Using Interval Analysis to Bound Numerical Errors in Scientific Computing

Josh Milthorpe

A subthesis submitted in partial fulfilment of the degree of
Bachelor of Information Technology (Honours) at
The Department of Computer Science
Australian National University

October 2005

© Josh Milthorpe

Typeset in Palatino by TeX and LaTeX 2_ε.

Except where otherwise indicated, this thesis is my own original work.

Josh Milthorpe
28 October 2005

To Amy, who worked so I could study
To Bridie, who had studies of her own
To Felix, who slept

Acknowledgements

Thanks go first of all to my supervisor, Alistair Rendell, for many hours of stimulating discussion and criticism. I have met few people who can conceive such great ideas; I have met even fewer who are able to explain them so well.

Thanks also to Bill Clarke, Rui Yang, Peter Strazdins, Andrew Over, Hsien-Jin Wong, Joseph Antony and Trystan Upstill, who provided advice, ideas and encouragement along the way.

Thanks to all my honours colleages for providing the thought environment that I came here looking for: Simon, Adhiti, Rob, Roy, Martin, Puthick and Jin in 2004, and Lei, Nick, Daniel, Song-Yang, Warren, Tom and Eleanor in 2005. I'm sure you will do brilliantly in whatever enterprise you undertake; I wish you all the best.

Thanks to my family: Mum, Dad, Nae, Jules and Steve for constant moral support over the two years and the all-important babysitting that allowed me to palm off the bairns so I could concentrate on my 'other baby'.

Thanks to Vickie and David, who also provided babysitting, and some very fine meals. There are few places I feel more at home than in your back garden, sitting back in the afternoon sun, with a glass of sangria in hand.

Thanks to the friends who were always there as a sanity-check during this occasionally trying time: Corri, Nick, Jake, Nadia, Ollie and Reem. I am sorry to have forgotten you so often in the last two years; but you've never forgotten me, Amy or the kids.

Thanks to Bridie, who cried so many times when I had to leave her for Uni or work. I wish I could spend every day with you, honey.

Thanks to Felix, who has lived up to his name and is the happiest son any father could wish for. You were the most welcome disruption ever to hit an exam period.

Finally, and most importantly, thanks to Amy, who has given me so much of what I truly value in my life. You have been patient and supportive as I've followed my dreams; I hope I can support you as well, as you follow yours.

Abstract

In scientific computing, the approximation of continuous physical phenomena by floating-point numbers gives rise to rounding error. The behaviour of rounding errors is difficult to predict, and most scientific applications ignore it. For applications in which the accuracy of the result is critical, this is not an acceptable choice.

Interval analysis is an alternative to conventional floating-point computation that offers guaranteed error bounds. Despite this advantage, interval methods have rarely been applied in high performance scientific computing. In part, this is because of the additional cost associated with performing interval operations over the corresponding floating-point operations. Another issue is the lack of example applications of interval analysis; many scientific users simply do not know that an alternative to floating-point computation exists.

This thesis develops and demonstrates techniques by which interval analysis can be feasibly applied to scientific computing. Methods are shown by which the performance of interval codes may be significantly improved on the UltraSPARC platform: hand-optimised codes for vector functions are developed that are around 60% faster than the Sun interval implementation, and an alternative method of interval multiplication is developed that is around 30% faster than the method used by other implementations.

The benefits of interval analysis are demonstrated through application to an example scientific problem: the evaluation of electrostatic potentials. Alternative methods of evaluation are analysed: two new accurate methods of summation are proposed and proven to reduce rounding error in this application. This demonstration includes what is believed to be the first interval implementation of the fast multipole method. The implementation is used to explore the balance between rounding and truncation errors in the method, which has previously been ignored. The results suggest that rounding error may affect the accuracy that can be practically achieved using the method. The results also provide guidance in choosing parameters to minimise error.

Contents

Acknowledgements vii

Abstract ix

1 Introduction 1
- 1.1 Purpose . . . 2
- 1.2 Contribution . . . 2
- 1.3 Organisation . . . 2

2 Background 5
- 2.1 Sources of error . . . 5
- 2.2 Numerical analysis . . . 6
- 2.3 Interval analysis . . . 6
  - 2.3.1 Development of interval analysis . . . 7
  - 2.3.2 Interval arithmetic . . . 8
  - 2.3.3 Interval functions . . . 9
  - 2.3.4 Drawbacks of interval methods . . . 9
  - 2.3.5 Applications . . . 10
    - 2.3.5.1 Global Optimisation . . . 10
    - 2.3.5.2 Other applications in science and engineering . . . 11
- 2.4 Alternative approaches to reducing floating-point error . . . 11
  - 2.4.1 Exact operations . . . 11
  - 2.4.2 Affine arithmetic . . . 12
  - 2.4.3 Probabilistic error analyses . . . 13

3 Interval arithmetic performance 15
- 3.1 Overview . . . 15
- 3.2 Benchmark platforms . . . 15
- 3.3 Benchmarking method . . . 16
- 3.4 Basic interval operations . . . 17
  - 3.4.1 Benchmarks F1/I1 and F2/I2: “Basic” . . . 17
  - 3.4.2 Cache effects . . . 18
  - 3.4.3 Reasons for poor interval performance . . . 19
    - 3.4.3.1 Inherent cost of interval addition . . . 19
    - 3.4.3.2 Inherent cost of interval multiplication . . . 20
    - 3.4.3.3 Procedure calls and loop unrolling . . . 20
    - 3.4.3.4 Rounding mode switching . . . 22

3.4.3.5 Summary of reasons for poor performance 23
3.4.4 Benchmarks to investigate reasons for poor performance 23
3.4.4.1 Benchmarks F3 and F4 "No unrolling" 23
3.4.4.2 Benchmarks F5 and F6 "Procedure call" 24
3.4.4.3 Benchmarks F7 and F8 "Switching" 24
3.4.4.4 Summary of investigation of poor performance 24
3.4.5 Intrinsic array functions 25
3.4.5.1 Benchmarks F9/I3 and F10/I4: "Intrinsics" 25
3.4.6 Handwritten interval operations 26
3.4.6.1 Benchmarks I5 and I6: "Handwritten sums" 26
3.4.6.2 Benchmarks I7 and I8: "Handwritten products" 28
3.4.7 Proposal: improved hardware support for interval multiplication 32
3.4.8 Comparison of interval performance: UltraSPARC vs. IA-32 . . . 33
3.4.8.1 Benchmarks I9, I10 and I11: "Handwritten interval", C++ on *alcatraz* 33
3.4.8.2 Benchmarks I12, I13 and I14: "Handwritten interval", C++ on *partch* 34
3.5 Compound operations 35
3.5.1 Dot product and AXPY 35
3.5.1.1 Benchmarks F11/I15 and F12/I16: "BLAS1 on *alcatraz*" 36
3.5.1.2 Interval instruction mix 36
3.5.1.3 Benchmarks I17 and I18: "Handwritten BLAS1 on *alcatraz*" 37
3.5.2 Matrix multiplication 38
3.5.2.1 Benchmarks F13/I19, I20 and I21: "BLAS3 on *alcatraz*" . 38
3.6 Conclusions 39

4 Example application: evaluation of electrostatic potentials 41
4.1 Problem description 41
4.2 Direct evaluation 43
4.2.1 Standard summation 43
4.2.2 Ordered summation 44
4.2.3 Partially-ordered summation 45
4.2.4 Summation with accumulators 46
4.2.5 Compensated summation 48
4.2.5.1 Proof 48
4.2.5.2 Implementation 50
4.2.6 Results 51
4.2.7 Qualifications on the results 53
4.2.8 Benefits of interval analysis 54
4.3 Fast multipole method 55
4.3.1 Overview of fast multipole method 55
4.3.1.1 Motivation 55
4.3.1.2 Hierarchical space 56

4.3.1.3 Multipole expansions 57
4.3.1.4 $\mathcal{O}(n \log n)$ method for potential evaluation 58
4.3.1.5 Translation of multipole expansions 59
4.3.1.6 Local expansions 59
4.3.1.7 $\mathcal{O}(n)$ method for potential evaluation 60
4.3.2 Interval FMM implementation 61
4.3.2.1 Results 62
4.3.3 Rounding and truncation error in the FMM 62
4.3.4 Qualifications on these results 64
4.3.5 Performance of the interval fast multipole method 65
4.4 Conclusions 65

5 Discussion **67**
5.1 Performance of interval computation 67
5.2 Usefulness of interval results 68

6 Conclusion **71**
6.1 Contributions 71
6.2 Further work 72
6.2.1 Interval tools 72
6.2.2 Fast multipole method 72
6.3 Conclusion 73

A Software tools for interval computation **75**
A.1 Sun compilers 75
A.2 filib 75
A.3 Boost interval arithmetic library 75
A.4 Interval arithmetic in Maple 76

B Mathematics for the fast multipole method **77**
B.1 Potential functions 77
B.2 Multipole and local expansions 78
B.3 Translation of multipole expansion 78
B.4 Transformation of multipole to local expansion 79
B.5 Translation of local expansion 79
B.6 Associated Legendre functions 80
B.7 Truncation error bounds for the fast multipole method 80
B.7.1 Error in a truncated multipole expansion 80
B.7.2 Error due to transformation of multipole to local expansion . . . 80

C Source code for the interval fast multipole method **81**
C.1 Source file: fmmgrid.F95 81
C.2 Source file: fmm.F95 82
C.3 Source file: inttypes.F95 95
C.4 Source file: intfuncs.F95 96

C.5 Source file: randomizer.F95 100

Bibliography **103**

Chapter 1

Introduction

> The computing industry has created *Moore's fear*: mission critical applications run on computers in which Murphy is free to roam with impunity.
>
> **G. William Walster**
> Keynote Address, SCAN 2000

Since their beginnings, computers have been used to advance the project of science in understanding the world. As computing power has increased, scientists have used ever more sophisticated models to analyse and predict physical phenomena. In fact, there are now whole branches of science in which computational simulations are either the most commonly used predictive method (for example, meteorology, seismology and fluid mechanics) or are fundamental to the science itself (as in chaos theory). After Theory and Experiment, Computation has been described as 'the third major domain in the Sciences'. [7].

It is alarming, therefore, to realise how little is known regarding the accuracy of computed results. Almost all scientific computation today uses the floating-point paradigm, in which the concept of a rounding error is inherent. When floating-point hardware was introduced in 1955 it was opposed by some numerical analysts on the grounds that important accuracy information would be lost [57]. While some improvements have been made such as increased precision, guard bits and sticky bits, the underlying problem with the floating-point paradigm remains: most computable operations will result in a rounding error. Despite this, the literature on scientific applications usually ignores computational error, either assuming that the results are accurate, or 'hoping for the best'.

However, recent developments in hardware and software have led to an increased interest in 'reliable computing': methods of computation that provide provable error bounds. The most promising of these is *interval analysis*, in which all computation is performed on intervals that can be proven to contain the correct results. Despite a lack of standards in this area, and the fact that interval computation suffers a performance penalty compared to standard floating-point computation, the time is now ripe for wider adoption of these methods.

Floating-point computation is now used in safety critical applications, and computer simulations are increasingly used as a substitute for experimental results. Spec-

tacular failures of floating-point computation have already occurred, in one case well-known case, leading to the death of 28 American soldiers [1]. It is incumbent on computer scientists to develop the tools for accurate computation. To do otherwise is to risk further disasters and a loss of confidence in our profession.

1.1 Purpose

The aim of this thesis is to develop and demonstrate techniques by which interval methods can be feasibly applied to scientific computing. To progress this aim, two subgoals are addressed: to analyse and improve upon the performance of tools for interval computation; and to demonstrate the utility of interval methods through error analysis of problems in scientific computing.

1.2 Contribution

This thesis contributes both to the tools available for interval analysis, and to its applications. Changes are proposed to the implementation of interval multiplication that would improve the speed of this operation on UltraSPARC and similar in-order architectures. An interval 'compensated sum' is proposed that provides narrow interval width for small additional cost. Interval analysis is applied to the calculation of electrostatic potentials through pairwise and fast multipole methods, yielding information on rounding errors that can guide the use of these methods.

1.3 Organisation

Chapter 2 gives an overview of the domain of this thesis. The problem of errors in scientific computing is discussed, and previous approaches to error analysis are reviewed. The techniques of interval analysis are introduced and a brief survey of previous applications is given. Particular attention is given to use of intervals in scientific and engineering applications.

Chapter 3 analyses the performance of selected tools for interval computation, in particular the compiler support for intervals in the SunStudio suite [4; 5]. Techniques are presented to improve the performance of interval arithmetic on target architectures.

Chapter 4 applies interval analysis to the pairwise computation of electrostatic potentials in a system of charged particles. Two methods of computation are considered: direct evaluation and the fast multipole method. The behaviour of different algorithmic orderings is analysed for direct evaluation, with regard to rounding error. Two new methods for accurate summation are introduced: an accumulator method and a compensated interval sum. In § 4.3, an interval implementation of a fast multipole method is constructed, and the behaviour of the method for various parameters is analysed with regard to rounding error. The performance of the method is evaluated for both floating-point and interval implementations.

Chapter 5 discusses the benefits and drawbacks of interval methods, given the results presented in Chapter 3 and Chapter 4.

Chapter 6 contains the conclusion. The contribution of this thesis is reviewed and further work is proposed.

Chapter 2

Background

There are very few problems, really,
which can be solved exactly by analysis.

Richard Feynman
Lecture on *Conservation of Momentum*
Caltech, 27 October 1961

Much of science is concerned with physical phenomena that are continuous, such as time, distance or temperature. In contrast, digital computing is limited to discrete representations. Traditionally, this divide has been bridged through the use of floating-point arithmetic, with all numbers approximated to a specified precision using a finite set of machine-representable numbers. Inherent in this approximation is the concept of a rounding error. The behaviour of rounding errors in a given computation can be difficult to predict in advance; they may cancel out or compound, depending on both the algorithm and the particular input data.

Currently, almost all scientific codes ignore rounding errors, instead using double-precision arithmetic and hoping for the best. However, today's large parallel computers are capable of performing in excess of 10^{12} floating-point operations per second (over 1 Teraflop). Each of these operations is a potential source of rounding error, so even when using double-precision arithmetic which has rounding errors of the order of 10^{-15}, it is easy to see that errors may quickly compound to be as large as the computed quantities themselves.

This chapter will review various approaches to rounding error. Traditional approaches are contrasted with interval analysis, which aims to compute error bounds at the same time as the quantities of interest. A brief explanation is given of interval methods, and previous applications are reviewed. Finally, alternative approaches are considered.

2.1 Sources of error

Errors in computation stem from three main sources: uncertainty in data, truncation and rounding [30]. Each will be discussed in turn.

Uncertainty in input data is well understood by scientists because it has a clear physical basis. For example, the value of the gravitational constant G can only be determined through experiment; the most precise measurements of G are greatly affected by experimental conditions and so it is usually reported with wide error bounds e.g. 6.67 ± 0.2. [22] Scientists and engineers are trained to represent quantities with their associated uncertainties, and to carry these uncertainties through the steps of a calculation.

Truncation errors are introduced by algorithms that approximate values to an arbitrary level of accuracy. For example, the value of e is given by the infinite series:

$$e = 1 + 1/1! + 1/2! + 1/3! + \ldots$$

This series may be truncated at a chosen number of terms to yield an approximate result within a known error bound. Truncation errors are sometimes bounded in advance using known properties of the algorithm.

Rounding errors are an artefact of finite-precision arithmetic, and can occur when a real number (of which there are infinitely many) is approximated by one of a finite set of machine-representable numbers. The behaviour of rounding errors in a given computation can be difficult to predict in advance; they may cancel out or compound, depending on both the algorithm and the particular input data. The focus of this thesis is on methods to bound rounding errors, with some discussion of other error types.

