, otherwise `interp_B()` is called.

```
    }
  }

  /* now recursively call c_to_f_nquad on cur_fin_nquad->next */

  if(cur_fin_nquad->next != NULL){
    c_to_f_nquad(coa_grid, fin_grid, coa_pquad, fin_pquad, coa_cquad,
                 fin_cquad, coa_nquad, fin_nquad, cur_fin_nquad->next,
                 coa_z, coa_y, fin_z, fin_y,
                 (void (*)(gridls*, pqptr, cqptr, nqptr, nptr,
```

```
                              gridls*, pqptr, cqptr, nqptr, nptr,
                              long, long, long, long, long, long))(interp_A),
                    (void (*)(gridls*, pqptr, cqptr, nqptr, nptr,
                              gridls*, pqptr, cqptr, nqptr, nptr,
                              long, long, long, long, long, long))(interp_B));
  }
}
```

i_func.c

This source file contains the eight coarse-to-fine interpolation functions. To illustrate these functions I include the source of `i_eight()` the most complex of the eight. The other functions follow a similar pattern. The numbering of the functions follows that of the particle search functions and I refer the reader to figure 7.5.

```
void i_eight(gridls *coa_grid, pqptr coa_pquad, cqptr coa_cquad, nqptr
             coa_nquad, nptr coa_node, gridls *fin_grid, pqptr
             fin_pquad, cqptr fin_cquad, nqptr fin_nquad, nptr fin_node,
             long coa_x, long coa_y, long coa_z,
             long fin_x, long fin_y, long fin_z)
{
  nptr coa_nodes[8];        /* pointer to four coa_nodes */
  cqptr n_coa_cquad;        /* pointer to next coa_cquad */
  cqptr store_cquad;        /* storage pointer to cquad */
  nqptr n_coa_nquad;        /* pointer to next coa_nquad */
  nqptr store_nquad;        /* storage pointer to nquad */
  int i;                    /* loop index */
  float node_corr[8];       /* u - Ru */
  float total_corr;

  /* initialize coa_nodes[i] for all i */
```

The first step is to find pointers to all eight surrounding coarse nodes.

```
  coa_nodes[0] = coa_node;    /* bottom left */

  /* find the next node (coa_nodes[1]) */

  if(coa_x == coa_grid->l1dim - 1)  /* are we at x-boundary? */
    coa_nodes[1] = coa_nquad->loc;    /* use first node  */
  else
    coa_nodes[1] = coa_nodes[0] + 1;      /* use next node  */

  /* find the next column(coa_nodes[2]) */

  if(coa_y == coa_grid->l1dim - 1)  /* are we at the boundary? */
    n_coa_nquad = coa_cquad->loc;    /* use first nquad */
  else
    n_coa_nquad = coa_nquad + 1;     /* use next nquad */

  /* search through nquad ll to find correct nquad */
  store_nquad = n_coa_nquad;     /* store it */

  while(coa_x > n_coa_nquad->x + n_coa_nquad->length)
    n_coa_nquad = n_coa_nquad->next;

  /* find correct node */

  coa_nodes[2] = n_coa_nquad->loc + (coa_x - n_coa_nquad->x);
```

```
/* find the next node (coa_nodes[3]) */

if(coa_x == coa_grid->l1dim - 1)  /* are we at x-boundary? */
  coa_nodes[3] = store_nquad->loc;    /* use first node  */
else
  coa_nodes[3] = coa_nodes[2] + 1;      /* use next node  */

/* find correct cquad */

if(coa_z == (coa_grid->l1dim - 1))  /* are we at the z-boundary? */
  n_coa_cquad = coa_pquad->loc;     /* use first cquad */
else
  n_coa_cquad = coa_cquad + 1;      /* use next cquad */

/* search through cquad ll to find correct cquad */
store_cquad = n_coa_cquad;      /* store it */
while(coa_y > n_coa_cquad->y + n_coa_cquad->length)
  n_coa_cquad = n_coa_cquad->next;

/* find the correct nquad */

n_coa_nquad = n_coa_cquad->loc + (coa_y - n_coa_cquad->y);
store_nquad = n_coa_nquad;

/* search through nquad ll to find correct nquad */

while(coa_x > n_coa_nquad->x + n_coa_nquad->length)
  n_coa_nquad = n_coa_nquad->next;

/* find correct node */

coa_nodes[4] = n_coa_nquad->loc + (coa_x - n_coa_nquad->x);

/* find the next node (coa_nodes[5]) */

if(coa_x == coa_grid->l1dim - 1)  /* are we at x-boundary? */
  coa_nodes[5] = store_nquad->loc;    /* use first node  */
else
  coa_nodes[5] = coa_nodes[4] + 1;      /* use next node  */

/* find the next column(coa_nodes[6]) */

if(coa_y == coa_grid->l1dim - 1)  /* are we at the boundary? */
  n_coa_nquad = store_cquad->loc;    /* use first nquad */
else
  n_coa_nquad = store_nquad + 1;     /* use next nquad */

/* search through nquad ll to find correct nquad */
store_nquad = n_coa_nquad; /* store it */
while(coa_x > n_coa_nquad->x + n_coa_nquad->length)
  n_coa_nquad = n_coa_nquad->next;

/* find correct node */

coa_nodes[6] = n_coa_nquad->loc + (coa_x - n_coa_nquad->x);

/* find the next node (coa_nodes[5]) */

if(coa_x == coa_grid->l1dim - 1)  /* are we at x-boundary? */
```

```
    coa_nodes[7] = store_nquad->loc;    /* use first node  */
  else
    coa_nodes[7] = coa_nodes[6] + 1;      /* use next node  */

  for(i = 0, total_corr = 0.0; i < 8; i++){
    node_corr[i] = coa_nodes[i]->pot - coa_nodes[i]->force.temp[1];
    total_corr += node_corr[i];
  }
  total_corr *= 0.125;
```

Finally the fine node potential given by the average potential of the eight surrounding coarse grid nodes.

```
  /* now do C-F interpolation */
  fin_node->pot = fin_node->pot + total_corr;
}
```

Calculating the Residual: `resid_norm.c`

The norm of the residual must be calculated to determine if convergence has been reached on a particular grid. This is done by the set of functions in the source file `resid_norm.c` and the utility function `resid()`. Five functions are defined in the source file: `resid_norm()`, `resid_pquad()`, `resid_cquad()`, `resid_nquad()` and `resid_node()`. The `resid_` ... `quad` functions sweep over the grid in usual fashion. The function `resid_norm()` controls the routine and the function `resid_node()` calculates the absolute value of the residual at each point on the grid.

`resid_norm()`

```
float resid_node(gridls *cur_grid, pqptr cur_pquad, cqptr cur_cquad,
                 nqptr cur_nquad, nptr cur_node, float residual, long z,
                 long y, long x, long *npts)
{
  struct diff_star dstar;        /* structure containing six */
  /* surrounding nodes */
  struct diff_star *dstar_ptr;   /* ptr to ... */
  float raw_resid;               /* unsquared residual */

  /* search for pointers if any are null return NULL */
  dstar_ptr = dstar_search(cur_grid, cur_pquad, cur_cquad, cur_nquad,
                           cur_node, z, y, x, &dstar);

  /* difference */
  if(dstar_ptr != NULL){
    raw_resid = resid(cur_grid, cur_node, dstar_ptr);
    cur_node->force.temp[0] = raw_resid;
    residual += fabs(raw_resid);
    (*npts)++;
  }
```

The `npts` variable, defined in the control routine is incremented to count the number of points on the grid.

```
  return residual;
}
```

7.5 Moving the Particles

Once the potential has been determined by the multigrid solver, the particle orbits must be integrated another step. The sequence that must be followed is given below.

1. Difference the potential to give forces at the grid nodes.
2. Assign forces to the particles and update particle velocities.
3. Update the particle positions.
4. Physically move the particles to the correct cell's linked-list.

This sequence is followed by the functions

- `fdiff_grid()`
- `force_assign()`
- `update_pos()`
- `move_part()`

The situation is complicated by the adaptive grid structure, requiring force interpolation from the coarse grid. This is performed by the `fi_ ...` hierarchy of functions. In the following section I will describe these routines in more detail.

7.5.1 Obtaining the Forces from the Potential: `force_diff.c`

The gradient of the potential is taken by the function `fdiff_grid()`. In the now familiar way, the function `fdiff_grid()` is the control routine and the looping over the grid is performed by the functions `fdiff_pquad()` through `fdiff_nquad()`. The force gradient is taken by the functions `fdiff_node()` and `fdiff()`, these are the important routines here and they are presented below.

```
void fdiff_node(gridls *cur_grid, pqptr cur_pquad, cqptr cur_cquad,
                nqptr cur_nquad, nptr cur_node, long z,
                long y, long x)
{
  struct diff_star dstar;        /* structure containing six */
  /* surrounding nodes */
  struct diff_star *dstar_ptr;   /* ptr to ... */

  /* search for pointers if any are null return NULL */

  dstar_ptr = dstar_search(cur_grid, cur_pquad, cur_cquad, cur_nquad,
                           cur_node, z, y, x, &dstar);
```

The potential gradient is given by differencing the potentials on the two neighbouring nodes along each axis. Hence the function `dstar_search()` is used to find the neighbouring nodes.

```
  /* difference */

  if(dstar_ptr != NULL)
    fdiff(cur_grid, cur_node, dstar_ptr);
```

The function `fdiff()` constructs the finite difference approximation. If the `dstar_ptr` returned from `dstar_search()` is NULL, then `cur_node` lies on the edge of a sub-grid and the forces are obtained by interpolation from the next coarser grid.

```
}

void fdiff(gridls *cur_grid, nptr cur_node, struct diff_star
          *dstar_ptr)
{
  cur_node->force.forces[X] = (dstar_ptr->nback->pot -
                               dstar_ptr->nfor->pot) /
```

```
      (2 * cur_grid->grid_spac);
    cur_node->force.forces[Y] = (dstar_ptr->nright->pot -
                                  dstar_ptr->nleft->pot) /
      (2 * cur_grid->grid_spac);
    cur_node->force.forces[Z] = (dstar_ptr->ndown->pot -
                                  dstar_ptr->nup->pot) /
      (2 * cur_grid->grid_spac);

}
```

7.5.2 Force Interpolation for Edge Nodes: `force_interp.c`, `fi_func.c`

On the edges of sub-grids it is clearly impossible to obtain some or all the force components as some of the nodes needed for the differencing do not exist. However, as I am using triangular-shaped-cloud mass and force assignment I need to have force components on these edge nodes in order to assign forces to particles in cells lying just inside the grid. The way around this is to interpolate the force components from the next coarser grid. This has the added bonus of smoothing the forces on particles across the coarse-fine grid boundary.