2.2 Numerical analysis

The traditional way to determine the accuracy of computed results is through numerical analysis. Algorithms are analysed given known properties of truncation and rounding errors, and an error model is constructed to predict the behaviour of these errors. An excellent text on this subject is Higham [30], which gives a comprehensive analysis of numerical linear algebra.

The main reason that numerical analysis is not performed on most scientific codes is its inherent difficulty. Analysing a code requires a detailed knowledge of the floating-point system that many users do not have. More importantly, it is not an automatic process and may take significant human effort. This had led many researchers to look for automatic methods of calculating error bounds.

2.3 Interval analysis

In 1959, Moore [45] proposed interval arithmetic as an extension to ordinary arithmetic on real numbers. An interval is a continuum of real numbers, defined by a lower bound and an upper bound. By substituting intervals for floating-point numbers in automated computation, an interval result can always be found to bound the correct real result.

Implemented correctly, interval computation can be used to bound errors from all sources, including input uncertainty, truncation and rounding errors. This has led interval researchers to label their field *Reliable Computing*. In many fields verified results, such as those produced by interval computation, are obviously valuable. It is no surprise that some researchers [25] have questioned why interval techniques are not more widely used.

However, the application of interval techniques is not always straightforward; in particular, it often requires algebraic manipulation of functions to find interval extensions that produce *sharp*[1] interval bounds. A further barrier to widespread adoption of interval methods is the potentially large performance penalty of interval computation; this barrier will be discussed further in Chapter 3.

2.3.1 Development of interval analysis

Moore was not the first to consider using intervals to represent uncertain values. Warmus [58] and Sunaga [55] both discussed the mathematics of intervals. However, Moore's treatment of the subject was most comprehensive.

Moore's first book [46] sets out many of the basic theorems on which the field is founded. Moore applies interval methods to common "real number" problems so as to obtain provable bounds on all errors. Key areas of discussion include:

- rational functions

> If $f(x_1, x_2, \ldots, x_n)$ is [a] real rational expression in which each variable x_i occurs only once and only to the first power ($x_1^2 = x_1 \cdot x_1$ is taken as two occurrences), then the corresponding interval expression $F(X_1, X_2, \ldots, X_n)$ will compute the actual range of values of f for $x_i \in X_i \ldots$ (p11)

- matrix computations

> . . . the product of two interval matrices using interval arithmetic is again an interval matrix consisting of interval elements each of which is exactly the range of values of the corresponding element of the product of a pair of real matrices whose elements are chosen independently from the corresponding elements of the interval matrices. (p26)

- techniques for improving the error bounds on rational functions over what can be achieved using straightforward interval-arithmetic evaluation, such as: intersection, variable cancellation, use of subdistributivity, the *centred form*, the *mean-value form*, and the use of root-finding methods to determine extreme values. (ch. 6–7)

- integral functions (ch. 8–9)

[1] An interval is said to be *sharp* if it is as narrow as possible.

- interval methods for systems of Ordinary Differential Equations (ODEs) (ch. 10–13)

2.3.2 Interval arithmetic

An interval may be represented as a pair of endpoints. There are alternative representations such as the *midpoint-radius* form; however these will not be used in this thesis.

For a computer with hardware support for floating-point arithmetic, the most natural representation for an interval is a pair of floating-point numbers (the lower bound $\underline{X}$ and upper bound $\overline{X}$). The ubiquitous IEEE 754 standard[33] for floating-point arithmetic defines two rounding modes to support interval arithmetic: upwards (round towards $+\infty$) and downwards (round towards $-\infty$).

For two intervals X and Y, the binary arithmetic operations are defined by

$$\begin{aligned} X \circ Y &\equiv \{x \circ y : x \in X; y \in Y\} \\ \circ &\in \{+, -, \times, /\} \end{aligned}$$

This definition can be made more specific for each operation:

$$X + Y = [\underline{X} + \underline{Y}, \overline{X} + \overline{Y}]$$

$$X - Y = [\underline{X} - \overline{Y}, \overline{X} - \underline{Y}]$$

$$\begin{aligned} X \times Y &= [a, b] \\ \text{where} & \\ a &= \min(\underline{X} \times \underline{Y}, \underline{X} \times \overline{Y}, \overline{X} \times \underline{Y}, \overline{X} \times \overline{Y}) \\ b &= \max(\underline{X} \times \underline{Y}, \underline{X} \times \overline{Y}, \overline{X} \times \underline{Y}, \overline{X} \times \overline{Y}) \end{aligned}$$

$$\begin{aligned} 1/Y &= \left[1/\overline{Y}, 1/\underline{Y}\right] \\ X/Y &= X \times 1/Y \end{aligned}$$

The interval power operator is

$$X^n = \begin{cases} [1, 1] & \text{if } n = 0 \\ \left[\underline{X}^n, \overline{X}^n\right] & \text{if } \underline{X} \geq 0 \text{ or } \underline{X} \leq 0 \leq \overline{X} \text{ and } n \text{ is odd} \\ \left[\overline{X}^n, \underline{X}^n\right] & \text{if } \overline{X} \leq 0 \\ \left[0, \max(\underline{X}^n, \overline{X}^n)\right] & \text{if } \underline{X} \leq 0 \leq \overline{X} \text{ and } n \text{ is even} \end{cases}$$

Where interval endpoints are represented by floating-point numbers, endpoint results that cannot be represented exactly must be rounded outwards to a representable

number. For the example of addition:

$$X + Y = [\lfloor \underline{X} + \underline{Y} \rfloor, \lceil \overline{X} + \overline{Y} \rceil]$$

and for multiplication:

$$\begin{aligned} X \times Y &= [a, b] \\ a &= \min(\lfloor \underline{X} \times \underline{Y} \rfloor, \lfloor \underline{X} \times \overline{Y} \rfloor, \lfloor \overline{X} \times \underline{Y} \rfloor, \lfloor \overline{X} \times \overline{Y} \rfloor) \\ b &= \max(\lceil \underline{X} \times \underline{Y} \rceil, \lceil \underline{X} \times \overline{Y} \rceil, \lceil \overline{X} \times \underline{Y} \rceil, \lceil \overline{X} \times \overline{Y} \rceil) \end{aligned}$$

Here the floor and ceiling symbols indicate that inexact results are to be rounded downward and upward respectively to a representable number.

The basic interval arithmetic does not define the result for division by an interval containing zero. For some applications, particularly in optimisation, it is necessary to use an *extended interval arithmetic* in which this result is defined. [27]

2.3.3 Interval functions

In the same way that the interval arithmetic operations were defined in terms of real arithmetic operations, interval functions can be defined that correspond to real functions. Given a real-valued function f, it is possible to create an *interval extension* F that is an interval valued function of interval variables, which reduces to f when all the arguments are real numbers (intervals of zero width). An *inclusion monotonic interval extension* of f accepts interval arguments $X_1, X_2, \ldots, X_n$ and has an interval result which contains all the possible values of f if the values of its arguments $x_1, x_2, \ldots, x_n$ were allowed to range independently in the intervals $X_1, X_2, \ldots, X_n$. That is

$$F(X_1, X_2, \cdots, X_n) \equiv \{f(x_1, x_2, \ldots, x_n) : x_1 \in X_1; x_2 \in X_2; \ldots; x_n \in X_n\}$$

Thus it is possible to define an interval sine, tangent, square root, exponential, and so forth.

It is sometimes no harder numerically to evaluate an interval extension than to evaluate the corresponding real function. For example, the trigonometric functions are usually evaluated by a series which converges around the correct value. The interval result is simply the last two calculated values for such a series.

2.3.4 Drawbacks of interval methods

The most commonly stated reason for the failure of interval methods is the *dependency problem* (Hansen [27] p11), in which the repeated occurrence of the same variable in interval expressions leads to overly wide interval results. Because some terms in the expression are dependent upon others, the application of straightforward (or "naïve") interval arithmetic means that the uncertainty in some variables becomes multiplied in the result.

This problem is documented in application of interval methods by Birdie and

Surana [11] and Makino and Berz [40]. Methods of detecting or reducing dependencies are discussed by Hansen [27; 28] and Makino and Berz [40]. However, Moore [46] comments on the difficulty of avoiding dependency:

> In general, it is not possible to rewrite a rational expression in which a number of real variables each occur several times in such a way that the new expression contains only one occurrence of each variable. (p28)

Neumaier [48] mentions two other common pitfalls of interval calculations: *wrapping* and *cancellation*. Wrapping occurs in long sequences of nested operations, magnifying rounding errors to give interval results so wide as to be meaningless. Cancellation occurs where the magnitude of the result of addition or subtraction is much smaller than the arguments. In floating-point computation, this results in a rapid loss of significant digits, that may go undetected. In interval computation this loss of significant digits is easily detected as it quickly expands the interval widths, which may grow to be wider than the magnitude of the probable values.

2.3.5 Applications

2.3.5.1 Global Optimisation

Hansen [27] applies interval analysis to the global optimisation problem. He stresses the use of interval analysis not just for error bounding, but for finding solutions to hitherto unsolvable real optimisation problems.

> ...our primary concern is to find zeros of functions or to minimize functions which originate in a noninterval context. We merely use interval methods to solve the problems involved. (p22)

As an example, Hansen uses the interval Newton method to solve questions of existence and uniqueness that were considered "unsolvable" by researchers in 1983 (p68).

Three years after Hansen's book was published, Hass, Hutchings and Schafly [29] announced what is arguably the most famous proof attributable to interval analysis, for the so-called *Double Bubble conjecture*. They solved what had been an open question in geometry since 1873: what is the least area surface enclosing two equal volumes? Interval techniques were used to prove that the *double bubble* (two spheres intersecting in a circle at an angle of 120°) is globally optimal.

An even older conjecture also appears to have been solved using interval analysis. In 1998, Tom Hales [26] announced a proof of the *Kepler conjecture*. In 1611, Kepler declared that the face-centred cubic arrangement achieves the highest possible density for a packing of equal spheres. While intuitively true, this statement has defied geometric proof since that time. Hales used interval analysis in a 'proof by exhaustion' which eliminates other likely arrangements as being less optimal. However his proof is difficult to check; the current estimate is that it will take 20 years to confirm its validity.

Stadtherr, Schnepper and Brennecke [52] apply interval methods to phase stability analysis in chemical process modelling. As this is basically a global optimisation problem, they build on the work of Hansen [27] and find that interval analysis provides efficient and completely reliable results.

Lin and Stadtherr [39] apply interval methods to several global optimisation problems in chemical engineering. This includes finding all stationary points of potential energy surfaces for selected triatomic molecules.

Neumaier [47] lists some "Grand Challenges", solutions for which (he claims) will change the way interval analysis is seen outside the research community. Global optimisation is singled out as

> ... the only area of wide practical applicability where ... intervals are likely to have a major impact in the near future.

2.3.5.2 Other applications in science and engineering

Makino and Berz [40] apply interval Taylor model methods to the calculation of Coulomb forces in many body dynamical systems.

Hoefkens, Makino and Berz [31] apply high-order Taylor model methods to obtain a validated orbit integration of a near earth asteroid.

Fefferman and Seco [17; 18] use interval arithmetic to establish an essential inequality in the proof of a precise asymptotic formula for the ground-state energy of a non-relativistic atom.

Watanabe et al. [59] use intervals to bound rounding errors in the computation of non-trivial solutions to the heat convection problem.

Holzmann, Lang and Schütt [32] use interval arithmetic to bound numerical error and several input errors in a physical experiment to calculate the value of the gravitational constant G. They calculate to what extent certain measurement uncertainties must be reduced in order to calculate G to a given desired accuracy.

Birdie and Surana [11] use intervals to bound uncertainty in input data in geohydrological problems.

2.4 Alternative approaches to reducing floating-point error

2.4.1 Exact operations

Bohlender et al. [12] argue that the IEEE 754 approach, in which accurate operations are defined on scalar floating-point values, should be extended to vectors. They propose an *accurate dot product* operation that accepts two vectors and returns a dot product that is accurate to within working precision. Key to this proposal is the idea of an *accumulator object* that stores floating-point values of very high precision; this is used in the accurate dot product to store the intermediate results, which are rounded to working precision at the end of the operation. They do not specify how large this accumulator would need to be to ensure that neither truncation nor overflow occur; their only comment is:

> Dependent on the processing power of the machine, a small number of extra digits is sufficient to avoid the occurrence of [overflow] in the lifetime of the computer. (p. 96)

It is easy to make a rough estimate of how many digits would be needed for this accumulator object. The first point to note is that, for a working precision of n bits in the significand, the accumulator need only store $\log_2 m$ additional bits to ensure an accurate result without overflow, where m is the total number of additions that may be performed in the lifetime of the computer. If the exponents of two numbers differ by more than $n + \log_2 m$, the smaller number may be safely truncated, because it is not possible in the lifetime of the computer to add enough of these smaller numbers to affect the final result.

The fastest single processors today perform on the order of 10^9 additions per second; over a lifetime of ten years such a processor could perform 10^{17} additions. An accumulator for such a processor therefore requires an additional $\log_2(10^{17}) = 57$ bits to ensure an accurate dot product with no overflow. For IEEE 754, a double-precision significand is 53 bits; therefore the total length of accumulator significand is around 110 bits to support an exact dot product for double-precision numbers; allowing for exponent bits, the accumulator object requires less than twice as much storage as a normal 64-bit register.

This proposal has great merit. Exact vector arithmetic could be incorporated into programming language specifications, for example, in the definition of the Fortran intrinsics `SUM`, `DOT_PRODUCT` and `MATMUL`. This would remove one of the most common sources of error in scientific and engineering codes, and greatly simplify error analysis of algorithms that use these operations. [2] The drawback is that the implementation would require significant changes to existing hardware; and manufacturers are usually slow to adopt changes that do not leverage their existing investments in design.

2.4.2 Affine arithmetic

Affine arithmetic is not really an alternative to interval arithmetic, but rather a generalisation of it. [9] Instead of using a single error bound to represent all sources of error, separate variables are maintained for each independent source of error. Thus it is possible to track error due to uncertainty in inputs and error due to rounding throughout a computation.

The advantage to treating errors in this way is that it is sometimes possible to cancel part of the error at a later step of the computation, thereby yielding a final result of narrower width than is possible using standard interval arithmetic.

The disadvantage is that affine arithmetic tends to be much slower in computation; and it is not as easy to understand, as there are more variables to track, and the operations are more complex. For these reasons there are very few researchers or applications using affine arithmetic.

[2] However, exact vector operations are certainly not a panacea. Even sequences of operations that are individually exact can generate significant rounding errors.

2.4.3 Probabilistic error analyses

Interval analysis provides *verified* error bounds on a quantity of interest. An alternative approach is to calculate *probabilistic* bounds on a quantity. An example is *stochastic arithmetic* [56], which performs arithmetic on Gaussian distributions. Quantities of interest are represented as a mean and standard deviation:

$$X = (m, \sigma)$$

which allows the calculation of a confidence interval:

$$P(r \in [m - \lambda_\beta \sigma, m + \lambda_\beta \sigma]) = 1 - \beta$$

This approach is well suited to calculations on quantities that closely follow a Gaussian distribution, particularly where the maximum or minimum is not known. Some researchers [8] have suggested that probabilistic schemes could also be used to account for rounding error, justifying the method with statements such as

> ...Round-off errors can be considered as uniformly distributed random variables for mantissa $p > 10$. [19]

In practice, probabilistic schemes are unlikely to be suitable for analysis of rounding error. While rounding error often behaves like a random variable, this is not always the case, and there are no hard and fast rules to judge when the assumptions underlying the schemes will fail. Given this unpredictability, and the potentially serious consequences of invalid assumptions, verified methods like interval analysis are to be greatly preferred in accounting for rounding errors.

Chapter 3

Interval arithmetic performance

Very few cartoons are broadcast live.
It's a terrible strain on the animators' wrists.

The Simpsons
Itchy & Scratchy & Poochie episode

3.1 Overview

Although few scientific codes are "broadcast live" (run in real-time), good performance is still a key requirement. Researchers need their codes to scale to ever more complex simulations and larger problem sizes. To prove the feasibility of interval methods for HPSC applications, it is necessary to show that, for typical problems, an interval solution can be computed without a prohibitive increase in the cost of the computation over a comparable floating-point solution.

It is desirable to have some performance data to compare commonly used operations implemented in floating-point and interval arithmetic. To this end, benchmarks were constructed for these operations and the behaviour of the benchmarks analysed to give an indication of the likely performance of interval arithmetic on "real world" applications.

The first operations chosen were addition and multiplication. They were chosen because they are the two simplest arithmetic operations, and because they form the basis of many more complex operations and functions. Further benchmarks were performed on compound operations: inner product, AXPY and general matrix multiplication.

3.2 Benchmark platforms

Two machines were used to benchmark the performance of interval and floating-point codes. The first was an UltraSPARC-based system running the Solaris operating system (*alcatraz*). The second was an IA-32 based system running GNU/Linux (*partch*). The details of each platform are given in Table 3.1.

	alcatraz	**partch**
CPU	900MHz UltraSPARC-IIICu	2.6GHz Intel Pentium 4
Level 1 Cache	64KB	8KB
Level 2 Cache	8MB	512 KB
Operating System	Solaris 5.10	GNU / Linux
		2.6.8-2-686-smp
Compilers	Sun Fortran 95 8.1 (Sun Studio 10)	gcc 3.4.4
	Sun C++ 5.7 (Sun Studio 10)	
	gcc 3.4.4	

Table 3.1: Specifications of platforms used to benchmark interval arithmetic performance

The two architectures have some notable differences: the IA-32 is a CISC machine with out-of-order execution whereas the UltraSPARC is an in-order RISC machine. These platforms were chosen to investigate whether these architectural differences have an effect on interval performance. The investigation focused mainly on the UltraSPARC platform.

3.3 Benchmarking method

Each of the benchmarks was run in a standard "harness" which executed a kernel procedure containing the operation that was being tested. The kernel procedures operate on vectors or matrices, the size of which is determined at runtime from command line arguments.

The benchmarks were written in Fortran and compiled using the Sun Fortran 95 compiler with full optimization. Except where noted, the compiler options used on *alcatraz* were:

```
-fast -xarch=v9b [-xia]
```

The `-xia` flag indicates that the Sun compilers should include interval support.

The compiler options used for gcc on *partch* were:

```
-O3 -march=pentium4 -funroll-loops
```

The procedure was run 1000 times to ensure a large enough runtime for accurate measurement, and to amortise the cost of initial cache loading across the whole benchmark. The elapsed time for the full 1000 repetitions of the kernel was measured using calls to the `gettimeofday` function (which returns the current system time). From the elapsed time, the number of nanoseconds per operation (one iteration of the body of the loop kernel) was calculated. The minimum of three runs was taken.

3.4 Basic interval operations

The first benchmarks were run to determine the performance of interval addition and multiplication on *alcatraz*, relative to the corresponding floating-point operations. The kernels of the benchmarks are as follows:

```
! F1/I1: Sum
do i = 1, size(input)
    result = result + input(i)
enddo

! F2/I2: Product
do i = 1, size(input)
    result = result * input(i)
enddo
```

Most implementations of interval arithmetic are done as class libraries, C++ template libraries, or Fortran modules. In contrast, the Sun Fortran compiler provides support for interval arithmetic as a language feature. Because of this, the interval features of Sun's Fortran compiler are very easy to use; in simple cases, floating-point codes can be changed to use interval arithmetic merely by changing the type declaration of the variables, for example

```
real(kind=8), dimension(:) :: input
```

becomes

```
interval(kind=8), dimension(:) :: input
```

The vectors were initialised with random numbers using the `random_number` Fortran intrinsic subroutine. Sun's Fortran compiler provides both floating-point and interval versions of this subroutine.

3.4.1 Benchmarks F1/I1 and F2/I2: "Basic"

The same code was run for floating-point (F1, F2) and interval arithmetic (I1, I2). The results are shown (as a function of vector size) in Table 3.2.

Both floating-point sum and product benchmarks took slightly over 1.5 cycles per operation for all vector sizes below 2048000. The interval benchmarks took significantly longer. It is worthwhile to compare the measured performance with what might be expected on this architecture.

The UltraSPARC-III is a four-way superscalar, pipelined RISC processor. Each cycle, it can issue up to four instructions; of these, up to two may be floating-point operations (FLOPs) and one may be a memory operation (load, store or prefetch).[1]

[1] Under certain (optimal) circumstances, two memory operations may be issued per cycle. [3, §4.4.3, §4.6.3] This depends on optimal use of the prefetch cache, and is very rarely observed in practice.

	Floating Point		Interval	
Vector size	Sum (F1)	Product (F2)	Sum (I1)	Product (I2)
1000	1.79	1.79	35.7	102
2000	1.73	1.73	35.7	102
4000	1.70	1.70	35.8	102
8000	1.68	1.68	51.9	117
16000	1.67	1.67	57.5	120
32000	1.67	1.67	56.0	122
64000	1.67	1.67	56.8	122
128000	1.67	1.67	56.4	122
256000	1.67	1.67	57.4	122
512000	1.68	1.67	56.6	122
1024000	1.69	1.69	58.0	125
2048000	6.26	6.32	58.6	124
4096000	7.15	6.99	61.9	124

Table 3.2: "Basic" benchmarks on *alcatraz*: nanoseconds per operation (1 cycle = 1.11. . . ns)

Floating-point add and multiply operations take four cycles to complete; peak floating point performance delivers one addition and one multiplication per cycle.

Each iteration of the floating-point sum loop requires one load and one floating-point addition; each iteration of the product loop requires one load and one floating-point multiplication. The UltraSPARC is capable of performing one addition *and* one multiplication per cycle, as well as one load per cycle. Thus these simple benchmarks are *load/store* limited, in the sense that the peak number of FLOPs (addition or multiplication) that can theoretically be performed is greater than the number of vector elements that can actually be loaded into registers.

3.4.2 Cache effects

It was noted in § 3.4.1 that the floating-point benchmark times are almost flat for the floating-point codes for vector sizes below 2048000. Above this, they get worse by a factor of 3–4. This is because the UltraSPARC's 8MB L2 cache is only large enough to store 1048576 double-precision floating point values. Once the vector size increases above this number, the runtime includes the latency of loads from main memory.

This can be confirmed using hardware performance counters, of which the UltraSPARC offers a wide selection [20]. The `cputrack` command was used to track the number of L2 cache misses (performance counter `EC_misses`) in two runs. The first run was for a vector size of 1024000 and experienced 2240212 cache misses. The second was for a vector of 2048000 and experienced 219734835 misses. There were over one hundred times as many L2 misses in the second run, even though the vector size was only twice as large.

It is worth noting for the floating-point results that there are no discernible L1 cache effects. The 64KB L1 cache is only large enough to fit 8192 double-precision

floating point values, so if L1 cache misses were affecting performance one would expect a change in the benchmark between vector sizes 8000 and 16000. It was determined by inspecting the assembly code that the compiler has generated prefetch instructions in the loop. The prefetch instruction is used to draw data into higher cache levels before they are needed, which reduces the effective cost of cache misses to zero, thus masking the L1 cache effect.

Conversely, the interval results do show a slight L1 cache effect. This results in a difference in performance between vector sizes 4000 and 8000; because intervals are represented by two floating-point numbers, only half as many interval elements can be stored in cache as compared to floating-point elements.

From the above results, there are three 'size regions' in which different performance might be expected. These correspond to vectors that are able to be stored in L1 cache, L2 cache, and main memory. Therefore for all subsequent benchmarks, results will only be reported for three different vector sizes, *small, medium & large*, as designated in Table 3.3.

	Vector size	
Size	Floating-point	Interval
Small	2000	1000
Medium	512000	256000
Large	40960000	20480000

Table 3.3: Vector size designations on *alcatraz*

3.4.3 Reasons for poor interval performance

The interval results are far worse than the floating-point results. Where the vector sizes fit in L2 cache, the interval sum is 19–35 times slower than the floating-point sum and the interval product is 57–74 times slower. For vectors that are larger than the L2 cache, the major component of the floating-point times is the cost of cache misses, which means that the interval code is less slow relative to the floating-point code.

To explain why the interval code is so much slower, it is necessary to look both at what is actually required for an interval arithmetic operation, and also at the assembler that the compiler has generated.

3.4.3.1 Inherent cost of interval addition

For each iteration of the floating-point kernel, the following floating-point addition is performed:

$$sum = sum + x$$

In the case of the interval kernel, in which each interval is represented using two floating-point values, this becomes

$$[\underline{sum}, \overline{sum}] = [\underline{sum} + \underline{x}, \overline{sum} + \overline{x}]$$

An extra floating-point addition is performed, and of course two values (the end-points) are loaded to represent x, instead of one. The 'theoretical' cost of an interval addition is therefore twice that of the floating-point case. Thus the inherent cost of interval addition does not by itself explain the slowdown that was shown in Table 3.2.

3.4.3.2 Inherent cost of interval multiplication

While it is easy to analyse the cost of the interval sum operation, the interval multiplication is not so straightforward. This is because there are several equivalent ways of performing the operation, which contain different mixes of instruction types (floating-point, branch, and binary). Three of these will be discussed further in § 3.4.6.2. At this point, it is remarked that the definition of interval multiplication in § 2.3.2 suggests that the cost of interval multiplication will be at least eight times that of regular floating-point multiplication, as it will require eight floating-point multiplies and two min/max operations.

3.4.3.3 Procedure calls and loop unrolling

If the interval sum function is compiled to assembly code, the following instructions can be seen in the body of the loop:

```
.L900000108:                        /* loop start */
    prefetch   [%i1+256],0
    or         %g0,%l1,%o0
    add        %l3,1,%l3
    prefetch   [%i1+264],0
    ldx        [%i1],%l5            /* load interval operands */
    ldx        [%i1+8],%o7
    ldx        [%i3+8],%l4
    add        %i1,16,%i1
    stx        %l6,[%fp+2023]       /* place operands on stack */
    stx        %l5,[%fp+2007]
    stx        %o7,[%fp+2015]
    stx        %l4,[%fp+2031]
    call       __di_add_f           /* add intervals */
    or         %g0,%l0,%o1
    std        %f0,[%i3]
    cmp        %l3,%l2
    std        %f2,[%i3+8]
    ble,a,pt   %icc,.L900000108 /* branch to loop start */
    ldx        [%i3],%l6            /* (branch delay slot) */
```

What has happened is that the compiler has turned the interval addition into a procedure call, in which the operands are placed on the stack.

Procedure calls can be quite detrimental to high-performance code. Parameters, local variables and a return counter may be placed on the stack before jumping to the start of the procedure, and reloaded to registers on return. This overhead may sometimes be avoided in the SPARC architecture through the use of 'register windows' [60]; which allow procedure calls without the overhead of placing parameters on the stack. However, in this case, the compiler has not taken advantage of the register window to pass data.

More importantly for this benchmark, the compiler may be unable to generate sequences of operations that can efficiently pipelined, which will reduce the throughput on a deeply pipelined processor like the UltraSPARC. Finally, an optimising compiler cannot unroll a loop that contains a procedure call, because it cannot see what side effects the procedure may have. Compare the interval assembly code with the body of the floating-point loop, which has been unrolled by eight (divided into eight partial sums) and contains no procedure calls:

```
.L900000105:                          /* loop start */
    prefetch   [%o0+528],0
    faddd      %f4,%f8,%f8            /* partial sum 1 */
    add        %o2,8,%o2
    ldd        [%o0],%f4
    faddd      %f2,%f6,%f6            /* partial sum 2 */
    cmp        %o2,%o5
    ldd        [%o0+8],%f2
    faddd      %f12,%f0,%f0           /* partial sum 3 */
    ldd        [%o0+16],%f12
    add        %o0,64,%o0
    faddd      %f10,%f4,%f16          /* partial sum 4 */
    ldd        [%o0-40],%f10
    prefetch   [%o0+496],0
    faddd      %f8,%f2,%f4            /* partial sum 5 */
    ldd        [%o0-32],%f14
    faddd      %f6,%f12,%f2           /* partial sum 6 */
    ldd        [%o0-24],%f8
    faddd      %f0,%f10,%f12          /* partial sum 7 */
    ldd        [%o0-16],%f6
    faddd      %f16,%f14,%f10         /* partial sum 8 */
    ble,pt     %icc,.L900000105       /* branch to loop start */
    ldd        [%o0-8],%f0
```

Loop unrolling is often beneficial because it allows a reduction in the proportion of loop management (counter increment, comparison and branch) instructions in the body of the loop. It also enables greater separation between dependent instructions,

which may reduce the number of cycles that the processor spends 'stalled' waiting for the result of a previous instruction.

3.4.3.4 Rounding mode switching

The other point of interest in the assembly code can be seen by disassembling the `__di_add_f` procedure used by the interval code. It contains the following instructions:

```
        siam        6
...
! some code to add the two intervals
...
        siam        0
```

The *siam* (Set Interval Arithmetic Mode) instruction on the UltraSPARC changes the rounding mode that is used for floating point arithmetic. [2, p21] The rounding modes specified in IEEE 754 [33] are supported, for example `siam 6` overrides the default rounding mode ("round to nearest") by setting it to "round towards positive infinity" and `siam 0` restores the default rounding mode.

Changing the rounding mode is necessary to calculate correct interval results: lower bounds must be rounded downward and upper bounds rounded upwards to ensure that floating point rounding errors are bounded in the result. However, the change of rounding mode is an instruction that is not required for the floating-point code and so must be considered as an overhead of interval arithmetic.

It is interesting to note that in the assembly code for `__di_add_f`, the rounding mode is only changed twice: once to "round towards positive infinity" and once to "round to nearest", i.e. the interval addition is performed entirely in one rounding mode. This is possible because the following sequences of operations give the same results in IEEE 754 arithmetic [37]:

```
interval a, b, c

// Sequence 1
set rounding mode (towards negative infinity)
inf(c) = inf(a) + inf(b)
set rounding mode (towards positive infinity)
sup(c) = sup(a) + sup(b)
set rounding mode (towards nearest)

// Sequence 2
set rounding mode (towards positive infinity)
inf(c) = -inf(a) - inf(b)
sup(c) = sup(a) + sup(b)
inf(c) = -inf(c)
set rounding mode (towards nearest)
```

A similar arrangement of negations permits an interval product to be calculated entirely in one rounding mode (see below).

To efficiently execute a number of interval arithmetic operations on the UltraSPARC, it is necessary to pipeline the operations. As well as taking an instruction slot, the `siam` operation cannot be grouped with other instructions. [3, §4.4.5.3] It is therefore desirable to perform all operations in a single rounding mode, because the `siam` operation reduces the number of operations that can be pipelined.

3.4.3.5 Summary of reasons for poor performance

In summary it appears that the increased execution time of the "basic" interval benchmark s I1 and I2 compared to the floating-point benchmarks F1 and F2 stems from the following sources:

1. The "inherent" cost of interval arithmetic (twice that of the floating-point operation for addition; at least eight times for product).

2. Inefficiency in the interval code due to lack of loop unrolling.

3. The overhead of the procedure calls that implement the interval operations.

4. The additional instructions required to set the correct rounding modes for interval arithmetic.

To examine the contributions from each of these sources, a series of benchmarks were constructed.

3.4.4 Benchmarks to investigate reasons for poor performance

3.4.4.1 Benchmarks F3 and F4 "No unrolling"

These benchmarks were built from exactly the same code as F1 and F2, the only difference being that the compiler was passed the option `-unroll=1`, which instructs it not to unroll any loops. The effect was confirmed by inspection of the assembly code, which contained a single floating-point addition or multiplication for each iteration of the loop body.

```
.L900000105:                   /* loop start */
    prefetch  [%o1+512],0
    faddd     %f2,%f0,%f2      /* add to sum */
    add       %o4,1,%o4
    add       %o1,8,%o1
    cmp       %o4,%o5
    ble,a,pt  %icc,.L900000105 /* branch to loop start */
    ldd       [%o1],%f0
```

The results are shown in Table 3.4, in comparison to the original (basic) benchmark results.