The force interpolation is controlled by the set of routines `fi_pquad()` through `fi_nquad()` in the source file `force_interp.c`. The interpolation is performed by the eight functions `fi_one()` through `fi_eight()` in the source file `fi_func.c`. The code here operates in an identical way to the `c_to_f_ ...` `()` hierarchy and `i_one()` through `i_eight()`[13]. The only difference is that only edge nodes are interpolated to. To test for edge nodes the function `dstar_search()` is used, if it returns `NULL` then the current node is an edge node. This is tested for in each of the eight interpolation functions. I illustrate with `fi_one()`.

```
void fi_one(gridls *coa_grid, pqptr coa_pquad, cqptr coa_cquad, nqptr
            coa_nquad, nptr coa_node, gridls *fin_grid, pqptr
            fin_pquad, cqptr fin_cquad, nqptr fin_nquad, nptr fin_node,
            long coa_x, long coa_y, long coa_z,
            long fin_x, long fin_y, long fin_z)
{
  diff_star dstar;          /* the dstar for search */
  int coord;                /* coord index */

  if(dstar_search(fin_grid, fin_pquad, fin_cquad, fin_nquad, fin_node,
                  fin_z, fin_y, fin_x, &dstar) == NULL){
    for(coord = X; coord <= Z; coord++)
      fin_node->force.forces[coord] = coa_node->force.forces[coord];
  }
}
```

7.5.3 Force Assignment: `force_assign.c`

Force assignment is done by the `fass_ ...` hierarchy. To conserve momentum the assignment scheme must be the same as that used by mass assignment, in this case *triangular-shaped cloud.* As the particle accelerations are not stored, the function `fass_node()` updates the particle momenta. The functions `fass_nquad()`, `fass_cquad()`, `fass_pquad()` and `force_assign()` form the usual sequence of control functions. The function `fass_node()` is illustrated below.

`fass_node()`

```
void fass_node(gridls *grid_ptr, pqptr cur_pquad, cqptr cur_cquad,
               nqptr cur_nquad, nptr cur_node, long z_coord,
               long y_coord, long x_coord, double timestep, double a_coeff,
               double b_coeff, boolean first_move, double *total_u)
```

[13]See section 7.4.6

```
{
  nptr tsc_nodes[3][3][3];  /* nodes to assign to */
  partptr cur_part;         /* current particle */
  fvect xyz_coords;         /* actual float coords of node */
  fvect pn_sep;             /* particle-node separation / grid_spac */
  int i_coord;              /* coord changing index */
  int i,j,k;                /* indices for 3D arrays */
  int n;                    /* index for components */
  fvect weights[3];         /* weights in each dimension */
  float pnarg_a;            /* pn sep temp arg */
  float pnarg_b;            /* pn sep temp arg */
  float tpnarg;             /* temp to calc pn args */
  fvect temp_coords;        /* float vector - calc un_mod coords */
  float force_arr[3][3][3]; /* array for force weights */
  float force_comp[NDIM];   /* array for force components */

  if(cur_node->ll != NULL){

    tsc_nodes[1][1][1] = cur_node;   /* current node in centre */

    /* calculate realspace coords of cur_node */
    temp_coords[X] = (((float) x_coord) /
                     (float)grid_ptr->l1dim) + (float)SHIFT;
    xyz_coords[X] = modulo_one(temp_coords[X]);
    temp_coords[Y] = (((float) y_coord) /
                     (float)grid_ptr->l1dim) + (float)SHIFT;
    xyz_coords[Y] = modulo_one(temp_coords[Y]);
    temp_coords[Z] = (((float) z_coord) /
                     (float)grid_ptr->l1dim) + (float)SHIFT;
    xyz_coords[Z] = modulo_one(temp_coords[Z]);
```

The first step to finding the particle-node separation is to find the real coordinates of `cur_node`.

```
    /* get pointers to tsc nodes */

    search3l_pl9(grid_ptr, cur_pquad, cur_cquad, cur_nquad, tsc_nodes,
                 &z_coord, &y_coord, &x_coord);

    /* loop over ll */
    if(test_tsc(tsc_nodes) == TRUE){
      for(cur_part = cur_node->ll; cur_part != NULL; cur_part =
            cur_part->ll){

        /* calc fraction to be assigned to each tsc node */

        for(i_coord = 0; i_coord < NDIM; i_coord++){
          pn_sep[i_coord] = (cur_part->pos[i_coord] -
                             xyz_coords[i_coord]) *
            grid_ptr->l1dim;
          if(fabs(pn_sep[i_coord]) > 0.5){
            tpnarg = cur_part->pos[i_coord] + 0.5;
            pnarg_a = modulo_one(tpnarg);
            tpnarg = xyz_coords[i_coord] + 0.5;
            pnarg_b = modulo_one(tpnarg);
            pn_sep[i_coord] = (pnarg_a - pnarg_b) * grid_ptr->l1dim;
          }
        }
```

The above section deals with the periodic case. Situations can arise where the physical coordinates of the cell centre and those of the particle can differ by of order 1.0, while the particle actually lies inside the boundaries of this cell. This can happen in cells which straddle the coordinate boundary of the box. If the particle-node separation is found to be large, both particle and node coordinates are shifted by half a grid (modulo the length of the grid). This will give the true particle node separation.

```
        for(i_coord = 0; i_coord < NDIM; i_coord++){
          weights[1][i_coord] = 0.75 - pow(pn_sep[i_coord], 2);
          weights[0][i_coord] = 0.5 * pow((0.5 - pn_sep[i_coord]), 2);
          weights[2][i_coord] = 0.5 * pow((0.5 + pn_sep[i_coord]), 2);
        }

        /* calculate weights for each of surrounding nodes */

        for(k = 0; k < 3; k++){
          for(j = 0; j < 3; j++){
            for(i = 0; i < 3; i++){
              if(tsc_nodes[k][j][i] != NULL){
                force_arr[k][j][i] = weights[k][Z] * weights[j][Y] *
                  weights[i][X];
              }
            }
          }/* assign forces to particles */
        }
```

This section of code calculates the weights to give to each of the 27 nodes involved in the *triangular-shaped cloud* assignment scheme.

```
        for(n = X; n <= Z; n++)
          force_comp[n] = 0.0;      /* initialize comp. array */
        for(k = 0; k < 3; k++)
          for(j = 0; j < 3; j++)
            for(i = 0; i < 3; i++)
              if(tsc_nodes[k][j][i] != NULL){
                for(n = X; n <= Z; n++)
                  force_comp[n] += force_arr[k][j][i] *
                    tsc_nodes[k][j][i]->force.forces[n];
                if(first_move)
                  (*total_u) += force_arr[k][j][i] * tsc_nodes[k][j][i]->pot;
              }
        for(n = X; n <= Z; n++){
#ifdef FORCE_TEST
```

Defining the constant FORTEST as a compile time option causes the forces on a particle to be stored rather than the updated momentum. This is used for testing the forces.

```
          cur_part->mom[n] = force_comp[n];
#else
          if(first_move) tpnarg = cur_part->mom[n];
          cur_part->mom[n] = (cur_part->mom[n] - ((cur_part->mom[n] * a_coeff) -
                             (b_coeff * cosmolog->force_coeff * force_comp[n]))
                             * timestep) / (1 + a_coeff * timestep);
          if(first_move){
            /* accumulate in gird_ptr->new_t 8 times KE at start of timestep */
            tpnarg += cur_part->mom[n];
            grid_ptr->new_t += tpnarg*tpnarg;
          }
#endif
        }
      }
    }
  }
}
```

7.5.4 Updating the Particle positions

The update to the particle positions is done by the set of functions in the source file `update_pos.c`. The usual hierarchy of grid loop functions is present (`update_pos()` through `update_nquad()` together with the node function `update_node()` which updates the coordinates of all the particles within that cell. The first question that must be asked is why loop over the grid at all? This arises from the use of multiple timesteps, as each grid has a different timestep the particles within each grid are moved independently. The only way to identify which particles are in a grid is to go through the particle linked-lists on that grid. The particles are not physically moved by `update_node()`, that is left to the functions in `move_part.c`.

```
void update_node(nptr cur_node, float timestep)
{
  partptr cur_part;       /* current particle in ll */
  int i;                  /* coord loop index */
  float fTimestepFrac;    /* fraction of timestep to do */

  /* loop over ll */

  /* zero new_ll pointer */
  cur_node->force.new_ll = NULL;

  for(cur_part = cur_node->ll; cur_part != NULL; cur_part =
      cur_part->ll){
    for(i = X; i <= Z; i++){
      /* calculate the remaining full timestep to do with two (!) drifts */
      fTimestepFrac = (pCurGrid->time_level + timestep - cur_part->stepsize);

      /* update the grid coordinates */
      cur_part->pos[i] += cur_part->mom[i] * fTimestepFrac;
      cur_part->pos[i] = modulo_one(cur_part->pos[i]);
    }
    /* update the stepsize */
    cur_part->stepsize += fTimestepFrac;
```

stepsize contains the fraction of a full timestep that this particle has move so far this timestep.

```
  }
}
```