The non-unrolled codes F3 and F4 are roughly 2.5–3.3 times slower than the original floating-point codes, where the vectors fit into L2 cache.

3.4.4.2 Benchmarks F5 and F6 "Procedure call"

To assess the effect of procedure calls on performance, the code for these benchmarks was altered so that the basic operations were contained in functions in separate files. This has the effect of making the operation into a procedure call, in the same way that the interval operations were turned into procedure calls by the compiler. The results are shown in Table 3.4.

The "procedure call" benchmarks are roughly 13–19 times slower than the original floating-point codes, where the vectors fit into L2 cache. As noted above, one of the effects of including a procedure call in the body of a loop is to prevent the compiler from unrolling the loop. Therefore the slowdown for the "procedure call" benchmarks should be considered to include the slowdown from the "no unrolling" benchmarks.

3.4.4.3 Benchmarks F7 and F8 "Switching"

The code for these benchmarks is the same as that for the "procedure call" benchmarks, with the addition of two calls to inlined assembly functions `round_up` and `round_nearest`, upon entry and exit of each iteration of the loop. The assembly functions are as follows:

```
.inline round_up_,      0
siam                    6
.end

.inline round_nearest_,0
siam                    0
.end
```

The addition of these instructions simulates the cost of the additional `siam` instructions borne by the interval code, and of performing arithmetic with directed rounding. The results are shown in Table 3.4.

The "switching" benchmarks are roughly 14–21 times slower than the original floating-point codes, where the vectors fit into L2 cache. The additional cost of switching rounding modes is around 4.4ns for the sum and 2ns for the product, approximately four and two cycles respectively.

3.4.4.4 Summary of investigation of poor performance

At this point, most of the slowdown of the interval sum has been attributed to the three factors of loop unrolling, procedure calls and switching rounding modes. The remainder is probably due to the inherent cost of the interval addition (twice that of

	Basic		No unrolling		Procedure call		Switching	
Vector size	Σ (F1)	Π (F2)	Σ (F3)	Π (F4)	Σ (F5)	Π (F6)	Σ (F7)	Π (F8)
Small	1.73	1.73	4.47	5.57	24.5	24.5	28.9	26.9
Medium	1.68	1.67	4.48	5.56	29.4	30.7	33.8	32.4
Large	7.15	6.99	6.99	7.75	54.5	57.1	56.4	54.4

Table 3.4: Floating-point benchmarks to investigate poor interval performance on *alcatraz*: ns per operation (1 cycle = 1.11. . . ns)

floating-point addition). Less of the slowdown of the product has been attributed, but the inherent cost of interval multiplication is expected to be greater than for addition (eight times that of floating-point multiplication).

In summary, the Sun Fortran compiler turns interval operations into calls to assembly procedures, which means that interval operations are significantly more expensive than the equivalent floating point operations.

3.4.5 Intrinsic array functions

As part of its support for interval arithmetic in Fortran 95, Sun provides optimised array intrinsics [5](1-3) for exactly the same operations (sum and product) that were performed in the previous benchmarks. Benchmarks were constructed to test the performance of these intrinsics, to find out to what extent Sun's optimisation improves performance.

3.4.5.1 Benchmarks F9/I3 and F10/I4: "Intrinsics"

To use these intrinsics, the kernels of the benchmarks were changed to:

```
! F9/I3: Sum Intrinsic
result = sum(input)

! F10/I4: Product Intrinsic
result = product(input)
```

The results are shown in Table 3.5, in comparison with the original (basic) benchmark results.

	Floating Point				Interval			
	Basic		Intrinsic		Basic		Intrinsic	
Vector size	Σ (F1)	Π (F2)	Σ (F9)	Π (F10)	Σ (I1)	Π (I2)	Σ (I3)	Π (I4)
Small	1.74	1.73	1.72	1.73	69.1	119	4.71	66.6
Medium	1.68	1.67	1.67	1.67	90.7	139	16.3	78.0
Large	7.15	7.14	7.09	7.02	92.3	141	73.0	136

Table 3.5: "Intrinsics" benchmarks on *alcatraz*: ns per operation (1 cycle = 1.11. . . ns)

Firstly, it is worth noting that, for the floating-point code, there is no great difference in times between the array intrinsics and the original benchmarks. This suggests that the Sun Fortran 95 compiler has generated the same quality of code as is used in the optimised intrinsics.

The interval intrinsics results display a few interesting features. They are significantly faster than the basic interval results – the sum is around fifteen times faster, but the product is only twice as fast – and there is a noticeable difference in performance between all three vector sizes.

As discussed in § 3.4.2, the L1 cache on *alcatraz* is large enough to fit 8192 double-precision values. Because each interval is represented by two such values, the L1 cache can not hold more than 4096 intervals. Inspection of the assembly code for the Sun interval array intrinsics `__f_sum_s1_ia16` and `__f_product_s1_ia16` shows that they do not contain prefetch instructions. They have effectively been optimised for small array sizes, and perform poorly on arrays that do not fit in the L1 cache.

3.4.6 Handwritten interval operations

While Sun's Fortran 95 interval support assist in rapid development of interval codes, the resulting codes fail to perform comparably well with floating-point codes. Sun's interval array intrinsics for sum and product operations are significantly slower than the corresponding floating-point operations. It has also been found that basic interval operations in Fortran are converted by the compiler into procedure calls, which causes problems for the optimising compiler. To see whether it is possible to improve on the performance of Sun's interval support, some handwritten interval codes were constructed.

The Fortran 95 codes are contained in a module `sintervals` which defines the type

```
type sinterval
  real(kind=8) :: inf
  real(kind=8) :: sup
end type
```

This type represents an interval in the simplest way possible, using double precision for the endpoints. Sum and product operations were written to perform operations on `sinterval` arrays.

3.4.6.1 Benchmarks I5 and I6: "Handwritten sums"

Handwritten sum

There is only one straightforward way to perform interval addition, and that is simply to add the endpoints. As previously mentioned, one endpoint can be negated to enable correct interval results to be computed using only one rounding mode. The kernel of the sum benchmark is:

```
! I5: Handwritten sum
  call round_up

  sum%inf = -0.0d0
  sum%sup = 0.0d0
  do i = 1, size(input)
    sum%inf = sum%inf - input(i)%inf
    sum%sup = sum%sup + input(i)%sup
  enddo
  sum%inf = -sum%inf

  call round_nearest
```

The subroutines `round_up` and `round_nearest` referenced in this code are implemented for the UltraSPARC as inline assembly procedures that call the `siam 6` and `siam 0` instructions.

Compensated sum

The compensated sum is a method intended to reduce the rounding error in summation. It is discussed in detail in § 4.2.5.

The number of floating-point operations required for a floating-point compensated sum of n numbers is $4(n-1)$, as compared to $n-1$ operations for a simple (recursive) sum. Similarly, the interval compensated sum requires $8(n-1)$ flops compared to $2(n-1)$ for a simple sum.

The code for this benchmark is as follows:

```
  real(8) :: si, ss, ei, es, xi, xs, ti, ts
  si = -0d0
  ss = 0d0
  ei = 0d0
  es = 0d0

  call round_up

  do i = 1, size(input)
    xi = input(i)%inf - ei
    xs = input(i)%sup + es
    ti = si
    ts = ss
    si = si - xi
    ss = ss + xs
    ei = ti - si - xi
    es = ts - ss + xs
  end do

  comp_sum_i%inf = -si
```

```
comp_sum_i%sup = ss

call round_nearest
```

Ogita, Rump and Oishi [49] present an algorithm to perform summation in $7(n-1)$ flops. They claim the result calculated in this way is more accurate than that of compensated summation. They also present a related algorithm to accurately calculate a dot product in $25n-7$ flops.

Results

The results of the handwritten sum benchmark are shown in Table 3.6, in comparison with the "basic" and intrinsic interval sum results. Results are also given for the handwritten interval compensated sum.

	Sun operations		Handwritten	
Vector size	Basic Σ (I1)	Intrinsic Σ (I3)	Standard Σ (I5)	Compensated Σ (I6)
Small	69.1	4.71	6.70	17.8
Medium	90.7	16.3	6.67	19.1
Large	92.3	73.0	36.6	37.8

Table 3.6: Fortran sum benchmarks on *alcatraz*: ns per operation(1 cycle = 1.11...ns)

The Sun intrinsic is faster for small array sizes; this is because it lacks prefetch instructions that are present in the compiled handwritten code. The handwritten sum is much faster for medium and large array sizes.

The compensated sum is around 2.6–2.9 times more expensive than the standard (handwritten) interval sum. This is less than the theoretical cost of four times the standard sum. The reason is that the interval sum is usually load/store limited, as the number of load and prefetch instructions needed for each iteration of the sum is greater than the number of flops, whereas no additional memory instructions are needed for the compensated sum.

3.4.6.2 Benchmarks I7 and I8: "Handwritten products"

The product is not so simple. At least three distinct methods of performing an interval multiplication are known; there may be more. Two methods are outlined below with performance results; the third (see § 3.4.7) requires additional hardware support and so no results are given.

Minimum / maximum of all products

The simplest definition of interval multiplication is

$$X \times Y = [a, b]$$

$$a = \min(\lfloor \underline{X} \times \underline{Y} \rfloor, \lfloor \underline{X} \times \overline{Y} \rfloor, \lfloor \overline{X} \times \underline{Y} \rfloor, \lfloor \overline{X} \times \overline{Y} \rfloor)$$

$$b = \max(\lceil \underline{X} \times \underline{Y} \rceil, \lceil \underline{X} \times \overline{Y} \rceil, \lceil \overline{X} \times \underline{Y} \rceil, \lceil \overline{X} \times \overline{Y} \rceil)$$

where the floor and ceiling symbols indicate that inexact products shall be rounded downward and upward to a representable floating-point number.

The most obvious way to implement interval multiplication is, following this definition, to compute all combinations of endpoint products and then take the minimum and maximum. The kernel for the product benchmark that uses this method of multiplication, "min/max product", is as follows:

```
call round_up

product_i%inf = -1.0d0
product_i%sup = 1.0d0

do i = 1, size(input)
  tmpi = product_i%inf *   input(i)%inf
  tmps = product_i%inf * (-input(i)%inf)
  tmpi = max(tmpi, product_i%inf *   input(i)%sup )
  tmps = max(tmps, product_i%inf * (-input(i)%sup))
  tmpi = max(tmpi, product_i%sup * (-input(i)%inf))
  tmps = max(tmps, product_i%sup *   input(i)%inf )
  product_i%inf = max(tmpi, product_i%sup * (-input(i)%sup))
  product_i%sup = max(tmps, product_i%sup *   input(i)%sup )
enddo

product_i%inf = -product_i%inf

call round_nearest
```

This code uses negation of the lower bounds in a similar way to the sum code, so that the entire operation may be performed in a single rounding mode. Of course the negation of the lower bounds means that the *min* operation in the definition becomes a call to `max` in the code.

Because the intervals must be rounded outwards to bound floating-point rounding errors, the four combinations of endpoint products must be computed twice: once rounding downwards (upwards negated) for the lower bound, and once rounding upwards for the upper bound. This is why the code contains eight floating-point multiplications, instead of the expected four.

The UltraSPARC-III provides a *conditional move* instruction, which sets an integer or floating-point register based on the result of a comparison. Inspection of the generated assembly code shows that the compiler has generated conditional move instructions to implement the calls to the `max` intrinsic.

Branching product

The second method for interval multiplication aims to reduce the number of floating-point multiplications at the expense of introducing branches into the code.

This method is based on the fact that it is possible in most cases to determine which combinations of the endpoints of the operands should be multiplied to form the result,

given only the signs of the endpoints. For example, if both operands are entirely non-negative, the result can be calculated by multiplying the two lower bounds and the two upper bounds:

$$\begin{aligned} x &= [\underline{x}, \overline{x}] \\ y &= [\underline{y}, \overline{y}] \\ \{\underline{x}, \overline{x}, \underline{y}, \overline{y}\} &\geq 0 \\ x \times y &= [\underline{x} \times \underline{y}, \overline{x} \times \overline{y}] \end{aligned}$$

It is possible to divide all interval multiplications into nine classes, depending on whether each of operands is entirely positive, entirely negative, or contains zero. In eight of these nine classes, the result can be calculated by performing only two multiplications, as in the example above.

In the case where both intervals contain zero, it is necessary to perform two multiplications for each end point and take the minimum or maximum.

Signs				Result	
$\underline{a}$	$\overline{a}$	$\underline{b}$	$\overline{b}$	$\underline{c}$	$\overline{c}$
+	+	+	+	$\underline{a} \times \underline{b}$	$\overline{a} \times \overline{b}$
+	+	-	-	$\overline{a} \times \underline{b}$	$\underline{a} \times \overline{b}$
+	+	-	+	$\overline{a} \times \underline{b}$	$\overline{a} \times \overline{b}$
-	-	+	+	$\underline{a} \times \overline{b}$	$\overline{a} \times \underline{b}$
-	-	-	-	$\overline{a} \times \overline{b}$	$\underline{a} \times \underline{b}$
-	-	-	+	$\underline{a} \times \overline{b}$	$\underline{a} \times \underline{b}$
-	+	+	+	$\underline{a} \times \overline{b}$	$\overline{a} \times \overline{b}$
-	+	-	-	$\overline{a} \times \underline{b}$	$\underline{a} \times \underline{b}$
-	+	-	+	$\min(\underline{a} \times \overline{b}, \overline{a} \times \underline{b})$	$\max(\underline{a} \times \underline{b}, \overline{a} \times \overline{b})$

The full kernel for this handwritten method of interval multiplication is significantly more complicated:

```
call round_up
product%inf = -1.0d0
product%sup = 1.0d0

do i = 1, size(input)
  if (product%inf > 0.0d0) then
    if (input(i)%inf > 0.0d0) then
      tmp = -product%inf * input(i)%inf
      product%sup = product%sup * input(i)%sup
      product%inf = tmp
    else if (input(i)%sup < 0.0d0) then
      tmp = -product%sup * input(i)%inf
```

```
      product%sup = product%inf * input(i)%sup
      product%inf = tmp
    else
      product%inf = -product%sup * input(i)%inf
      product%sup = product%sup * input(i)%sup
    end if

  else if (product%sup < 0.0d0) then
    if (input(i)%inf > 0.0d0) then
      product%inf = -product%inf * input(i)%sup
      product%sup = product%inf * input(i)%inf
    else if (input(i)%sup < 0.0d0) then
      tmp = -product%sup * input(i)%sup
      product%sup = product%inf * input(i)%inf
      product%inf = tmp
    else ! input(i) contains 0
      tmp = -product%inf * input(i)%sup
      product%sup = product%inf * input(i)%inf
      product%inf = tmp
    end if

  else ! product contains 0
    if (input(i)%inf > 0.0d0) then
      product%inf = -product%inf * input(i)%sup
      product%sup = product%sup * input(i)%sup
    else if (input(i)%sup < 0.0d0) then
      tmp = -product%sup * input(i)%inf
      product%sup = product%inf * input(i)%inf
      product%inf = tmp
    else ! both product and input(i) contain 0
      tmp = -product%inf * input(i)%sup
      tmp2 = product%inf * input(i)%inf
      product%inf = -product%sup * input(i)%inf
      product%inf = max(product%inf, tmp)
      product%sup = product%sup * input(i)%sup
      product%sup = max(product%sup, tmp2)
    end if
  end if

  product%inf = -product%inf
enddo

call round_nearest
```

There are eight `if` tests in this code, each of which translates to a branch instruction in the assembly code, hence the name "Branch product" for this method. Inspection of the FILIB § A.2 code (FILIB source file `interval/interval_arith.icc`) shows that this is the method used in FILIB for interval multiplication. Inspection of the Sun assembly code for multiplying two intervals, `di_mul_f`, reveals a large number of branches, which suggests that Sun also use this method.

Branch instructions can be problematic if it is difficult to predict in advance which branch the code will take. The UltraSPARC executes instructions in order, has a deep pipeline, and performs branch prediction. This means that during execution of code that contains branch instructions, it may have many instructions underway in the pipeline that depend on the results of a branch prediction that has not yet been resolved. If the prediction is incorrect, the pipeline must be flushed and reloaded with the instructions corresponding to the correct branch, which results in wasted cycles. This means that branches that have a high miss rate (such as those in the branch product code that are based on the signs of [random] intervals) will result in a large average cost. To ensure that intervals of different signs were used, the intervals for these benchmarks were generated using the Sun Fortran intrinsic function `random`.

Results

Given the considerations discussed above, it is no surprise that the "branch product" runs more slowly than the "min/max product" on the UltraSPARC-III (see Table 3.7). These results suggest that the performance of Sun's interval multiplication support on the UltraSPARC could be improved by making greater use of the conditional move instructions.

	Sun operations		Handwritten	
Vector size	Basic (I2) Π	Intrinsic Π (I4)	Min/Max Π (I7)	Branch Π (I8)
Small	119	66.6	26.8	37.0
Medium	139	78.0	27.2	38.5
Large	141	136	56.5	48.5

Table 3.7: Fortran product benchmarks on *alcatraz*: ns per operation(1 cycle = 1.11... ns)

3.4.7 Proposal: improved hardware support for interval multiplication

While the min/max product runs faster than the compiler's branching implementation on the UltraSPARC, it is still not optimal.

Currently, eight floating-point multiplications are required, corresponding to the four combinations of operand endpoints, rounded upwards and downwards. However, it should only be necessary to perform four full floating-point multiplications, where the architecture provides a means to check whether the previous operation was rounded. A pair of instructions are proposed to achieve this end.

The first instruction, 'multiply and set condition code' (*fmulc*), would perform a rounded floating-point multiplication, and set a condition code to indicate whether the operation was exact or required rounding.

The second instruction, 'move and change interval rounding' (*fmvi*), would first copy from one floating-point register to a destination register. The destination register may then be increased to the next highest (or decreased to the next lowest) floating-point number, depending on the previously set condition code. By executing these two instructions in sequence, it is possible to calculate the result of a floating-point multiplication rounded in both directions with only a single full multiplication.

These new instructions by themselves would not greatly improve the performance of interval multiplication on the UltraSPARC. The UltraSPARC executes floating-point instructions in two separate pipelines: FGA (floating-point addition) and FGM (floating-point multiply). [3] Each floating-point instruction type may be executed on only one of the two pipelines. Inspection of assembly code shows that each min or max operation comprises a floating-point comparison (*fcmp*) and a conditional move (*fmovd*). Both of these instructions are executed in the UltraSPARC FGA pipeline, which means that the min/max code actually requires more instructions in the FGA pipeline than in the FGM pipeline.

To reduce the number of FGA instructions required for an interval multiplication, a third new instruction type is proposed. This is a 'floating-point maximum' (*fmax*) instruction that would accept two input registers and place the greater of the two values in an output register. (An analogous *fmin* instruction would implement the floating-point minimum.) The latency of *fcmp* is one cycle and *fmovd* is three cycles; for each max operation, the total latency is four cycles. The latency of *fmin* / *fmax* need not be less than four cycles; simply combining these operations into a single instruction would mean that twice as many interval multiplications could be overlapped in the FGA pipeline.

None of the three new instruction types proposed above would require significant change to existing hardware. Used in conjunction, they could greatly improve the speed of interval multiplication on the UltraSPARC platform, by reducing the number of instructions executed the FGA pipeline, and the latency of instructions in the FGM pipeline.

Stine and Schulte[53] have proposed an interval multiplier that would avoid the need for branches in most interval multiplications, thereby greatly increasing the speed. Their *combined interval and floating-point multiplier* would incorporate additional control logic for switching rounding modes, and multiplexors to determine which endpoints to multiply together. However, because this scheme requires significant changes to existing hardware, it is unlikely to be implemented for widely-used architectures in the near future.

3.4.8 Comparison of interval performance: UltraSPARC vs. IA-32

3.4.8.1 Benchmarks I9, I10 and I11: "Handwritten interval", C++ on *alcatraz*

To allow a proper comparison of interval performance between the UltraSPARC and IA-32 platforms, it is desirable that the source codes for the two platforms should be as similar as possible.

At the time these benchmarks were conducted, the SunStudio Fortran and C++ compilers were available for Linux as an alpha release. The SunPerf library was not yet available. and so could not be used on *partch*. The performance of the Sun interval support was not compared between UltraSPARC and IA-32. However, it was possible to compare the performance of handwritten codes.

The handwritten Fortran codes were rewritten in C++ and compiled using the Sun C++ compiler. The results are shown in Table 3.8.

It is notable that the handwritten sum performs markedly better when compiled with the Sun C++ compiler than when compiled with the Sun Fortran compiler (for which results are also shown in Table 3.8). The assembly code from the C++ compiler shows less prefetch (two rather than four) and loop increment (one rather than four). Thus it seems that the C++ compiler is able to make more efficient use of the UltraSPARC prefetch cache.

	C++			Fortran
Vector size	Sum (I9)	Min/Max Product (I10)	Branch Product (I11)	Sum (I5)
Small	3.65	27.0	26.8	6.70
Medium	3.35	26.8	38.0	6.67
Large	13.9	31.0	90.0	36.6

Table 3.8: "Handwritten" C++ benchmarks on *alcatraz*: ns per operation (1 cycle = 1.11... ns)

3.4.8.2 Benchmarks I12, I13 and I14: "Handwritten interval", C++ on *partch*

The Intel Pentium 4 is a three-way superscalar, pipelined CISC processor that implements the IA-32 instruction set. Each cycle, it can issue up to three instructions, of which up to two may be floating-point operations and one may be a memory operation. The sum and product benchmarks are therefore load/store limited on the Pentium 4 just as they are on the UltraSPARC.

In contrast to the UltraSPARC, the Pentium 4 allows out-of-order and speculative execution [6] (2-6). Instructions are retired (the results are committed to memory and/or registers) only once branch comparisons and other dependencies are completed. It was expected therefore that the branching method of interval multiplication would be better suited to the Pentium 4 than to the UltraSPARC.

The Pentium 4 has eight general purpose registers and eight floating-point registers; parameters are passed to and from procedures via the stack [6] (3-3). It is therefore expected that the relative cost of procedure calls will be greater than on the UltraSPARC. This is probably the reason why the FILIB, for which the standard implementation is a template library, provides a C++ Macro implementation, in which interval operations are inlined. [37]

The Pentium-4 has only a 8KB L1 cache, which is too small to fit the smallest of the interval vector sizes that were benchmarked on *alcatraz*. Therefore all benchmarked vector sizes from 1000 to 4096000 were either sourced from L2 cache or main memory.

The L2 cache is 512KB, large enough to fit 64000 double-precision values or 32000 double precision intervals. Results for *partch* are reported for only two sizes, listed in Table 3.9.

		Vector size
Size	Floating-point	Interval
Small	2000	1000
Large	40960000	20480000

Table 3.9: Vector size designations on *partch*

The same handwritten C++ codes that were benchmarked on *alcatraz* were compiled on *partch* using gcc 3.4.4. The results are shown in Table 3.10. Because these benchmarks used the same codes that were compiled on alcatraz, they may be directly compared with the results in Table 3.8.

Vector size	Sum (I12)	Min/Max Product (I13)	Branch Product (I14)
Small	2.59	35.2	7.58
Large	6.54	36.3	8.69

Table 3.10: "Handwritten" C++ benchmarks on *partch*: ns per operation (1 cycle = 0.385 ns)

As expected, the branching product code performs significantly better on the Pentium 4 than does the min/max code. This provides justification for the choice of this method of interval multiplication for the FILIB code.

The cache effect is significant for the sum but not for the products. This is because the time taken for other operations (branch, multiplication etc.) masks the latency of loads.

3.5 Compound operations

In the previous section, all benchmarks were aimed at testing the floating-point and interval performance of the basic addition and multiplication operations. This section will discuss the use of these operations to form higher level (compound) operations.

3.5.1 Dot product and AXPY

At the next level of complexity, two widely used operations are the inner- or dot-product and AXPY (Alpha times X plus Y). Again, these operations are components of more complex operations, notably matrix multiplication (see § 3.5.2). These operations are often called Basic Linear Algebra Subprograms, Level 1 (BLAS1) [36; 16].

For two vectors X and Y both of length n, the dot product is $X.Y = \sum_i x_i y_i$. To compute the dot product requires n multiplications and $n-1$ additions. To this must be added $2n$ loads for the elements of X and Y.

AXPY is computed as $Y = Y + \alpha X$, which requires n multiplications and n additions. As well as $2n$ loads from X and Y, n stores are performed to update the elements of Y.

3.5.1.1 Benchmarks F11/I15 and F12/I16: "BLAS1 on *alcatraz*"

Two benchmarks were constructed to compare the performance of floating-point and interval implementations of these operations. The dot product is implemented by Sun as a Fortran intrinsic function; AXPY is implemented as a subroutine in the Sun Performance Library. The kernels of the benchmarks are as follows:

```
! F11 / I14: Dot Product
result = dot_product(input, input2)

! F12: AXPY (Floating point)
call axpy(numElements, alpha, x, 1, y, 1)

! I15: AXPY (Interval)
call axpby_i(x, y, alpha)
```

The results are shown in Table 3.11. They show a marked difference in times between medium and large sizes for the floating-point codes, which is the difference between loads from L2 cache and from main memory. The interval codes show a smaller but still significant cache effect between each size. The interval results are 14–32 times slower for the dot product and 16–46 times slower for the AXPY.

	Floating Point		Interval	
Vector size	Dot Product (F11)	AXPY (F12)	Dot Product (I15)	AXPY (I16)
Small	3.43	6.32	81.4	250
Medium	3.34	6.63	108	306
Large	14.6	18.6	216	314

Table 3.11: Fortran intrinsic/library BLAS1 benchmarks on *alcatraz*: ns per operation (1 cycle = 1.11...ns)

Just like the simple (sum and product) benchmarks, the floating-point benchmarks are load/store limited on the UltraSPARC, as they require two or three memory operations for each pair of elements, and only two floating-point operations.

It is not surprising, given that the AXPY operation requires an additional store for every element of the vectors, that this operation is slower than the dot product for all vector sizes.

3.5.1.2 Interval instruction mix

It is interesting to note that while the floating-point dot-product is load/store limited, the interval dot product is not. The numbers of load/store and floating-point

instructions required for dot product and AXPY are shown in Table 3.12, in comparison with the number of each instruction type that can be dispatched per cycle on the UltraSPARC and Pentium 4 architectures.

		Floating-point ops		Memory ops		Ratio
		Add	Multiply	Load	Store	FP : Mem
Dot product	Floating-point	n	n	2n	1	1:1
Requirement	Interval	2n	8n	4n	2	**5:2**
AXPY	Floating-point	n	n	2n	n	2:3
Requirement	Interval	2n	8n	4n	2n	**5:3**
	UltraSPARC cycle	1	1	1		2:1
Limit	Pentium-4 cycle	1	1	1		2:1

Table 3.12: BLAS1 operations: required instructions versus optimal instruction mix for the UltraSPARC-III

This illustrates an interesting property of interval arithmetic, which appears to have general application to non-trivial computation:

An interval arithmetic code requires a higher ratio of floating-point or flow-control operations to memory operations than does a corresponding floating-point code.

The consequence of this is that codes that are load/store limited for floating-point arithmetic tend towards an optimal or floating-point limited instruction mix for interval arithmetic.

3.5.1.3 Benchmarks I17 and I18: "Handwritten BLAS1 on *alcatraz*"

Handwritten codes were developed to implement the dot product and AXPY operations in a similar way to the handwritten sum and product codes. The results are shown in Table 3.13, in comparison with the results from the corresponding Sun intrinsic and library functions.

	Sun subroutines		Handwritten subroutines	
Vector size	Dot Product (I15)	AXPY (I16)	Dot Product (I17)	AXPY (I18)
Small	81.4	250	18.1	34.1
Medium	108	306	18.4	85.0
Large	216	314	94.7	152

Table 3.13: "Handwritten" Fortran BLAS1 benchmarks on *alcatraz*: ns per operation (1 cycle = 1.11...ns)

The handwritten dot product is approximately 4–6 times faster (within L2 cache) than the Sun intrinsic. The handwritten AXPY is approximately 3–7 times faster than the Sun performance library subroutine. It is noteworthy that the handwritten dot product is *faster* than the handwritten vector product (see Table 3.7). This is because each the vector product is implemented as a loop in which each iteration is dependent

on the result of the previous iteration. In comparison, the multiplications between vector elements for the dot product are independent, therefore they may be overlapped, making more efficient use of the floating-point pipelines.

3.5.2 Matrix multiplication

Matrix multiplication is fundamental to many scientific and engineering codes. The product C of two matrices A and B is a matrix defined by:

$$C_{ik} = \sum_j A_{ij} B_{jk}$$

It appears from this definition that, for the common case where $dim(i) = dim(j) = dim(k)$, matrix multiplication is an $\mathcal{O}(N^3)$ operation. In fact, optimisations are known that reduce the complexity to $\mathcal{O}(N^{2.8})$ [54] or as low as $\mathcal{O}(N^{2.376})$ [15]. As well, optimisations are known for sparse, symmetric, triangular and other special classes of matrices [16]. However, for the purposes of benchmarking interval matrix multiplication, the $\mathcal{O}(N^3)$ general matrix multiply was chosen.

3.5.2.1 Benchmarks F13/I19, I20 and I21: "BLAS3 on *alcatraz*"

There are two obvious ways in which a matrix product can be computed, which calculate partial results using either the dot-product or AXPY operations that were discussed in § 3.5.1. The two methods were implemented as benchmark kernels as follows:

```
type(sinterval), dimension(:,:), target :: a, b, c

! F18 / I21: General Matrix Multiply based on dot-product
call round_up
do n = 1, size(a(1,:))
  do p = 1, size(b(:,1))
    c(n,p) = inner_product_i(a(n,:), b(:,p))
  enddo
enddo
call round_nearest

! F19 / I22: General Matrix Multiply based on AXPY
call round_up
c%inf = -0.0d0
c%sup = 0.0d0
do p = 1, size(b(1,:))
  do m = 1, size(b(:,1))
    call daxpy_i(b(m,p), a(:,m), c(:,p))
  enddo
```

```
enddo
call round_nearest
```

Given the results in § 3.5.1, it was expected that the AXPY-based matrix product would be slower for floating-point arithmetic.

The results are shown in Table 3.14 for selected matrix sizes. All matrices are small enough to fit in the L2 cache. For all previous benchmarks, an 'operation' was defined as the total time divided by the number of vector elements and the number of iterations. Here, the definition changes: because this formation of matrix multiplication is an $O(N^3)$ operation, the total runtime was divided by N^3 for a $N \times N$ matrix, thereby giving the time for each pair of add and multiply operations.

	Sun subroutines		Handwritten subroutines	
Matrix dimension	Floating-point SunPerf (F13)	Interval SunPerf (I19)	Interval (I20) (Dot product-based)	Interval (I21) (AXPY-based)
20	2.58	82.9	22.2	37.7
100	1.28	67.0	20.1	18.7
200	1.23	66.5	19.6	18.6

Table 3.14: "General Matrix Multiply" benchmarks on *alcatraz*: ns per operation (1 cycle = 1.11...ns)

The handwritten interval code takes approximately 15 times longer than the Sun Performance Library floating-point subroutine to multiply matrices of the same size. The SunPerf interval routine is three times slower again.