7.5.5 Moving the Particles

All that remains to be done in the timestep is to move the particles to the correct linked-list. The is performed by the routines of the source files `move_part.c` and `new_ll2ll.c`. The particles in each cell are moved to their new locations by the function `move_node()` with the remainder of the `move_pquad()` through `move_nquad()` looping through the grid. Particles are attached to a temporary head-of-chain `new_ll` in the `force` union part of the particle structure (see chapter 5). Once all the particles have been moved to their new cell linked-lists, the lists are moved back to the original head-of-chain `ll` by `new_ll2ll_nquad()`. The functions `move_node()` and `new_ll2ll_nquad()` are illustrated below.

```
void move_node(gridls *grid_ptr, pqptr cur_pquad, cqptr cur_cquad,
               nqptr cur_nquad, nptr cur_node, long z_coord,
               long y_coord, long x_coord)
{
  nptr sur_nodes[3][3][3];  /* nodes to move particles to */
  partptr cur_part;         /* current particle */
  fvect xyz_coords;         /* actual float coords of node */
  fvect temp_coords;        /* un-modulo-ed coords */
```

```
  int new_loc[NDIM];        /* array for ints showing new location */
  int i;                    /* loop index (coords) */
  float pnarg_a;            /* pn sep temp arg */
  float pnarg_b;            /* pn sep temp arg */
  float tpnarg;             /* temp to calc pn args */
  fvect pn_sep;             /* particle - node separation */

  if(cur_node->ll != NULL){
```

If the ll is NULL (ie the cell is empty of particles) we do not progress further.

```
    sur_nodes[1][1][1] = cur_node;   /* current node in centre */

    /* calculate realspace coords of cur_node */
    temp_coords[X] = (((float) x_coord) /
                     (float)grid_ptr->l1dim) + (float)SHIFT;
    xyz_coords[X] = (float)modulo_one((double)temp_coords[X]);
    temp_coords[Y] = (((float) y_coord) /
                     (float)grid_ptr->l1dim) + (float)SHIFT;
    xyz_coords[Y] = (float)modulo_one((double)temp_coords[Y]);
    temp_coords[Z] = (((float) z_coord) /
                     (float)grid_ptr->l1dim) + (float)SHIFT;
    xyz_coords[Z] = (float)modulo_one((double)temp_coords[Z]);

    /* get pointers to surrounding nodes */

    search31_p19(grid_ptr, cur_pquad, cur_cquad, cur_nquad, sur_nodes,
                 &z_coord, &y_coord, &x_coord);

    /* loop over ll */
```

The first step is to calculate the x, y, and z indices of the new cell containing each particle.

```
    for(cur_part = cur_node->ll; cur_part != NULL;){
      /* calc new location for cur_part */
      for(i = X; i <= Z; i++){
        pn_sep[i] = (cur_part->pos[i] - xyz_coords[i]) *
          grid_ptr->l1dim;
        if(fabs((double)pn_sep[i]) > (0.5 * grid_ptr->l1dim)){
          tpnarg = cur_part->pos[i] + 0.5;
          pnarg_a = (float)modulo_one((double)tpnarg);
          tpnarg = xyz_coords[i] + 0.5;
          pnarg_b = (float)modulo_one((double)tpnarg);
          pn_sep[i] = (pnarg_a - pnarg_b) * grid_ptr->l1dim;
        }
        new_loc[i] = (int) (pn_sep[i] + 1.5);
```

One the coordinates of the new cell have been found the particle must be moved to the new linked list. First we check to see if the particle has not moved cell or moved to on of the surrounding 26 nodes (in which case `search31_p19()` will have provided pointers to those cells). If the particle remains in the same cell or has moved to a neighbouring cell, linking is performed by the function `x_link()` (illustrated below). If, however, the particle has moved further the function `extended_ll` is called to do the linking.

```
      }
      /* link to new ll in new cell */

      if((new_loc[X] < 0) || (new_loc[X] > 2) ||
         (new_loc[Y] < 0) || (new_loc[Y] > 2) ||
         (new_loc[Z] < 0) || (new_loc[Z] > 2))
```

```
          extended_ll(&cur_part, grid_ptr, grid_ptr);
        else{
          if(sur_nodes[new_loc[Z]][new_loc[Y]][new_loc[X]] != NULL)
            x_link(sur_nodes[new_loc[Z]][new_loc[Y]][new_loc[X]], &cur_part);
          else
            extended_ll(&cur_part, grid_ptr, grid_ptr);
        }
      }
    }
  }

void newll2ll_nquad(gridls *grid_ptr, pqptr cur_pquad, cqptr
    cur_cquad, nqptr cur_nquad)
{
  nptr cur_node;          /* current node */

  /* loop over nodes */
  for(cur_node = cur_nquad->loc; cur_node <
cur_nquad->loc + cur_nquad->length; cur_node++){
    cur_node->ll = cur_node->force.new_ll;
```

It is convenient to set several parts of the node structure back to zero at this point.

```
    cur_node->force.temp[0] = 0.0;
    cur_node->force.temp[1] = 0.0;
  }
}
```

Linking particles to a new cell: `x_link()` & `extended_ll()`

Two functions are provided for the process of linking particles to a new cell. The first of these, `x_link()`, deals with the restricted case of particles not moving cells or moving to one of the 26 surrounding cells. `xlink()` is more efficient that the function `extended_ll()` which deals with the general case of linking to any cell on this grid or on a coarser grid if no cell exists on the current level of refinement. The functions are illustrated below.

```
void extended_ll(partptr *cur_part, gridls *cur_grid, gridls *ini_grid)
{
  long n[NDIM];                                /* particle - node coords */
  int i;                     /* increment index */
  pqptr cur_pquad;           /* current pquad */
  cqptr cur_cquad;           /* current cquad */
  nqptr cur_nquad;           /* current nquad */
  nptr cur_node;             /* current node */
  int exp;                   /* exponent for frexp */
  float edge_shift;          /* coords of edges of first cell */

  /* find cell containing particle */
  frexp((double) (cur_grid->l1dim / MAXDL1D) , &exp);
  exp -= 1;
  if(exp == 0)
    edge_shift = 0.0;
  else{
    for(i = 1, edge_shift = 0.0; i <= exp; i++)
      edge_shift += (float)pow((double)2, (1 - exp));
    edge_shift /= (4 * MAXDL1D);
```

This first section is to determine which grid `cur_grid` is and hence determines the edge shift (see figure 5.8)

```
}

for(i = X; i <= Z; i++)
  n[i] = (long) (cur_grid->l1dim * (float)modulo_one((double)(*cur_part)->pos[i] -
                                                  edge_shift));
/* find correct pquad */
for(cur_pquad = cur_grid->gridptr;
    (n[Z] > cur_pquad->z + cur_pquad->length)
      && (cur_pquad->next != NULL); cur_pquad = cur_pquad->next)
  ;
if((n[Z] >= cur_pquad->z) && (n[Z] < cur_pquad->z +
                              cur_pquad->length)){
  for(cur_cquad = cur_pquad->loc + (n[Z] - cur_pquad->z);
      (n[Y] > cur_cquad->y + cur_cquad->length)
        && (cur_cquad->next != NULL); cur_cquad = cur_cquad->next)
    ;
  if((n[Y] >= cur_cquad->y) && (n[Y] < cur_cquad->y +
                                cur_cquad->length)){
    for(cur_nquad = cur_cquad->loc + (n[Y] - cur_cquad->y);
        (n[X] > cur_nquad->x + cur_nquad->length)
          && (cur_nquad->next != NULL); cur_nquad = cur_nquad->next)
      ;
    if((n[X] >= cur_nquad->x) && (n[X] < cur_nquad->x +
                                  cur_nquad->length)){
      cur_node = cur_nquad->loc + (n[X] - cur_nquad->x);
      {
        partptr *tmp_pptr;            /* temp. particle pointer */
        partptr mem_pptr;             /* stores tmp_pptr->ll */
        partptr *hoc_ptr;             /* pointer to head-of -chain */

        if(cur_grid == ini_grid)
          hoc_ptr = &(cur_node->force.new_ll);       /* initialize hoc_ptr */
        else
          hoc_ptr = &(cur_node->ll);

        /* assign mass */
        massign(cur_grid, cur_pquad, cur_cquad, cur_nquad, cur_node,
                n[Z], n[Y], n[X], *cur_part);
```

Mass is assigned for each particle transferred in this way. This mass assignment is ignored for particles moved to a distant location in the same grid (recall the function `new_ll2ll_nquad()` above) but is essential for particles moved to coarser grids.

```
        /* if at end of current linked-list add current particle */
        if(*hoc_ptr == NULL){
          *hoc_ptr = (*cur_part);
          (*cur_part) = (*cur_part)->ll;
          (*hoc_ptr)->ll = NULL;
        }
        /* if particles x-coord greater than that of current particle, */
        /* increment */
        else{
          for(tmp_pptr = hoc_ptr ; (*tmp_pptr != NULL) &&
                ((*cur_part)->pos[X] > (*tmp_pptr)->pos[X]) ; tmp_pptr =
                &(*tmp_pptr)->ll)
            ;
          mem_pptr = *tmp_pptr;
          *tmp_pptr = (*cur_part);
          (*cur_part) = (*cur_part)->ll;
          (*tmp_pptr)->ll = mem_pptr;
```

```
          }
        }
      }else{
        (*cur_part)->stepsize = 1;
        extended_ll(cur_part, (cur_grid - 1), ini_grid);
```

If the cell the particle is to be moved to does not exist, `extended_ll()` is called recursively on the next coarser grid.

```
      }
    }else{
      (*cur_part)->stepsize = 1;
      extended_ll(cur_part, (cur_grid - 1), ini_grid);
    }
  }else{
    (*cur_part)->stepsize = 1;
    extended_ll(cur_part, (cur_grid - 1), ini_grid);
  }
}

/* x_link: link into new, x-sorted linked-lists */
void x_link(nptr new_node, partptr *cur_part)
{
  partptr *tmp_pptr;            /* temp. particle pointer */
  partptr mem_pptr;             /* stores tmp_pptr->ll */
  partptr *hoc_ptr;             /* pointer to head-of -chain */

  hoc_ptr = &(new_node->force.new_ll);       /* initialize hoc_ptr */

  /* if at end of current linked-list add current particle */
  if(*hoc_ptr == NULL){
    *hoc_ptr = (*cur_part);
    (*cur_part) = (*cur_part)->ll;
    (*hoc_ptr)->ll = NULL;
  }

  /* if particles x-coord greater than that of current particle, */
  /* increment */
  else{
    for(tmp_pptr = hoc_ptr ; (*tmp_pptr != NULL) &&
          ((*cur_part)->pos[X] > (*tmp_pptr)->pos[X]) ; tmp_pptr =
          &(*tmp_pptr)->ll)
      ;
    mem_pptr = *tmp_pptr;
    *tmp_pptr = (*cur_part);
    (*cur_part) = (*cur_part)->ll;
    (*tmp_pptr)->ll = mem_pptr;
  }
}
```

7.6 Control Routines

So far I have not discussed any of the top level control or input/output routines, this is remedied in this the last part of this chapter. Here I will discuss the `main()` function and the `step()` function which moves the code through the timestep sequence. In addition to these two control routines are several associated functions that, for example, keep track of energy conservation. These functions are discussed where appropriate. Finally I also discuss the input and output routines.