Both the dot-product- and AXPY-based methods for multiplying interval matrices require $2N^3$ floating-point additions and $8N^3$ floating point multiplications each (assuming the use of the min/max method of interval multiplication).

3.6 Conclusions

There are a range of software products to support interval computations on popular architectures. This chapter has given detailed performance results for one such tool, the SunStudio interval support for the UltraSPARC platform.

While the Sun interval implementation supports the easy development of interval code, the performance of this code is slower than might be expected. The reason for this is primarily that the basic interval operations are compiled to procedure calls, which prevents interval operations from being pipelined or optimised by the compiler.

Sun provides some interval subroutines as Fortran intrinsics or as part of the Sun Performance Library. However, the performance of these is still less than optimal. In particular, the Sun subroutines use a branching method of interval multiplication, which has been shown to be less efficient than the predictable 'min/max' method on the UltraSPARC platform.

Despite the performance of the Sun implementation, the work in this chapter shows that it is possible to achieve acceptable interval performance on the UltraSPARC platform by using hand-optimised interval code. With the techniques developed in this chapter, it becomes easier to apply interval analysis to scientific problems, as is demonstrated in Chapter 4.

It is to be expected that the performance of commercial interval packages will improve, as improvements like the ones suggested in this chapter are applied. Therefore, in the future, it should not usually be necessary to use hand-optimised code; the 'off-the-shelf' performance will be acceptable for almost all applications.

Chapter 4

Example application: evaluation of electrostatic potentials

> "Let there be electricity and magnetism!"
> . . . and there was light.
>
> **Richard Feynman**
> Lecture on *Electromagnetic Radiation*
> Caltech, 16 February 1962

The previous chapter dealt with the performance of interval methods. In contrast, this chapter is concerned with the use of interval methods to bound numerical errors.

Interval methods are applied to bound errors in an example problem from computational chemistry. Interval analysis is used to test the numerical stability of alternative algorithms, and to provide verified error bounds on the results. The aim is to show that interval analysis is a useful tool for scientific computation.

Verified results will be given for a range of different methods. Two new methods are developed for accurate summation, and interval analysis is used to prove that these methods give more accurate results for the example problem.

An interval version of a fast multipole method is developed that gives verified bounds on the rounding error of the method. To our knowledge, this is the first time that an interval version of the method has been demonstrated. An extension is proposed that would combine the bounds on rounding error with an *a priori* error bound, to give an interval guaranteed to contain the correct result.

The results reported in this chapter are quantitative statements about the accuracy of computational methods, which could not have easily been obtained in any other way. This demonstrates that interval analysis is a useful tool for scientific computation.

4.1 Problem description

A problem that arises in computational science is that of calculating the potential energy of a system of charged particles under electrostatic forces.

An electric charge exerts a force on another charge that is in the direction of the vector between the two charges, proportional to the product of the charges, and inversely proportional to the square of the distance between them. For example, in Figure 4.1, charge q_i exerts a force on charge q_j of $\frac{kq_iq_j}{{r_{ij}}^2}$, where k is the Coulomb force constant that depends on the permittivity of the medium.

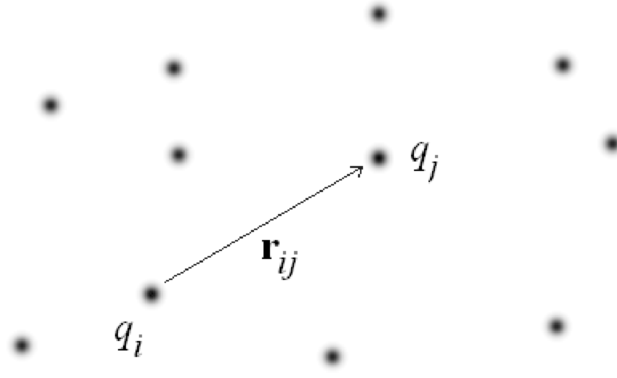


Figure 4.1: Particles of charges q_i, q_j, separated by distance $\boldsymbol{r}_{ij}$, interacting under Coulomb potential

The potential energy of the pair of particles is the work that would be required to move the particles from infinity to current locations, and is given by

$$\frac{kq_iq_j}{|\boldsymbol{r}_{ij}|}$$

Given n particles with charge q_n at locations $\boldsymbol{R}_n$, and Coulomb force constant k, the total potential energy U of the system is

$$U = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{1}{2} \frac{kq_iq_j}{\|\boldsymbol{R}_j - \boldsymbol{R}_i\|}$$

The factor of $\frac{1}{2}$ is necessary to avoid 'double counting' the interaction potential of each pair of particles.

It is required by Newton's third law that the force on particle i due to particle j is of the same magnitude and in the opposite direction as the force on particle j due to particle i. The potential energy is a scalar (directionless) quantity, thus the potential of each particle due to the pairwise interaction is the same. Therefore only one calculation is required for each pair of particles in the system.

From this definition it is apparent that exact calculation of the potential energy is of $\mathcal{O}(n^2)$ complexity, making direct pairwise evaluation impractical for large numbers of particles. Approximate methods exist to calculate potentials for large systems to an arbitrary level of accuracy, most notably the fast multipole method (see § 4.3), which

scales as $\mathcal{O}(n)$. However pairwise evaluation may still be used where the potential calculation is part of a larger computation, other parts of which scale as $\mathcal{O}(n^2)$ or worse.

4.2 Direct evaluation

This section will deal with methods of evaluating the potential that are mathematically equivalent to the definition given above. These methods are collectively called *direct evaluation* methods.

4.2.1 Standard summation

A typical Fortran code to directly evaluate the potential energy is as follows:

```
function system_potential(r, q)
  real(8), intent(in) :: r(:, 3), q(:)
  real(8) :: Eij, system_potential
  system_potential = 0d0
  do i = 1, size(q)
    do j = i+1, size(q)
      Eij = interaction_potential(r(:,i), r(:,j), q(i), q(j))
      system_potential = system_potential + Eij
    end do
  end do
end function system_potential

function interaction_potential(r1, r2, q1, q2)
  real(8) :: r1(3), r2(3), q1, q2, interaction_potential
  real(8), parameter :: k = 8.8551878176d-12

  interaction_potential = k * q1 * q2 / sqrt(sum((r1-r2)**2))
end function interaction_potential
```

In the function `system_potential` above, it can be seen that the interaction between each pair of particles is calculated only once.

There are several potential sources of rounding error in this code. In the function `interaction_potential`, the multiplication, division and square root operations are all sources of rounding error. In the function `system_potential`, the sum over i, j may truncate some or all of the interaction potentials for certain pairs.

It is likely that the rounding error from the multiplications, divisions and square roots will tend to cancel out. This is because each of these operations is a 'round-to-nearest' operation. While the sum also uses 'round-to-nearest' addition, where all summands are positive numbers, the rounding errors tend not to cancel, but to subtract from the sum. To see why, consider the addition of two numbers a, b where

$a \gg b$. If the difference in their exponents is greater than the number of digits in the significand (for example, $exponent(a) - exponent(b) > 52$ in IEEE double precision), then the smaller term is completely cancelled, as in Figure 4.2.

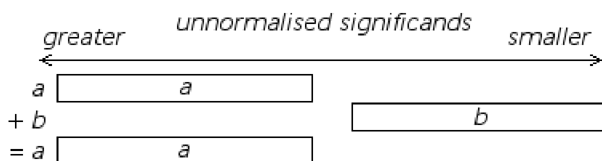


Figure 4.2: Addition of floating-point numbers of greatly differing magnitudes

In a randomly distributed system of particles, there are a small number of close-range (greater) interactions and a large number of long-range (smaller) interactions. If these terms are added together in a random order, many of the long-range interactions will be completely truncated as in Figure 4.2. In the code above, the interactions are simply added together in the order in which they are calculated. This shall be referred to as the *standard summation* method [1].

It is apparent that the accuracy of the direct evaluation of the system potential is greatly dependent on the order of summation of the pairwise interactions. Much has been published on the accuracy of summation; Higham[30] devotes an entire chapter to the subject, and McNamee [42] gives experimental results for fourteen different summation methods proposed by previous authors.

Five methods of summation are considered here:

1. standard (unordered)
2. ordered
3. partially ordered
4. accumulator
5. compensated

The accuracy of four of the methods was evaluated using interval analysis. Codes were constructed to calculate the potential energy of the system using each of the methods.[2] The codes were constructed in such a way that the quantities of interest (distances and energies) could be represented by either real or interval variables, by changing the type declarations using preprocessor statements.

4.2.2 Ordered summation

One method of increasing the accuracy of summation is to order the terms by magnitude. Increasing order is more accurate where all terms are of the same sign (as

[1] called *recursive summation* by Higham
[2] ordered summation was not evaluated experimentally, for reasons given in § 4.2.2

in this case), whereas decreasing order of magnitude is usually more accurate where the terms differ in sign and heavy cancellation occurs. A variant of ordered summation, the *insertion* method, guarantees a result accurate to machine precision where the summands are of the same sign [30].

Unfortunately, ordered summation is not practical for this application, due to the space required to store the ordered summands. Whereas other summation methods allow partial sums to be calculated progressively as individual summands become available, ordered methods require all summands to be accessible to the sorting operation. For a relatively small problem size of 2^{17} particles, this requires all 2^{34} interactions to be stored in memory, requiring 128GB of space. Thus storage size will quickly become the limiting factor if ordered summation is used for this problem. For this reason, ordered summation is inappropriate to this problem, and no results are given for this method.

4.2.3 Partially-ordered summation

Although ordered methods are not feasible, it is possible to construct a partially-ordered method using known features of the problem. Because the magnitudes of the interactions are dependent on the distance between the particles, it is possible to sort the interactions into a finite number of 'buckets', where the interaction distance for each bucket is within a certain range.

As each interaction is calculated, a 'rough distance' is calculated for the interaction, and it is added to the partial sum within the appropriate bucket. After all interactions have been calculated, the partial sums for the buckets are added together from furthest to nearest. The code for this method is as follows:

```
function system_potential_ordered(r, q)
  real(8), intent(in) :: r(:, :), q(:)
  real(8) :: system_potential_ordered
  real(8) :: Eij, dist_bucket(0:max_distance)
  integer :: rough_dist

  dist_bucket(:) = 0d0

  do i = 1, size(q)
    do j = i+1, size(q)
      Eij = interaction_potential(r(:,i), r(:,j), q(i), q(j))
      rough_dist = &
    &   rough_distance(box_index(r(:,i)), box_index(r(:,j)))
      dist_bucket(rough_dist) = dist_bucket(rough_dist) + Eij
    end do
  end do

  system_potential_ordered = 0d0
```

```
  do i = max_distance, 0, -1
    system_potential_ordered = &
  &   system_potential_ordered + dist_bucket(i)
  end do
end function system_potential_ordered

function box_index(r)
  real(8), intent(in) :: r(3)
  integer(4) :: box_index(3)

  box_index = int((r - RMIN) * scale)
end function box_index

function rough_distance(a_idx, b_idx)
  integer(4), intent(in) :: a_idx(3), b_idx(3)
  integer(4) :: rough_distance
  real(8) :: dist

  dist = sqrt(dreal(sum((a_idx - b_idx)**2)))
  rough_distance = dist
end function rough_distance
```

In these calculations, `scale` was chosen so that the mean number of particles in each bucket would be 64. Thus the number of buckets increases with the problem size. Since the distance must be calculated anyway, there is no significant cost in sorting in this way. However, the interactions are not sorted within the buckets, and rounding errors will occur both in the calculation of the sum for each bucket, and in the final summation of the buckets. Nevertheless, the use of this method significantly reduces the cancellation of smaller terms.

4.2.4 Summation with accumulators

It was argued in § 4.2.1 that the source of much of the rounding error in this code is the addition of numbers of differing magnitudes. To attack this problem more directly, it is possible to construct a scheme whereby the majority of additions are between summands of the same magnitude, and in no addition does the magnitude of the summands differ by more than two. This scheme is similar to that proposed by Wolfe [62]; it should not be confused with the scheme proposed by Malcolm [41], in which summands are split into unnormalised segments.

A series of accumulators is constructed, one for each possible exponent in the floating-point representation used. For example, for IEEE 754 double precision, there are 2048 accumulators, with exponents ranging from 2^{-1023} to 2^{1024}. As each pairwise interaction is evaluated, the exponent is determined and the interaction is added to the corresponding accumulator. If this addition causes the exponent of the current

value of the accumulator to increase, it 'overflows'; the accumulator is set to zero and the overflow value is added to the next largest accumulator, as shown in Figure 4.3.

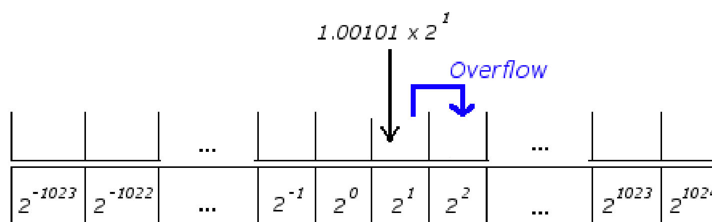


Figure 4.3: Accumulator summation: overflow upon adding to an accumulator

It is possible that the overflow may cause an overflow in the next accumulator, and so on, cascading upwards as far as necessary. In calculating the sum of n terms using a set of m accumulators where $n \gg m$ there are approximately $n/2$ overflows in total. After all pairwise interactions have been added to the accumulators, the accumulators are summed from smallest to largest. Because this scheme avoids (as far as possible) adding together numbers of different magnitude, it was expected that this scheme would give a result accurate to close to machine precision. The code for this scheme is as follows.

```
function system_potential_accum(r, q)
  real(8), intent(in) :: r(:,:), q(:)
  real(8) :: system_potential_accum
  real(8) :: Eij, E_exp(minexponent(r):maxexponent(r))
  integer(4) :: e, e_new

  system_potential_accum = 0d0
  E_exp(:) = 0d0
  do i = 1, size(q)
    do j = i+1, size(q)
      Eij = interaction_potential(r(:,i), r(:,j), q(i), q(j))
      e = exponent(Eij)
      E_exp(e) = E_exp(e) + Eij
      e_new = exponent(E_exp(e))
      do while (e_new /= e)
        ! exponent has changed - overflow to next bucket
        E_exp(e_new) = E_exp(e_new) + E_exp(e)
        E_exp(e) = 0d0
        e = e_new
        e_new = exponent(E_exp(e))
      end do
    end do
  end do
```

```
  do n = minexponent(r), maxexponent(r), 1
    system_potential_accum = system_potential_accum + E_exp(n)
  end do
end function system_potential_accum
```

4.2.5 Compensated summation

A clever scheme to reduce the rounding error in summation was developed for fixed-point arithmetic by Gill [21] and extended to floating-point arithmetic by Kahan [34] and Møller [44]. This scheme, known as *compensated summation*, calculates an error term for each partial sum which is fed into the next partial sum so as to reduce the total error.

Higham [30] gives pseudocode for the sum $s = comp \sum_{i=1}^{n} x_i$ as follows:

$$\begin{aligned} & s = 0; e = 0 \\ & for \quad i = 1 : n \\ & \qquad temp = s \\ & \qquad y = x_i + e \\ & \qquad s = temp + y \\ & \qquad e = (temp - s) + y \\ & end \end{aligned}$$

It is possible to construct an interval compensated sum that performs a separate compensation for each bound. In summing intervals X_i, the lower and upper bounds are treated separately, i.e.

$$comp \sum_{i=1}^{n} X_i = \left[comp \sum_{i=1}^{n} \underline{X_i}, comp \sum_{i=1}^{n} \overline{X_i} \right]$$

Thus the only difference in the evaluation of the interval bounds is the rounding mode.

4.2.5.1 Proof

For an interval sum to be valid, it must satisfy the containment criterion. That is, the sum must completely contain the correct result of adding the interval summands together in infinite precision. It is therefore necessary to prove that the endpoints of the compensated interval sum lie outside the correct endpoints that would be calculated using infinite precision arithmetic. This shall be proved first for the upper bound.