7.6.1 Moving through a timestep: step.c

The function `step()` controls the hierarchy of multiple timesteps and moves all the grids through two coarse grid timesteps. `step()` is a complex function and it is best described stage by stage which I will now do.

```
void step(partptr fst_part, long unsigned int no_part, gridls
                  **grid_list, int *no_grids, double *timestep, cosprm* cosmolog,
                  double *int_u, double *t, double *c, double inip,  double *p,
                  FILE *logfile, FILE *energyfile, int no_timestep)
```

Most of the inputs are self explanatory. The two file pointers, `*energyfile` and `*logfile`, point to files that store the energy measurements and simulation log respectively (see §7.6.2). `cosmolog` is the cosmological parameter structure containing all the main parameters. `*int_u` is the integral of the potential energy, `*c` is the measured value of the energy constant in the Layzer-Irvine equation, `*t` is the kinetic energy at the previous coarse-grid timestep. `inip` is the initial time and `*p` is the elapsed time.

```
{
  gridls *cur_grid;        /* the current grid              */
  gridls *fin_grid;        /* fine grid pointer             */
  gridls *coa_grid;        /* coarse grid pointer           */
  gridls *for_grid;        /* grid pointer in fdiff loop    */
  gridls *for1_grid;       /* grid pointer in fdiff loop    */
  pqptr t_cur_pquad;       /* temporary pquad pointer       */
  boolean refined;         /* refinement flag               */
  double cur_timestep;     /* current grid's timestep       */
  double a_p, b_p;         /* A(p) and B(p) functions       */
  double a_dot;            /* time derivative of a          */
  double a_ddot;           /* second time derivative of a   */
  double new_a;            /* new value of a                */
  double vel_a;            /* value of a to calc conversion */
  double total_u;          /* total potential energy        */
  double new_ke;           /* total ke at v(n)         */
  int exp;                 /* exponent of 2 - difference in */
                           /* grid levels                   */
  double u_to_v;           /* conversion to comoving units  */
  long unsigned int nPart; /* particle index                */

  /* generate adaptive grid structure */
#ifndef NOREFINEMENT
```

The `NOREFINEMENT` is a symbol which may be defined at compilation to prevent the code from using any refinements, i.e. turning MLAPM into a standard PM code.

```
  refined = YES;
  while(refined != NO && *no_grids <= MAXNOGRIDS)
    refined = gen_adaptive_grid(grid_list, no_grids);
```

The previously described function, `gen_adaptive_grid()` (see §7.3.3), tests for an additional level of refinement and generates it, if required. The function returns FALSE when no refinement has been generated and the while loop ends.

```
  printf("No grids: %i\n",*no_grids);
#endif

  /* locate domain grid */
  dom_grid = find_dom_grid(*grid_list);
```

The function `find_dom_grid()` returns a pointer to the domain grid.

```
/* enter multiple timestep loop */
total_u = 0.0;    /* zero potential energy */
while(smallest_counter(*grid_list, *no_grids) < 1.0){
```

The main timestep loop starts at this point. The body of the loop contains the necessary processing to move a grid through a timestep irrespective of what level of refinement that grid represents. To determine the correct ordering of the timesteps for each level (The reader will recall the interlacing timestep scheme of §6.5.1) a counter is kept on each grid, counter is incremented by

$$1.0/2.0^{(\text{no. of grids below domain grid})}$$

for each timestep on that grid. The function, `smallest_counter()`, checks the counters on each grid, when the smallest is greater than 1.0 all the grid levels have been processed and the loop is exited. Immediately below the start of the loop is a call to the function `find_next_grid()` which is responsible for finding the next grid to process in the timestep hierarchy, this function is illustrated later in this section.

```
/* find the grid for the next timestep */
cur_grid = find_next_grid(*grid_list, *no_grids);

/* calc the timestep for this level */
cur_timestep = (*timestep * MAXDL1D) / cur_grid->l1dim;

/* calc a_dot and a_ddot */
a_dot = calc_a_dot(cur_grid->a, cosmolog);
a_ddot = calc_a_ddot(cur_grid->a, cosmolog);

/* drift */
```

Here we do the first half of the timestep, we move the particles using the particle velocities at the end of the timestep (see §6.5.3).

```
update_pos(cur_grid, (cur_timestep / 2.0));
cur_grid->time_level += cur_timestep / 2.0;
cur_grid->a = update_a(*p + cur_grid->time_level);
move_part(cur_grid);
newll2ll(cur_grid);

/* mass assignment */
/* restore old density field to all grids */
for(coa_grid = *grid_list + *no_grids - 2; coa_grid >= dom_grid;
    coa_grid--)
  restore_dens(coa_grid);

if(cur_grid != dom_grid)
  relink_bk_edge((cur_grid - 1), cur_grid);

/* assign mass on this grid*/
zero_dens(cur_grid);
assign(cur_grid);
mass_to_dens(cur_grid);

/* store coarse grid density */
for(coa_grid = *grid_list + *no_grids - 2; coa_grid >= dom_grid;
    coa_grid--)
  store_dens(coa_grid);

repack(*grid_list, *no_grids);

mg_control(*grid_list, (*no_grids));
```

Here we see the call the the multigrid solver control routine, mg_control, to solve for the potential.

```
#ifdef VERBOSE
    printf("left mg_control\n");
#endif
    for(for_grid = dom_grid; for_grid < *grid_list + *no_grids; for_grid++){
      fdiff_grid(for_grid);
      if(for_grid != dom_grid){
        /* if not the domain grid interpolate the force */
        fi_pquad((for_grid - 1), for_grid, (for_grid - 1)->gridptr,
                 for_grid->gridptr, for_grid->gridptr);
        relink_bk_edge((for_grid - 1), for_grid);
      }
    }
```

After solving for the potential the next step is to difference the potential to get the forces on the grid. The above loop does this by calling fdiff_grid on each grid from the domain grid down. On adaptive grids it is essential to interpolate the force to the edges of the grids, this is done by the function fi_pquad. The remaining function call in the block is to relink_bk_edge() which re-links particles on the edges of sub-grids back to the grid above, ready to be moved with the rest of the particles on that grid.

```
    /* assign forces and move the particles */
    a_p = calc_a_p(cur_grid->a, a_dot, a_ddot);
    b_p = calc_b_p(cur_grid->a, a_dot);
    if(cur_grid->counter == 0.){
      total_u = 0.0;
      cur_grid->old_t = cur_grid->new_t;  /* update the old ke value */
      cur_grid->new_t = 0.0;              /* zero new ke value */
    }

    /* kick */
```

After finding the forces we now *kick* the particles by updating their velocities.

```
    force_assign(cur_grid, cur_timestep, a_p, b_p, &total_u, cosmolog);

    /* drift */
```

The particles now complete their drift to the end of the timestep. Various book-keeping steps are also performed at this point. The correct values of the variables a_p and b_p are calculated before they are used by force_assign and the grid counter is updated now the timestep has been performed. The reader will also notice the resetting of the kinetic energy parts of the grid structure and the calculating of the code-to-realspace velocity conversion factor, u_to_v.

```
    update_pos(cur_grid, (cur_timestep / 2.0));
    cur_grid->time_level += cur_timestep / 2.0;
    cur_grid->a = update_a(*p + cur_grid->time_level);

    /* restore old density field to all grids */
    for(coa_grid = *grid_list + *no_grids - 2; coa_grid >= dom_grid;
        coa_grid--)
      restore_dens(coa_grid);
```

With the use of multiple timesteps it is necessary to store the density at each grid node. As mass assignment is only performed on a grid at the start of the timestep cycle or when a grid has had a timestep, the original density field has to be restored to coarse grids to compensate for the modifications to the coarse grid density field done by the multigrid solver.

```
    /* move the particles */
    move_part(cur_grid);
```

```
    newll2ll(cur_grid);

    /* update counters */
    if(cur_grid->counter == 0.)
      cur_grid->u = total_u;
    cur_grid->counter += (1.0 * MAXDL1D / cur_grid->l1dim);

    /* assign mass */
    if(cur_grid != dom_grid)
      relink_bk_edge((cur_grid - 1), cur_grid);
    zero_dens(cur_grid);
    assign(cur_grid);
    mass_to_dens(cur_grid);

    /* store coarse grid density */
    for(--cur_grid; cur_grid >= dom_grid; cur_grid--)
      store_dens(cur_grid);

    /* interpolate density back to coarser grids */
    repack(*grid_list,*no_grids);
```

The `repack` function performs a number of book-keeping interpolation operations.

```
  }

  /* calculate u and c */
  vel_a = pow(*p, ((double) 1.0 / (double)P_EXPON));

  u_to_v = P_EXPON * pow(vel_a, (double)(P_EXPON - 1))
    * calc_a_dot(vel_a, cosmolog);
  for(total_u = 0.0, new_ke = 0.0, cur_grid = dom_grid;
      cur_grid < *grid_list + *no_grids; cur_grid++){
    total_u += cur_grid->u;
    new_ke += cur_grid->new_t;
    cur_grid->counter = 0.0;
    cur_grid->u = 0.0;
  }
```

Now the main timestep loop has ended, some tidying up remains on the domain grid. Above we see a loop which updates the total potential and kinetic energy variables. In the remainder of `step()` all particles are re-linked back to the domain grid and all adaptive sub-grids are destroyed. Mass assignment then takes place on the domain grid. Finally energy conservation is checked by the function `layzer_irvine()`.

```
  /* with have 8 times ke in new_ke - see force_assign */
  new_ke *= 0.125 * u_to_v * u_to_v * vel_a * vel_a;
  *t= new_ke;
  total_u *= 0.75 * cosmolog->omega  / vel_a ;
  *c = layzer_irvine(vel_a, total_u, int_u, *timestep, new_ke,
                     logfile, energyfile, *no_grids, no_timestep);

  /* zero dom grid counter */
  dom_grid->time_level = 0.;

  /* synchronise a and update p */
  *p = inip + (no_timestep + 1) * *timestep;

  new_a = pow(*p, ((double) 1.0 / (double)P_EXPON));

  for(cur_grid = dom_grid; cur_grid < *grid_list + *no_grids; cur_grid++)
```

```
    cur_grid->a = new_a;

  /* calc a_dot and a_ddot */
  a_dot = calc_a_dot(new_a, cosmolog);
  a_ddot = calc_a_ddot(new_a, cosmolog);

  /* relink all particles back to domgrid */
  for(fin_grid = *grid_list + (*no_grids - 1),
      coa_grid = (fin_grid - 1); fin_grid != dom_grid;){
    relink_back(coa_grid, fin_grid);
    free_grid(grid_list, fin_grid, no_grids);
    /* re-init fin_grid, coa_grid, dom_grid */
    fin_grid = *grid_list + (*no_grids - 1);
    coa_grid = fin_grid - 1;
    dom_grid = find_dom_grid(*grid_list);
  }
  zero_dens(*grid_list + (*no_grids - 1));
  assign(*grid_list + (*no_grids - 1));
  mass_to_dens((*grid_list + (*no_grids - 1)));

  /* loop over particles setting stepsize to zero */
  for(nPart = 0; nPart < no_part; nPart++){
    fst_part[nPart].stepsize = 0.0;
  }
}
```

Support functions for `step()`

The main support function for `step()` is `find_next_grid`, which also has an associated function `time_sync_below()`. These functions determine the next grid to be processed via the multiple-timestep loop. The reader will recall from §6.5.1 that each timestep proceeds only if the all the timesteps on grids finer than the current one are synchronized at the midpoint of the timestep to come on the current grid. This is the condition these two functions look for. The functions are presented below.

```
boolean time_syn_below(gridls *grid_list, gridls *cur_grid, int no_grids)
{
  gridls *next_grid;             /* pointer to the next grid */

  /* test if cur_grid is finest grid - return next coarser grid */
  if(cur_grid == (grid_list + (no_grids - 1)))
    return TRUE;

  /* test time synchronization */
  else if(((cur_grid + 1)->counter == cur_grid->counter) &&
          (time_syn_below(grid_list, (cur_grid + 1), no_grids) == TRUE))
    return TRUE;
  else
    return FALSE;
}

gridls *find_next_grid(gridls *grid_list, long
                       unsigned int no_grids)
{
  gridls *fin_grid;            /* pointer to finer grid        */
  gridls *coa_grid;            /* pointer to coarser grid      */

  if(dom_grid == grid_list + (no_grids - 1))
    return dom_grid;
```

If there has been no refinement return the domain grid.

```
  for(coa_grid = dom_grid, fin_grid = dom_grid + 1; fin_grid <
        grid_list + no_grids; fin_grid++, coa_grid++){
    if((fin_grid->counter > coa_grid->counter) &&
       (time_syn_below(grid_list, fin_grid, no_grids) == YES))
      return coa_grid;
  }
  return (grid_list + (no_grids - 1));
}
```

The `repack()` function is also used by `step()`, this function performs various book-keeping interpolation operations.

```
void repack(gridls *grid_list, int no_grids)
{
  gridls *cur_grid;
  pqptr t_cur_pquad;

  /* interpolate density back to coarser grids */
  for(cur_grid = grid_list + no_grids - 1; cur_grid > dom_grid;
      cur_grid--){
    for(t_cur_pquad = (cur_grid)->gridptr; t_cur_pquad != NULL;
        t_cur_pquad = t_cur_pquad->next){
      f_to_c_pquad((cur_grid - 1), (cur_grid - 1)->gridptr, (cur_grid),
                   t_cur_pquad,
                   (void (*)(gridls *, pqptr, cqptr, nqptr, nptr, gridls *,
                             pqptr, cqptr, nqptr, nptr, long,
                             long, long))
                   (f_to_c_node_reverse));
    }
```

This loop corrects for errors in the assignment of particles with clouds that overlap the boundary of an adaptive sub-grid (See the discussion of the file `fine_to_coarse.c` in §7.4.5).

```
    /* interpolate density to domain grid */
    for(t_cur_pquad = cur_grid->gridptr; t_cur_pquad != NULL;
        t_cur_pquad = t_cur_pquad->next){
      f_to_c_pquad((cur_grid - 1), (cur_grid - 1)->gridptr, cur_grid,
                   t_cur_pquad,
                   (void (*)(gridls *, pqptr, cqptr, nqptr,
                             nptr, gridls *, pqptr, cqptr,
                             nqptr, nptr, long, long, long))
                   (f_to_c_node_dens_only));
    }
    zero_pot(cur_grid);
  }
  zero_pot(dom_grid);
}
```

7.6.2 Energy conservation: `layzer_irvine()`

The function `layzer_irvine()` calculates all of the terms in the Layzer-Irvine cosmic energy equation and hence keeps track of the energy conservation of the code. The results of the calculations are store in the file accessed through the file pointer `energyfile`. This function also writes the timestep log entry. The remaining input parameters are described with the function below.

```
double layzer_irvine(double a, double w, double *intX, double timestep,
                     double k, FILE *logfile, FILE *energyfile, int no_grids,
                     int no_timestep)
```

The input parameters are as follows: a is the scale factor, w is the potential energy at the end of the current coarse grid timestep, int_X is the integrated energy up to the start of the current coarse grid timestep, timestep is the length of the timestep and k contains the kinetic energy of all the particles at the start of the current timestep.

```
{
  double da;            /*increment in a */
  double econst;    /* energy constant */
  double a2_w;       /* a times u */
  double a2_k;      /* a to power 4 times t */
  double resolution;/* virtual resolution of finest grid */

  da = timestep/(P_EXPON*pow(a,P_EXPON-1));

  /* calculate a times u */
  a2_w = a * a * w;

  /* calculate constant */
  a2_k = a * a * k;
  econst =  a2_k + a2_w - (*intX);

  /* calc the resolution */
  resolution = pow(2.0, no_grids);

  /* print out the parameters */
  fprintf(logfile, "\n");
  fprintf(logfile, "Timestep no. = %d\n", no_timestep);
  fprintf(logfile, "Scale Factor = %f, Timestep Size = %f\n", a, timestep);
  fprintf(logfile, "Energy Constant = %f\n", econst);
  fprintf(logfile, "a^K = %f, a^W = %f, IK = %f\n", a2_k, a2_w, *intX);
  fprintf(logfile, "Virtual resolution of finest grid = %f\n", resolution);
  fprintf(logfile, "------------------------------------\n");
  fflush(logfile);

  /* print out to ascii file - easy to analyse */
  fprintf(energyfile, "%f %f %f %f\n", a2_k, a2_w, *intX, econst);

  *intX += a * w * da;

  return econst;
}
```

7.6.3 main(): mlapm_main.c

The main routine is of course the routine which controls the simulation run. Here main initializes the simulation run through calls to startrun() and input(), runs through the timestep sequence with calls to step(), and calls the output routine output() at the appropriate timesteps.

```
int main()
{
  partptr fst_part;          /* pointer to first particle in list */
  gridls *grid_list;         /* pointer to list of grids          */
  int no_grids;              /* total number of grids             */
  long no_part;              /* total number of particles         */
  int no_timestep;           /* number of coarse grid timesteps   */
  int total_timesteps;       /* total no. of timesteps            */
  double a;                  /* scale factor                      */
  double timestep;           /* timestep size                     */
  cosprm cosmolog;           /* omega etc. for this universe      */
```

```
double int_u;              /* potential energy of particles(int)*/
double t;                  /* kinetic energy of particles      */
double c;                  /* total energy                     */
double p;                  /* time variable                    */
double inip;               /* initial time                     */
FILE *dumpfile;            /* pointer to dump file             */
FILE *infile;              /* pointer to input file            */
FILE *logfile;             /* contains simulation log          */
FILE *energyfile;          /* contains energy numbers (ascii)  */
char out_name[100];        /* base name of output files        */
int out_freq;              /* current number of outputs        */
int output_count;          /* output counter                   */
int output_no;             /* output number                    */

/* get the filenames */
get_filenames(&dumpfile, out_name, &infile, &logfile, &energyfile);

/* startup: input the initial data from infile */
startup(infile, &fst_part, &no_part, &a, &timestep, &cosmolog, &t,
        &c, &out_freq, &int_u, &inip, &total_timesteps);
```

The function `startup()` starts the new simulation run and calls the `input()` function to read in the initial particle data.

```
p = inip;
printf("data read for %d particles\n", no_part);

/* set particle mass so that int dx rho = M sum m_i and grid->dens
is classical overdensity delta rho/rhobar                    */
p_mass=1./(float)no_part;

/* calc the force coeff */
cosmolog.force_coeff = (1.5F * cosmolog.omega );
```

The particle mass and force coefficient are global constants and are calculated here, once the simulation parameters are known.

```
/* generate the domain grids */
grid_list = generate_grids(&no_grids, a);
dom_grid_num = (no_grids - 1) ;
dom_grid = grid_list + dom_grid_num;

/* initial link and mass assignment */
ll(no_part, fst_part, (grid_list + no_grids - 1));
```

The function `ll()` performs the initial linking of the particles to the domain grid generated by `generate_grids()` above.

```
assign((grid_list + no_grids - 1));
mass_to_dens((grid_list + no_grids - 1));
(grid_list + no_grids - 1)->new_t = t;

/* timestep loop */
no_timestep = 0;
output_count = 0;
output_no = 1;
while(no_timestep <= total_timesteps){
  if(output_count == out_freq){
    outdata(&fst_part, &a, &cosmolog, &c,
            &no_part, &output_no, &int_u, out_name);
```

Outputs are done when the counter reaches the user defined output frequency.

```
      output_count = 0;
    }
    output_count++;
    step(fst_part, no_part, &grid_list, &no_grids, &timestep, &cosmolog,
         &int_u, &t, &c, inip, &p, logfile, energyfile, no_timestep);
    a = update_a(p);
    printf("Step %d: ",no_timestep);
    no_timestep += 1;
  }

  /* final output */
  outdata(&fst_part, &a, &cosmolog, &c,
          &no_part, &output_no, &int_u, out_name);
  return 0;
}
```

`get_filenames()`

This function requests the user supply filenames for the input and output files. The function then opens the files as appropriate.

generate_grids()

This function creates the initial set of domain grids, from the 2^3-sized grid to the MAXDL1D^3-sized grid.