The upper bound is calculated with rounding towards $+\infty$ as follows:

$$
\begin{array}{ll}
1: & s_0 = 0; e_0 = 0 \\
2: & for \quad i = 1 : n \\
3: & \qquad temp = s_{i-1} \\
4: & \qquad y_i = \overline{X_i} + e_{i-1} \\
5: & \qquad s_i = temp + y_i \\
6: & \qquad e_i = (temp - s_i) + y_i \\
7: & end
\end{array}
$$

To simplify the proof, the values of y, s, e for a given iteration i are denoted by y_i, s_i, e_i. However, it is not necessary to actually maintain vectors of these values, as they are only needed for a maximum of one iteration; in practice a single local variable will suffice for each of y, s, e (with a temporary variable to hold s_{i-1}).

It is apparent that the result of the floating-point addition in line 4 may not be exact; the result y_i may be rounded upwards and therefore includes an error $\alpha_i >= 0$ such that $y_i = \overline{X_i} + e_{i-1} + \alpha_i$.

It is also the case that the addition in line 5 may be rounded upwards by $\beta_i >= 0$, therefore $s_i = temp + y_i + \beta_i$.

The benefits of this method stem from the fact that the operations in line 6 are exact. This will be the case in any floating-point system in which

$$c - a = b \longrightarrow a - c = -b$$

This is the case for IEEE 754 floating-point arithmetic.

It is obvious that the assignment in line 3 is exact and generates no error.

Having thus analysed the errors in each line of the loop body, it is possible to determine the values of y_i, s_i, e_i for a given iteration i:

$$
\begin{aligned}
y_i &= \overline{X_i} + e_{i-1} + \alpha_i & &= \overline{X_i} - \beta_{i-1} + \alpha_i \\
e_i &= (s_{i-1} - s_i) + y_i & &= -\beta_i \\
s_i &= s_{i-1} + y_i + \beta_i & &= \sum_{j=1}^{i} \overline{X_j} + \sum_{j=1}^{i} \alpha_j + \beta_i
\end{aligned}
$$

From the definition of s_i it can be seen that value of the upper bound of the compensated interval sum will be greater than the correct value by the total of the rounding errors α_i and the final rounding error β_n:

$$s_n - \sum_{i=1}^{n} \overline{X_i} = \sum_{i=1}^{n} \alpha_i + \beta_n$$

Therefore the calculation of the upper bound satisfies the containment criterion, and is a verified result.

The proof of the lower bound is analogous, except that the calculation is carried out in downwards rounding. Therefore for the lower bound, $\alpha_i, \beta_i <= 0$. It is also possible to calculate the lower bound in upwards rounding mode by negating the

lower bounds of the summands and *subtracting* them from the partial sum s_i. Accordingly, the error term is calculated as $e_i = (s_{i-1} - s_i) - y_i$ instead of $e_i = (s_{i-1} - s_i) + y_i$.

Because the rounding errors α_i are the result of the addition $\overline{X_i} - \beta_{i-1}$, the rounding error for each iteration will be $\alpha_i \leq max(\overline{X_i}, \beta_{i-1})$. By contrast, the rounding error for standard summation is $\beta_i \leq max(\overline{X_i}, s_i)$. Therefore, $\alpha_i \leq \beta_i$, which means that for summands of one sign, compensated summation is guaranteed to be no less accurate than standard summation. Experimental results suggest that the compensated sum is orders of magnitude more accurate for some problem types (see § 4.2.6).

4.2.5.2 Implementation

The floating-point code for this method is as follows:

```
function system_potential_comp(r, q)
  real(8), intent(in) :: r(:, :), q(:)
  real(8) :: system_potential_comp, Eij
  real(8) :: y, e, temp

  system_potential_comp = 0d0
  e = 0d0
  do i = 1, size(q)
     do j = i+1, size(q)
        Eij = interaction_potential(r(:,i), r(:,j), q(i), q(j))
        temp = system_potential_comp
        y = Eij + e
        system_potential_comp = temp + y
        e = (temp - system_potential_comp) + y
     end do
  end do
end function system_potential_comp
```

Because the interval endpoints are handled separately, it is worth also showing here the interval code:

```
function system_potential_comp(r, q)
  real(8), intent(in) :: r(:, :), q(:)
  interval(8) :: system_potential_comp, Eij
  real(8) :: ss, ys, es, ts, si, yi, ei, ti

  si = -0d0
  ss = 0d0
  ei = 0d0
  es = 0d0
  do i = 1, size(q)
     do j = i+1, size(q)
```

```
        Eij = interaction_potential(r(:,i), r(:,j), q(i), q(j))
        call round_up
        yi = inf(Eij) - ei
        ys = sup(Eij) + es
        ti = si
        ts = ss
        si = si - yi
        ss = ss + ys
        ei = (ti - si) - yi
        es = (ts - ss) + ys
        call round_nearest
     end do
  end do
  system_potential_comp = interval(-si, ss)
end function system_potential_comp
```

4.2.6 Results

The potential energies for systems of various numbers n of identical particles were evaluated. All particles were located within a box of side length 3; all particles were given an equal charge of $1/n$. This 'charge normalisation' was done to ensure that the energy was constant for different numbers of particles, as it depends only on the spatial extent and total charge of the system.

Particles were distributed randomly within the box. It is useful to consider how this distribution affects the number and strength of interactions of varying distances. For an infinite system of random distribution (such as the distribution of matter in the universe), the number of particles within a (thin) spherical shell of width t around radius r will be proportional to r^2, as the volume of the shell is $4\pi r^2 t$. The potential due to these particles is proportional to r. Therefore the potential due to particles at radius r will be proportional to r; in simple terms, the total potential due to distant interactions is greater than that of nearer interactions. For a system of finite extent, the total potential does not continue to increase with distance, but must reach a maximum value, because there are simply no interactions greater than a certain distance within the system. For the cube with side-length 3 in this example, the total potential due to interactions at distance r will increase from zero at $r = 0$, reaching a maximum value at around $r = 1.5$[3], and dropping again to zero at $r = \sqrt{27}$.

Table 4.1 shows the *relative interval width* for each of the four summation methods. The relative interval width is simply the width divided by the midpoint. This is analogous to a floating point relative error which is the error divided by the magnitude of the actual result. It is intended to provide a scale-free measure of the uncertainty of the interval result. For example, a relative interval width of 4.33×10^{-11} indicates that the interval endpoints agree to at least 10 decimal digits.

[3] the radius of the largest sphere completely contained by the box

	Relative Interval Width			
Number of particles	Standard	Partially-ordered	Accumulator	Compensated
2^{10}	4.33×10^{-11}	2.62×10^{-12}	2.53×10^{-15}	8.97×10^{-16}
2^{11}	1.74×10^{-10}	5.21×10^{-12}	3.23×10^{-15}	7.36×10^{-15}
2^{12}	6.95×10^{-10}	1.04×10^{-11}	3.59×10^{-15}	7.36×10^{-16}
2^{13}	2.78×10^{-9}	2.09×10^{-11}	3.57×10^{-15}	7.34×10^{-16}
2^{14}	1.11×10^{-8}	4.17×10^{-11}	4.12×10^{-15}	7.36×10^{-16}
2^{15}	4.45×10^{-8}	8.34×10^{-11}	4.47×10^{-15}	7.67×10^{-16}
2^{16}	1.78×10^{-7}	1.67×10^{-10}	4.82×10^{-15}	7.67×10^{-16}
2^{17}	7.12×10^{-7}	3.34×10^{-10}	5.52×10^{-15}	9.25×10^{-16}

Table 4.1: Relative interval widths for the potential energy of a system of particles, calculated using four summation methods, using double-precision interval arithmetic

For standard summation, the interval relative error increases with system size. This was expected because the number of additions, and therefore, the number of potential rounding errors, increases as n^2. However, the interval results even for a small system of 2^{10} particles is still very uncertain - the lower and upper bounds differ in the eleventh decimal digit, whereas machine precision is of the order of 15 decimal digits.

Partial ordering significantly reduces the width of the interval results, as well as reducing the rate at which the intervals expand with increasing system size.

Both accumulator and compensated summation methods gave interval widths that are near machine precision.

The relative interval widths from each method are plotted in Figure 4.4.

Before conducting this experiment, it was expected that the accumulator method would give the most accurate results, and therefore the narrowest intervals. However, the widths of the intervals obtained by the accumulator method are in fact 3–5 times the widths obtained by the compensated sum. This is probably due to the increased number of additions that are required for the accumulator method ($1.5n^2$ instead of simply n^2).

The compensated sum method gave the narrowest width intervals, which means that the known maximum error for the compensated sum is smaller than the known maximum error for the other methods. However, it should be emphasised that all the interval results contain the correct result; the compensated sum is only the least *uncertain* of the results.

There is another metric that can determine whether one method is *certainly* more accurate than another. Because each of the four methods is mathematically equivalent to standard summation, all of the interval results must bound the correct result. Consequently, the intersection of all interval results must bound the correct result. Suppose there is an interval I which is the intersection of all interval results, and a

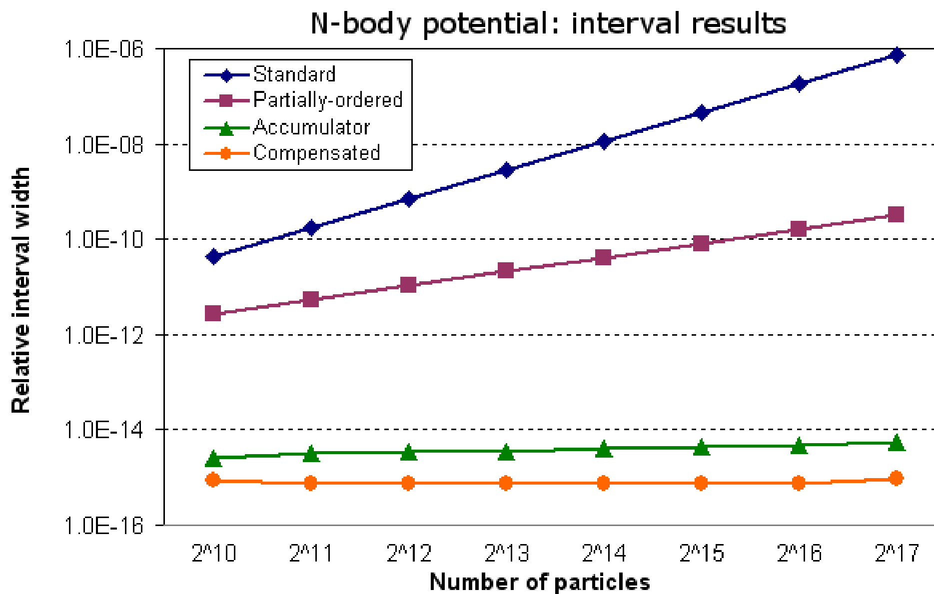


Figure 4.4: Potential energy: interval widths from four methods of calculation

floating-point value f which is the result from a particular method using floating-point arithmetic. If I does not contain f, then the floating-point result must be wrong by at least $min(abs(f - \underline{I}), abs(f - \overline{I}))$. This metric shall be denoted *known error*, and is equal to zero where the floating-point result falls within all equivalent interval results. The relative known error for each method is plotted in Figure 4.5. The known error for compensated summation was always zero, therefore it is not shown.

4.2.7 Qualifications on the results

In the codes given above, the location and charge of each particle was taken to be exact, i.e. these values were represented by real numbers rather than intervals. In a practical application it may be necessary to represent uncertainty in physical values by representing them as intervals.

In this case, the interval results reported are likely to be pessimistic; that is, the floating-point results for each of the methods are likely to be much more accurate than the width of the corresponding intervals would suggest. The reason for this is the dependency problem, discussed in § 2.3.4. The potential energy is the sum of $\frac{n^2}{2}$ pairwise interactions that are dependent on only $4n$ independent variables (n charges and $3n$ position variables). Even though the pairwise interactions share dependencies, they are treated as independent variables in this naïve interval analysis. For example, representing the x coordinate of a given particle as an interval means that this coordinate is taken to vary independently in each of the n interactions involving the particle.

To avoid the dependency problem, it is necessary to find a function for the potential energy in which each independent variable appears only once. It is not clear how

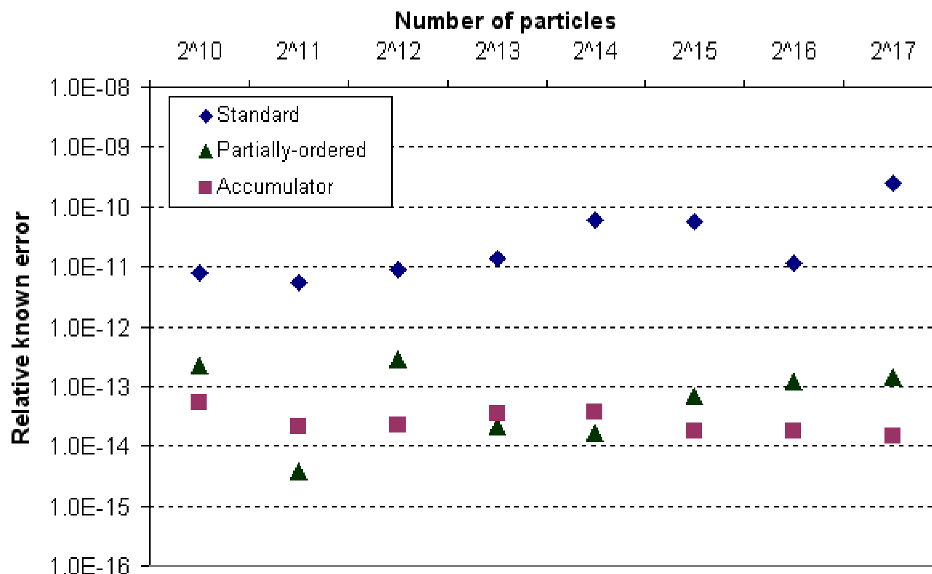


Figure 4.5: Potential energy: relative known errors from three methods of calculation

the problem can be avoided for systems of this type; however, the dependencies may be somewhat reduced in the fast multipole method (see § 4.3).

While the results from the compensated summation method are very good for this example, the method may be far less accurate for certain 'pathological' inputs. For example, where the summands differ greatly in magnitude and are ordered so as to oscillate between summands of great and small magnitude, compensation will fail to improve the result over standard summation. For example, take the following sequence of ten summands: $\{1.0 \times 10^0, 1.0 \times 10^{-16}, 1.0 \times 10^0, 1.0 \times 10^{-16}, -1.0 \times 10^0, 1.0 \times 10^{-16}, 1.0 \times 10^0, 1.0 \times 10^{-16}, -1.0 \times 10^0, 1.0 \times 10^{-16}\}$. Compensated summation will fail to give the correct result in this case because the compensation terms e_i are below machine precision compared to the summands x_i. However, for this sequence of summands, the accumulator method will give a more accurate result.

4.2.8 Benefits of interval analysis

In this section, it has been shown that interval analysis allows the calculation of guaranteed bounds on the rounding error in a floating-point calculation. These bounds were obtained without the need to construct an error model specific to the problem, and were calculated at the same time as the result itself. If the bounds are narrow enough for a given application, then the calculation using intervals allows a confidence in the result that is not possible with normal floating-point computation.

From the error bounds, it was possible to ascertain that one method of calculation – using compensated summation – was more accurate than other methods. That is, it is known for certain that some of the floating-point results are wrong by *at least* a certain amount. Again, this is information that is not available through normal floating-point

computation.

Another feature of interval analysis is that for any mathematically equivalent algorithm to exactly evaluate a given function, the corresponding interval result must contain the correct value. In this example, the four mathematically equivalent algorithms generated different intervals for the potential energy, but each interval contains the correct result. This flexibility in choice of algorithm can sometimes be useful, because the *intersection* of the various interval results may be taken to give a narrower final interval. In comparison, there is no statistically valid way of combining varying floating-point results for the same problem; neither mean, median, maximum or minimum of a set of results have any a priori mathematical significance.