Starting a simulation run: startrun.c

The initial startup procedures are performed by the function `startup()`. This function calls the `input()` function to read in the particles and simulation data. `startup()` also calls upon the user to input the number of timesteps to perform and when to output the particle data. Finally, `startup()`, calculates the initial values of the mean density, $\dot{a}$, the particle kinetic energy, the initial time variable, p, and the *triangular-shaped-cloud* weights.

```
void startup(FILE *infile, partptr *fst_part, long *no_part, double *a,
             double *timestep, cosprm *cosmolog, double *t, double *c,
             int *out_freq, double *int_u, double *p,
             int *total_timestep)
{
  double a_dot;           /* time derivative of a                   */
  int i,j,k;              /* loop indices */
  float onedweights[3];   /* weights in one dim */
  float fTemp;

  /* input the data file */
  *no_part = input(fst_part, a, cosmolog, c, infile, int_u);

  /* get the length of the timestep */
  fprintf(stdout, "Please give stepsize, the number of timesteps
                   and the output frequency (in no. of timesteps)\n");
  fscanf(stdin,"%g %d %d", &fTemp, total_timestep, out_freq);
  *timestep = (double)fTemp;

  printf("Doing %d steps of size %G with %d outputs\n",
         *total_timestep,*timestep,*out_freq);

  /* set the mean density */
```

```
  mean_dens = 1.;

  /* calculate a_dot */
  a_dot = calc_a_dot(*a, cosmolog);

  /* calculate the kinetic energy */
  *t = kinetic(*fst_part, *a, a_dot, *no_part);

  /* calculate p */
  *p = pow(*a, P_EXPON);

  /* calculate tsc_weights */
  onedweights[0] = 0.25;
  onedweights[1] = 0.5;
  onedweights[2] = 0.25;
  for(k = 0; k <= Z; k++)
    for(j = 0; j <= Z; j++)
      for(i = 0; i <= Z; i++)
        tsc_weights[k][j][i] = onedweights[k] * onedweights[j] * onedweights[i];
}
```

Reading in the data: `input.c`

The function `input.c` reads in the data from an unformatted C data file. The simulation parameter values are read in from the first part of the file and once the number of particles is known, the particle array is allocated using the function `c_part()`. The last part of the file contains the initial particle position and velocities.

```
long input(partptr *fst_part, double *a,
           cosprm *cosmolog, double *c, FILE *instream,
           double *int_u)
{
  partptr cur_part;              /* current particle */
  int i,j;                       /* loop index */
  long no_part;                  /* number of particles */
  float h, omega, box_length;    /* temp floats for reading input */
  double da;

  /* read in no. of particles */
  if(fread((void *)&no_part, sizeof(long unsigned int), 1, instream)
     != 1){
    perror("input:could not read no. of particles\n");
    exit(1);
  }
```

The function `fread()` is a standard library function declared in the `stdio.h` header file. The function reads from `instream` at most *n* (1 here) objects of size *size* (`sizeof(double)` here) into the array *(*ptr) (here the variable a).

```
  /* generate the particle array */
  *fst_part = c_part(no_part);
  if(fread((void *)&da, sizeof(double), 1, instream)
     != 1){
    perror("input:could not read scale factor\n");
    exit(1);
  }
  *a = da;
  if(fread((void *)&omega, sizeof(float), 1, instream)
     != 1){
    perror("input:could not read omega\n");
```

```
    exit(1);
  }
  if(fread((void *)&h, sizeof(float), 1, instream)
     != 1){
    perror("input:could not read h\n");
    exit(1);
  }
  if(fread((void *)&box_length, sizeof(float), 1, instream)
     != 1){
    perror("input:could not read box_length\n");
    exit(1);
  }
  cosmolog->omega = omega;
  cosmolog->h = h;
  cosmolog->box_length = box_length;

  if(fread((void *)c, sizeof(double), 1, instream)
     != 1){
    perror("input:could not read energy (c)\n");
    exit(1);
  }
  if(fread((void *)int_u, sizeof(double), 1, instream)
     != 1){
    perror("input:could not read integral of u\n");
    exit(1);
  }

  cur_part = *fst_part;
  for(i = 0; i < no_part; i++){
    for(j = X; j <= Z; j++){
      fread((void*)&(cur_part->pos[j]), sizeof(float), 1, instream);
      fread((void*)&(cur_part->mom[j]), sizeof(float), 1, instream);
    }
    cur_part->stepsize = 0.;
    cur_part++;
  }
  fclose(instream);
  return no_part;
}
```

Creating the particle linked-lists: `link.c`

Once the particles have been read in, the particles must be organized into linked-lists on the domain grid. Each particle should be placed into a linked-list attached to the cell within which it resides. The linked-lists are also sorted into order of increasing x-coordinate.

```
void ll(long unsigned int npart, partptr fpart, gridls *gridlist)
{
  partptr ipart;            /* pointer to increment down the particle */
                            /* list */
  unsigned int n[NDIM];     /* particle - node coords */
  int i;                    /* increment index */
  pqptr curpquad;           /* current pquad */
  cqptr curcquad;           /* current cquad */
  nqptr curnquad;           /* current nquad */
  nptr curnode;             /* current node */
  unsigned int gridsize;    /* current grid size */
  double fpos;              /* floating point position */
  double ipos;              /* "integer" part of fpos */
```

```
  for(ipart = fpart; ipart < fpart + npart; ipart++){

    /* find cell containing particle */

    for(i = X; i <= Z; i++)
      n[i] = (unsigned int) (gridlist->l1dim * (ipart->pos[i]));

    curpquad = gridlist->gridptr;     /* find correct plane */
    curcquad = curpquad->loc;
    curcquad += n[Z];
    curnquad = curcquad->loc;       /* find correct column */
    curnquad += n[Y];
    curnode = curnquad->loc;            /* find correct vertex */
    curnode += n[X];
    {
      partptr *tmp_pptr;          /* temp. particle pointer */
      partptr mem_pptr;                 /* stores tmp_pptr->ll */
      partptr *hoc_ptr;                 /* pointer to head-of -chain */
      hoc_ptr = &(curnode->ll);         /* initialize hoc_ptr */

      /* if at end of current linked-list add current particle */
      if(*hoc_ptr == NULL){
*hoc_ptr = ipart;
ipart->ll = NULL;
      }
      /* if particles x-coord greater than that of current particle, */
      /* increment */
      else{
for(tmp_pptr = hoc_ptr ; (*tmp_pptr != NULL) &&
      (ipart->pos[X] > (*tmp_pptr)->pos[X]) ; tmp_pptr =
      &(*tmp_pptr)->ll)
  ;
mem_pptr = *tmp_pptr;
*tmp_pptr = ipart;
ipart->ll = mem_pptr;
      }
    }
  }
}
```

Outputting the Data: `output`

The function `output()` outputs the simulation data and particle coordinates and velocities to an unformatted C file. This function is essentially the inverse of the `input()` function, with the standard library function `fread()` replaced by `fwrite()`.

Chapter 8

Tests of the code

All N-body simulation codes must be throughly tested before they can be applied to cosmological simulations with any degree of confidence. This chapter describes a set of tests that have been applied to MLAPM and the results obtained from them. The first section discusses direct tests of the force law. The second section discusses *Zel'dovich plane wave tests* which compares the results of a simulation against a known analytic solution. A set of real simulations are analyzed in detail in section §8.3, and the final section contains performance information.

8.1 The force law

Before testing the code as a whole it is important to test how forces between particles behave as a function of separation and orientation with respect to the grid. As discussed in §3.3.2 Particle-Mesh type codes have a force resolution that is approximately two cell lengths and in addition show anisotropies in the force depending on the relative locations of the particles to the grid.

To test the forces we need something we can compare with directly. Naively we may expect to compare with a $1/r^2$ force law, however this ignores the effect of the periodic boundary conditions. To account for periodicity we can use *Ewald summation* (Sangster and Dixon, 1976) which calculates the correct potential in the presence of the periodic image particles.

8.1.1 Ewald Summation

Using a notation similar to that of Sangster and Dixon we label the jth image particle in the cell $\mathbf{n}$, $j\mathbf{n}$, where $\mathbf{n}$ is an integer triple describing the cell location relative the original cell. We will label the displacement vector from the ith particle in the original cell to the $j\mathbf{n}$ by $\mathbf{r}_{i,j\mathbf{n}}$ with $r_{i,j\mathbf{n}}$ being the modulus.

The force due to the image charges would follow the Newtonian force law[1]

$$\mathbf{F} = \sum_{j\mathbf{n},j\neq i} m_i m_j \frac{\mathbf{r}_{i,j\mathbf{n}}}{r^3_{i,j\mathbf{n}}}. \tag{8.1}$$

This slowly converging summation can be replaced by two rapidly converging summations

$$\mathbf{F} = \frac{1}{2\pi}\left(\mathbf{F}^{(1)} + \mathbf{F}^{(2)}\right) \tag{8.2}$$

where

$$\mathbf{F}^{(1)} = m_i \sum_{j\neq i} \sum_{\mathbf{n}} m_j \frac{\mathbf{r}_{i,j\mathbf{n}}}{r^3_{i,j\mathbf{n}}} \left\{ \mathrm{erfc}(\alpha r_{i,j\mathbf{n}}) + \frac{2\alpha}{\sqrt{\pi}} r_{i,j\mathbf{n}} \exp(-\alpha^2 r^2_{i,j\mathbf{n}}) \right\} \tag{8.3}$$

[1] With $4\pi G = 1$ in these units

and

$$\mathbf{F}^{(2)} = \frac{2m_i}{L} \sum_{j \neq i} \sum_{\mathbf{h}} m_j \frac{\mathbf{h}}{|\mathbf{h}|^2} \exp\left(-\frac{\pi^2 h^2}{\alpha^2 L^2}\right) \sin\left(\frac{2\pi}{L} \mathbf{h} \cdot \mathbf{r}_{i,j\mathbf{n}}\right). \tag{8.4}$$

In these expressions L is the length of the side of each cell, α is an arbitrary parameter and $\mathbf{h}$ is a reciprocal lattice vector such that the components are integers. Choosing $\alpha = 2/L$ allows the two summations to be truncated at $r_{i,j\mathbf{n}} \leq 2.6L$ and $|\mathbf{h}|^2 \leq 8$ respectively.

8.1.2 Test Results

To test the force a particle was placed in the centre of a 32^3 grid and the potential due to the particle (and images) calculated using the multi-grid solver. The grid was not allowed to adapt in this first test. Next test particles were placed at random positions within the grid and the force on the test particle calculated using the MLAPM force calculation. The magnitude of the force on the test particles was then compared with that calculated using Ewald summation.

The results from the force test can be seen in figure 8.1. In both the non-adaptive (a) and adaptive (b) cases the expected result is found. The force remains close to the Ewald force until the separation is less than two grid cells. Within two grid cells the MLAPM force falls off, as all PM codes of this type cannot resolve inter-particle forces for separations less than this.

For the adaptive case, in each layer of adaptive grid we expect to resolve forces down to the two cell limit. In addition we expect the forces to be smooth across the boundary of grids. Figure 8.1 (b) shows the result for a test with one layer of adaptive grid. The force follows the Ewald force down to one coarse grid cell length. A small number of particles can be seen with forces falling off around two cell lengths, this is to be expected. Test particles that remain on the coarse grid will only have the force resolution of the coarse grid, i.e. two cell lengths.

8.2 Zel'dovich Plane Wave

The *Zel'dovich plane wave* test is a popular test of N-body simulation codes. The initial conditions for the test are given by

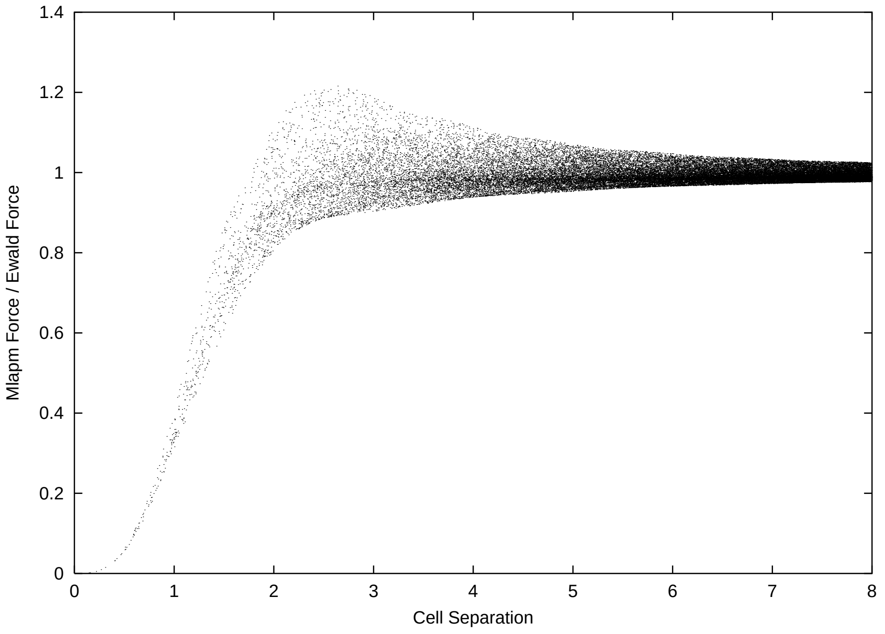
$$\mathbf{x_i^Z} = \mathbf{q_i} + \frac{\mathbf{A}}{2\pi k} \cos(2\pi \mathbf{k} \cdot \mathbf{q}) \tag{8.5}$$

$$\mathbf{v_i^Z} = \dot{\mathbf{x}}_\mathbf{i}^\mathbf{Z} \tag{8.6}$$

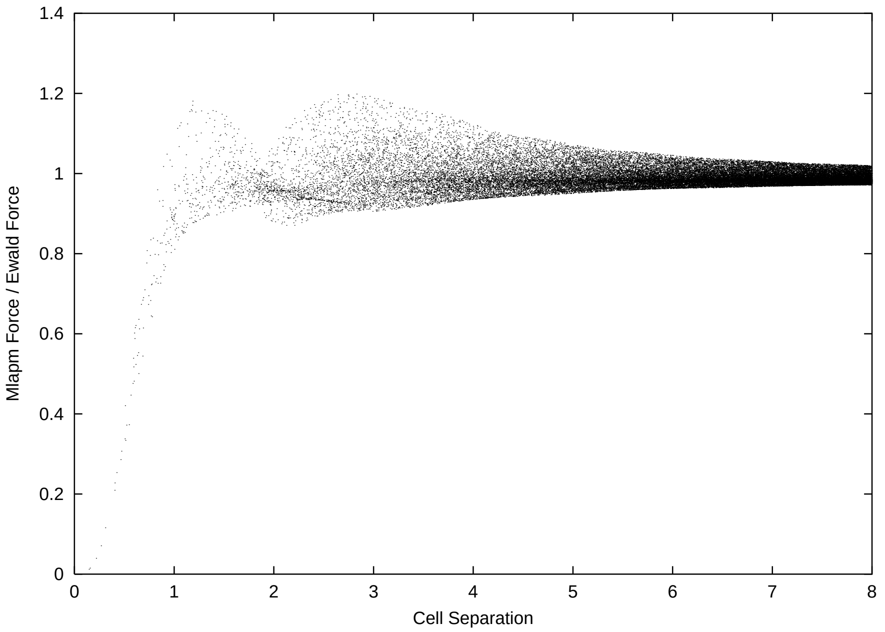
where $\mathbf{q_i}$ are the unperturbed positions, $\mathbf{k}$ is an integer triple (Efstathiou et al., 1985), (Klypin and Shandarin, 1983) and $\mathbf{A}$ is an appropriate function of time. Equations (8.5) and (8.6) are an exact solution of the equations on motion up to "shell-crossing" where infinite densities occur, this corresponds to the point where the wave amplitude $A = 1$. For MLAPM $\Omega = 1$ equations of motion are currently implemented, and in this case A is proportional to the scale factor.

Equations (8.5) and (8.6) were used as initial conditions for four separate simulations using MLAPM, two with a single wave ($k = 1$) and two with two waves ($k = 2$). Within in each pair of simulations, one simulation was allowed to have an adaptive grid and one was not. In each case the scale factor was set initially to be 0.25 and the simulation ran forward until the scale factor was approximately equal to 1.0. The waves were aligned with the simulation x axis. The wave amplitude was set so that $A = 1$ occurred when the scale factor, $a = 1$. Large timesteps of $\Delta p = 0.025$ were used, giving 35 (domain grid) timesteps for the whole simulation. In all four simulations 32^3 particles were used and the domain grid contained 32^3 cells.

Figure 8.2 contains a set of projections of the simulation box onto the x-y plane for the non-adaptive case with one wave, Figure 8.3 contains the corresponding adaptive case. The projections of the two wave case are not illustrated.

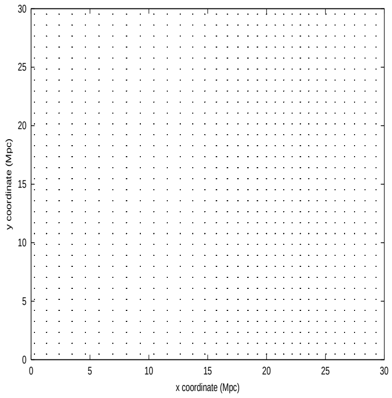


(a)

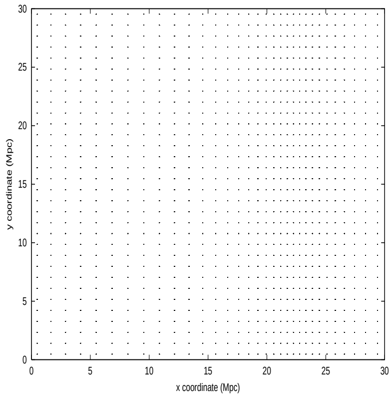


(b)

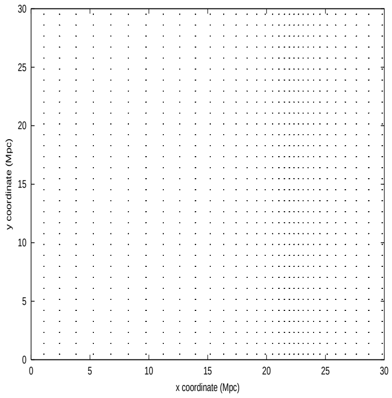
Figure 8.1: Ratio of MLAPM force to that calculated by Ewald summation plotted against particle separation (in cell lengths). (a) Non-adaptive (b) Single level of adaptive grid.



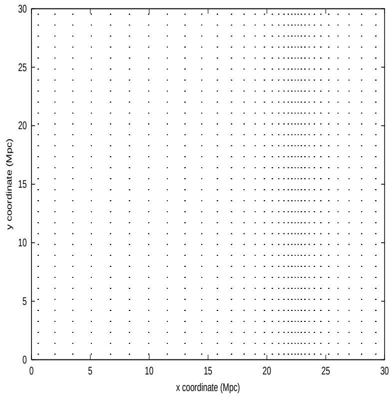
(a) a=0.2500



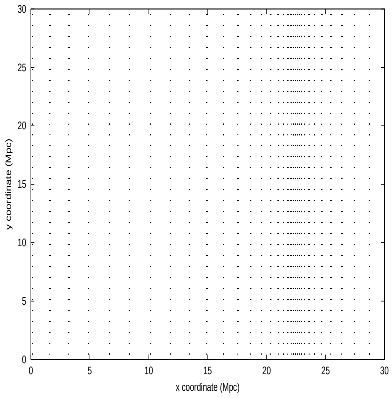
(b) a=0.4481



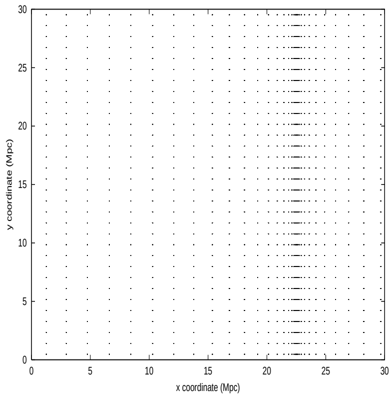
(c) a=0.6088



(d) a=0.7504

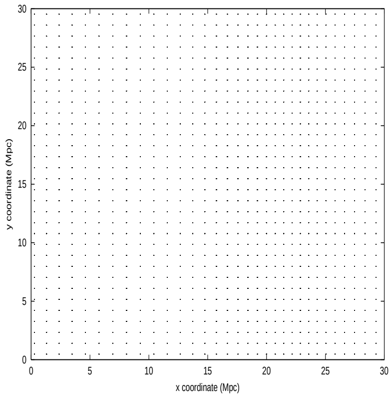


(e) a=0.8796

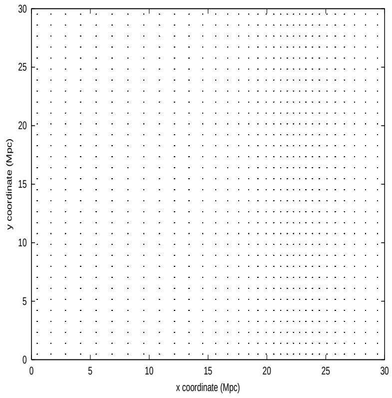


(f) a=1.0000

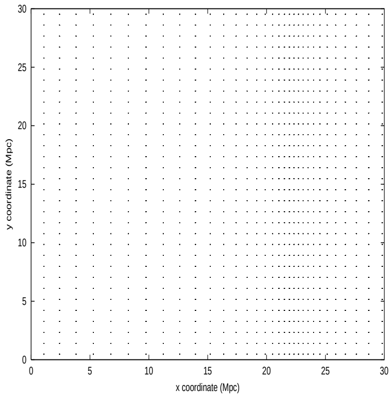
Figure 8.2: Evolution of a Zel'dovich plane wave solution using non-adaptive MLAPM. Projections onto the x-y plane for various scale factors



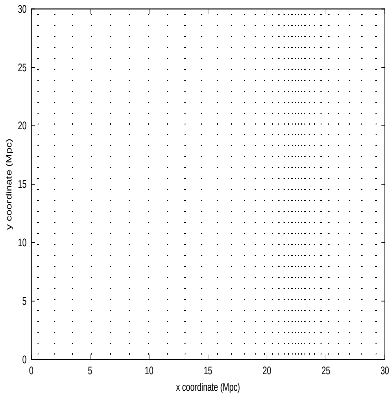
(a) a=0.2500



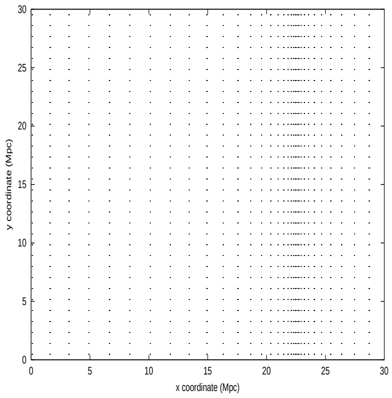
(b) a=0.4481



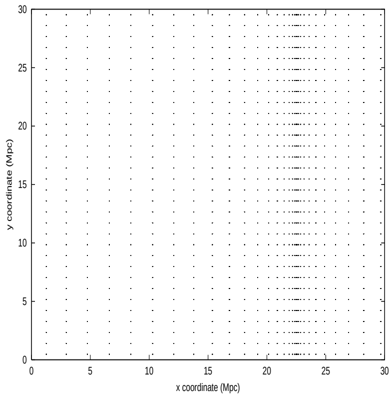
(c) a=0.6088



(d) a=0.7504



(e) a=0.8796



(f) a=1.0000

Figure 8.3: Evolution of a Zel'dovich plane wave solution using Adaptive MLAPM. Projections onto the x-y plane for various scale factors

The particle positions and velocities were compared with the analytic expressions, equations (8.5) and (8.6). The RMS position and velocity errors were calculated for each of the five scale factor values illustrated in figures 8.2 and 8.3. The RMS errors are defined by

$$\Delta x_{\rm rms} = \left[\sum_i (\mathbf{x}_i - \mathbf{x}_i^Z)^2 / \sum_i (\mathbf{x}_i^Z - \mathbf{q}_i)^2\right]^{1/2}, \tag{8.7}$$

$$\Delta v_{\rm rms} = \left[\sum_i (\mathbf{v}_i - \mathbf{v}_i^Z)^2 / \sum_i (\mathbf{v}_i^Z)^2\right]^{1/2}. \tag{8.8}$$

The results for the four simulations can be found in the table 8.1. The adaptive case clearly shows smaller errors.

	Non-adaptive Grid		Adaptive Grid	
a	$\Delta x_{\rm rms}$	$\Delta v_{\rm rms}$	$\Delta x_{\rm rms}$	$\Delta v_{\rm rms}$
$k = 1.0$				
0.448140	0.007569	0.027647	0.007569	0.027647
0.608783	0.013890	0.035741	0.013805	0.034148
0.750370	0.018655	0.043754	0.017728	0.034469
0.879635	0.022981	0.055286	0.020303	0.036184
1.000000	0.027655	0.074143	0.022014	0.035262
$k = 2.0$				
0.448140	0.014877	0.056877	0.014877	0.056877
0.608783	0.029117	0.080881	0.029117	0.080881
0.750370	0.040788	0.102420	0.038742	0.081893
0.879635	0.051375	0.125553	0.045383	0.086722
1.000000	0.061703	0.151883	0.050458	0.091224

Table 8.1: RMS position and velocity errors for Zel'dovich plane wave test with a single wave.

8.3 Test Simulations

To test MLAPM in a realistic setting four simulations were performed using some initial conditions from a COBE normalized standard CDM model. The results of these three simulations are compared with results obtained from the publicly available code by Couchman (1991)[2]. The parameters of the simulations are listed in table 8.2.

Parameter	Value
Ω_0	1.0
Λ_0	0.0
Starting red shift z	25
σ_8	1.18
h	0.5
Box length	$30h^{-1}Mpc$
Timestep size	0.001985[3]
Timesteps	500
No. of particles	32^3

Table 8.2: Parameters of the simulation used to test MLAPM

[2]This data set and the results from the Couchman code were kindly provided by Dr. Alexander Knebe.

The four MLAPM simulations[4] were set up as follows; the first a non-adaptive simulation using a 32^3 grid, the second an adaptive simulation starting on a 32^3 grid with adaptive sub-grids allowed down to a "virtual" resolution of 128^3 the third simulation was a non-adaptive simulation with a 128^3 grid, and the fourth an adaptive simulation starting on a 32^3 grid with adaptive sub-grids allowed down to 256^3[5]. The Couchman code was been run with a large softening length, giving a force resolution of 4 grid cells on a 32^3 grid. In this mode the Couchman code behaves as if it were a PM code rather than a AP^3M code.

Comparing different simulations is not a simple task. The naive thing to do would be to compare the final positions and velocities of each particle in each simulation. However, we should not expect these to be the same in different types of code, or in the same code with a different choice of simulation parameters. We have a dynamical system and differences in the trajectories of particles will accumulate over time. So to perform a comparison of different simulations, I have chosen a several different methods. The first comparison method is a simple visual comparison, where final projections of the final particle positions are compared. A second comparison is made using RMS differences in final particle positions and velocities. The distribution of particle separations is also presented. The power-spectra from the end of the simulations are compared, as these give a good measure of the overall evolution of all the particles. The density cross-correlation coefficient is given for all the simulations, this gives a measure of how closely the density fields match. Production of halos, found using a friends-of-friends algorithm, is considered. Finally some information is provided about grid placement for the adaptive simulation and some code performance measures.

8.3.1 Visual Comparison

The projection of the final particle positions onto the xy-plane can be found in figure 8.4. This figure contains the projection of the first three MLAPM simulations and the Couchman simulation.

Consider the top two sub-figures in figure 8.4, containing the projections of the 32^3 non-adaptive MLAPM simulation and the Couchman simulation. The gross features of both simulations are very similar with the large particle concentrations clearly in the same positions. In particular the large particle concentration is visible in both simulations as is its companion immediately below it. The two smaller companion objects to the right are also present in both simulations. Also clearly present are the five objects in the upper left quadrant. Several of the smaller objects in the MLAPM simulation are missing from the Couchman simulation or are smeared out. This is consistent with the lower force resolution of the Couchman simulation. Specifically, the dense knot of particles near the coordinates $(18, 25)$ is absent from the Couchman simulation.

Next consider the bottom two simulations in figure 8.4, the adaptive MLAPM simulation and the non-adaptive 128^3 MLAPM simulation. Again the gross features are very similar and in the same locations as in the previous two simulations. In both these higher resolution simulations, much more substructure is present. For example, five small objects are now visible just to the left of the large particle concentration, this region was not resolved in the earlier two simulations. There are some small differences between the two simulations, for example, three small knots of particles are present in the 128^3 non-adaptive simulation at approximate coordinates $(18, 4)$. At the same location in the adaptive simulation, small objects are visible, but are not as widely separated. In general the distribution of particles in the *void* regions in the adaptive simulation is smoother than in the non-adaptive one. Of course these regions have only low-resolution grids in the adaptive simulation.

[4]The MLAPM simulations were performed with the a parameter of §4.4.1 set equal to 0.1 rather than 0.025.

[5]The fourth MLAPM simulation was not included in all parts of the analysis.