4.3 Fast multipole method

In the previous section, it was shown that it is possible to calculate the potential of a system of charged particles, accurate to near machine precision. The cost of this evaluation is $\mathcal{O}(n^2)$. This computational complexity means that as system size increases, direct evaluation quickly becomes infeasible.

To address this problem, in 1987 Greengard & Rokhlin [23] proposed the *fast multipole method* (FMM) for pairwise interactions. This method approximates a pairwise interaction to an arbitrary level of accuracy, but scales as $\mathcal{O}(n)$. Thus it is possible to use this method to evaluate potentials for a much larger system than is feasible using direct evaluation.

The FMM introduces an error by truncation of infinite series. One of the attractive features of the method is that it permits the calculation of *a priori* error bounds on this truncation error. However, this bound does not include rounding errors, and little effort has previously been made to determine the effect of rounding on the FMM. The magnitude of each type of error is determined by a number of different parameters to the method. The purpose of the work presented here is to investigate the balance between rounding and truncation error in the FMM, due to the different parameters.

4.3.1 Overview of fast multipole method

This section presents a brief overview of the FMM for the calculation of electrostatic potentials. For a more general and complete introduction, see Beatson & Greengard's course notes [10].

4.3.1.1 Motivation

It is recalled from § 4.2 that the potential energy of a charge q_j due to a set of charges q_i at locations $\boldsymbol{R}_i$ is given by

$$U_j = \sum_{\substack{i=1 \\ i \neq j}}^{n} \frac{k q_i q_j}{\|\boldsymbol{R}_j - \boldsymbol{R}_i\|}$$

Because the potential at a given location depends on each of the n particles, direct evaluation of the potential at the location of each particle is an $\mathcal{O}(n^2)$ operation. This would seem to be a barrier to increasing the size of simulations. However, the result may only be required to be calculated to a specified level of accuracy. There are some features of the problem that allow approximations to be made, to reduce the computational complexity.

The strength of interaction is inversely related to the distance between the two particles, therefore the contribution of distant particles to the potential at a point is smaller than the contribution of near particles. Therefore it may be possible to ignore the effect of a small number of distant particles, where their combined contribution is less that the allowable error. This is done, for example, in molecular dynamics codes, where near interactions are evaluated frequently, and distant interactions are evaluated less frequently (or not at all). [61]

It is not always possible, however, to ignore distant particles. An alternative approach is to approximate a group of particles as a single 'virtual particle' located at the charge centre of the group. This approximation is valid as long as the group is sufficiently distant from the point of evaluation. This kind of approximation is used routinely in celestial mechanics; it is far more efficient to treat each planet or star as a point mass than to calculate the gravitational potential due to its component atoms! The fast multipole method approximates a group of distant particles, not as a 'virtual particle', but as a multipole expansion (§ 4.3.1.3) centred at a distant point.

4.3.1.2 Hierarchical space

The first key to all fast multipole methods is that space is divided in a hierarchical manner. The simplest method of division is bisection along each of the axes: thus one-dimensional space is divided in a binary tree, 2D space in an quadtree, 3D space in an octree and so on.

Because the particle system is a three-dimensional problem, this discussion will follow the 3D case. Space is divided into a tree of levels $0, 1, 2, \ldots$ where each level i is formed by dividing each of the boxes of level $i - 1$ into eight smaller boxes (the *children*), as shown in Figure 4.6.

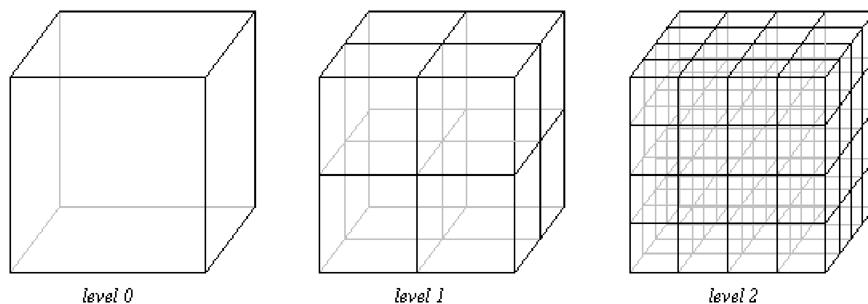


Figure 4.6: FMM: hierarchical division of space

The division is performed so as to yield a chosen average number of particles per box at the lowest level. (It is possible to use an adaptive scheme in which only boxes with more than a certain number of particles are subdivided; this improves performance for systems of nonuniform distribution.) With space divided in this manner, it is possible to distinguish several relationships between the boxes at a given level.

- Two boxes are said to be *near neighbours* if they share a vertex, and a box is a near neighbour of itself. Thus each box has at most 27 near neighbours.
- Two boxes are said to be *well separated* if smallest distance between them is at least a certain integer multiple $ws > 0$ of the side length of the boxes. Thus for $ws = 1$, well-separated boxes must have at least one box between them; for $ws = 2$ there must be at least boxes, and so on.
- For a given box, there is an *interaction list*, which is the set of all boxes that are well separated from the box, and are children of the near neighbours of the box's parent.

Figure 4.7 shows the relationships between boxes for $ws = 1$ (in two dimensions to simplify the diagram). For the box marked with a cross, the light-shaded boxes are near neighbours; the dark shaded boxes are well-separated from it.

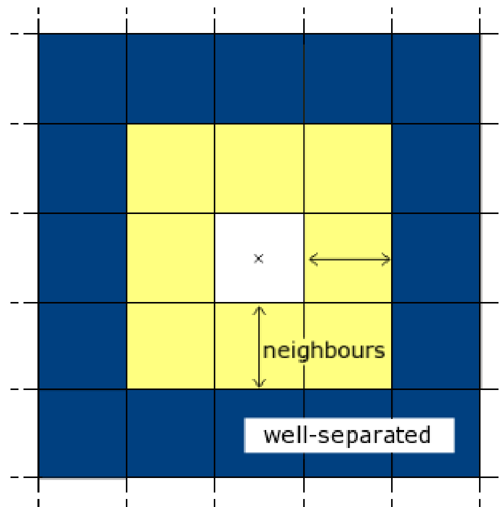


Figure 4.7: FMM: spatial relationships

4.3.1.3 Multipole expansions

The second key to the FMM is the use of multipole expansions to approximate the influence of distant groups of particles. Multipoles are used as a basis for representing the asymmetry in charge distribution for a group of particles within a sphere centred at a given point. The first term in the expansion is a *monopole*, and represents the

average charge of the particles; the second term is a dipole, which represents the axial asymmetry of the charge distribution; and so on. A pictorial representation is given in Figure 4.8.

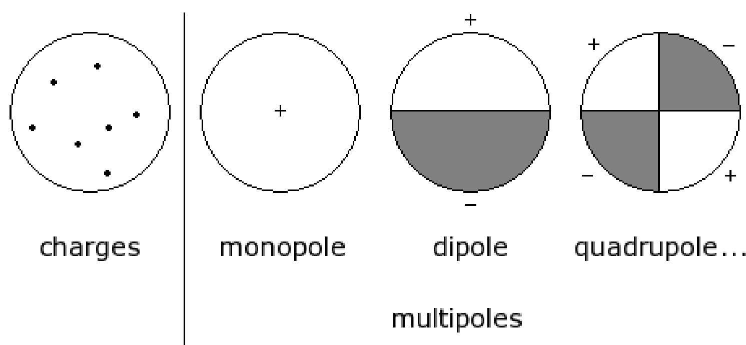


Figure 4.8: FMM: series of multipole expansions

The electrostatic potential Φ is one of a class of functions known as *potential* or *harmonic* functions (§ B.1). The standard solution of this equation in spherical coordinates yields an infinite series of *spherical harmonics* or *multipoles*. These may be used to form *multipole expansions* (§ B.2) of the potential due to distant charges.

For the potential at r due to a set of point charges located within a radius $a < r$ from the origin, the series of multipoles will converge quickly. Thus the series may be truncated at a finite number of terms p to yield an approximation which is more accurate as the number of terms increases (§ B.7.1).

4.3.1.4 $\mathcal{O}(n \log n)$ method for potential evaluation

Using multipole expansions, it is possible to construct a scheme to evaluate the potential of a system of n particles in only $\mathcal{O}(n \log n)$ time. This scheme is as follows:

1. Divide space into an octree so that on average there are n_0 particles per box at the lowest level. The number of levels required is $s = \lceil \log_8 \frac{n}{n_0} \rceil$

2. Construct multipole expansions for each box in the tree. There are $\frac{n}{n_0}$ boxes at the lowest level, thus this step takes $\mathcal{O}(n)$ time.

3. Recurse through the levels of the tree from lowest to highest. At every level, evaluate the far-field potential for each particle by adding the multipole expansions for each box that is well separated from the particle. Do not add multipole expansions for regions that have previously been covered at lower levels. The amount of work required per level is $\mathcal{O}(n)$; there are $\mathcal{O}(\log n)$ levels, thus the total scaling for this step is $\mathcal{O}(n \log n)$.

4. For all interactions between particles in boxes that are not well-separated at the lowest level, evaluate the potential directly (rather than by using multipole ex-

pansions). The number of direct evaluations will be roughly constant because the number of non-well-separated boxes at the lowest level remains constant.

The proper fast multipole method is of $\mathcal{O}(n)$ cost. To achieve this requires some additional tools.

4.3.1.5 Translation of multipole expansions

The centre of a multipole expansion may be shifted using a translation operator (§ B.3). Because potential functions may be added, lower-level multipole expansions may be formed as follows:

1. For each of the eight children of a box, shift the centre of the child's multipole expansion to the centre of the parent box.

2. Sum the shifted child expansions to form the expansion for the parent.

The 2-dimensional diagram in Figure 4.9 shows the combination of four small expansions (the small circles) to form an expansion for the parent box (the large circle).

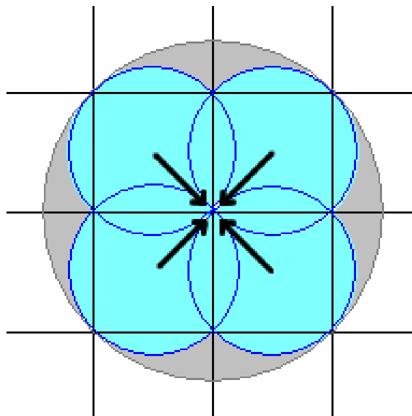


Figure 4.9: FMM: combination of multipole expansions to form expansion for parent box

4.3.1.6 Local expansions

A local expansion gives the potential at a point near the origin due to a set of distant charges. For any point r within a sphere of radius a centred at the origin, a local expansion may be constructed to describe the influence of particles that are further than $2a$ from the origin.

The expansion is a Taylor series; it may be truncated at a finite number of terms p to approximate the potential to a given level of accuracy.

A multipole expansion for a set of particles may be converted into a local expansion around a given point (§ B.4); this conversion is shown in Figure 4.10. This operation introduces an error due to the truncation of the Taylor series at p terms (§ B.7.2).

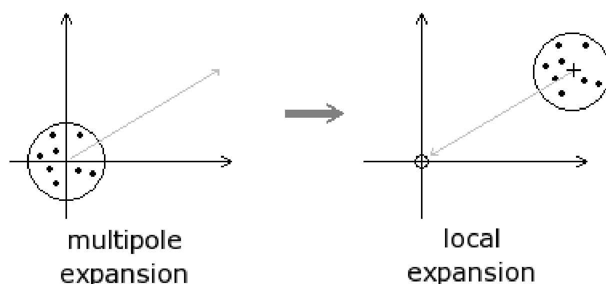


Figure 4.10: FMM: transformation of a multipole to a local expansion

Local expansions, like multipole expansions, may be summed if they share a common centre. They may also be translated (§ B.5).

4.3.1.7 $\mathcal{O}(n)$ method for potential evaluation

With the set of tools described in the preceding sections, a method can be constructed to evaluate the potential of a system in $\mathcal{O}(n)$ time. The method is divided into two passes.

Upward pass

- Construct multipole expansions for boxes at the lowest level. (§ B.2)
- Moving upwards through the tree, combine expansions from child boxes to form an expansion for the parents at each level. Translate the eight child expansions to the centre of the parent box (§ B.3) and then sum them to form the parent expansion.

Downward pass

- At each level, convert the multipole expansion for each box to local expansions for all boxes on its interaction list. (§ B.4)
- Transmit local expansion information downwards by translating a parent expansion (§ B.5) to the centre of each of its eight child boxes. The local expansion for each box is the sum of the local expansion of its parent and the expansions due to boxes on its interaction list.

After the downwards pass is complete, there is a local expansion for each box at the lowest level. Each particle within a box is interacted with the local expansion to give that particle's potential due to particles in all well-separated boxes. Finally, near-field interactions are calculated directly and added to the far-field potential.

4.3.2 Interval FMM implementation

A working FMM code was obtained from Rui Yang [63] that follows the implementation given by White and Head-Gordon [61]. To aid experimentation, the code was modified to accept the following values as parameters:

- p: the number of terms at which to truncate the multipole and local expansions;
- ws: the number of box widths between well separated boxes; and
- n_0: the number of particles per lowest level box.

A naïve interval implementation was constructed by replacing some real values with intervals. The full code is given in Appendix C.

Two 'virtual types' were created called `quantity` and `complexquantity`; preprocessor declarations were used to replace these strings with either real or interval types, depending on whether a floating-point or interval version of the code was desired. The `quantity` type was used to represent physical quantities (distance, potential etc.) or values dependent on these quantities (coefficients of multipole expansions, Legendre polynomials etc.). The `complexquantity` type was used to represent complex numbers that are dependent on physical quantities (i.e. the spherical harmonics in multipole and local expansions).

The SunStudio compilers do not support a complex interval type. Therefore it was necessary to create such a type. A cartesian representation was chosen in which each complex interval is rectangular with the axes of the complex plane. This choice of representation means that, while addition yields an acceptably narrow interval result, the multiplication and division operations tend to expand the interval width much more than addition. [50] This is known as 'wrapping' and is due to the fact that each component of the result is dependent on both the real and complex components of each operand. An informative visual demonstration of this effect is given by Miehe et al. [43] An alternative representation for complex intervals uses polar coordinates; A.4 this reduces the wrapping effect for multiplication but still suffers wrapping in division.

The definition of the complex interval type is simple enough: represent both real and imaginary parts as intervals.

```
type icomplex
   interval(8) :: rl, im
end type icomplex
```

However, to use this type in the FMM requires the definition of a large set of functions. Two examples are given here; the full set is given in Appendix C.

```
elemental function add(a, b)
  type(icomplex), intent(in) :: a, b
  type(icomplex) :: add
```

```
  add%rl = a%rl + b%rl
  add%im = a%im + b%im
end function add

elemental function multiply_yy(a, b)
  type(icomplex), intent(in) :: a, b
  type(icomplex) :: multiply_yy

  multiply_yy%rl = a%rl * b%rl - a%im * b%im
  multiply_yy%im = a%im * b%rl + a%rl * b%im
end function multiply_yy
```

The code was used to evaluate the potentials for the same systems used in § 4.2. The results from the FMM were compared with the results from direct evaluation using compensated summation (the best known method of direct evaluation).

4.3.2.1 Results

The series of graphs in Figure 4.11 shows the interval widths obtained from the fast multipole method against the result from direct evaluation using compensated summation, for a system of 8192 particles with a well-separatedness of 1, 2 and 3 respectively. There were four tiers in the tree, yielding on average $n_0 = 4$ particles per box at the lowest level.

The interval results display two key features. The first is that the truncation error reduces when either the number of expansion terms p or the 'well-separatedness' ws is increased. This is the same behaviour as seen for the floating-point results. The second feature is that the interval width increases as the number of terms is increased.

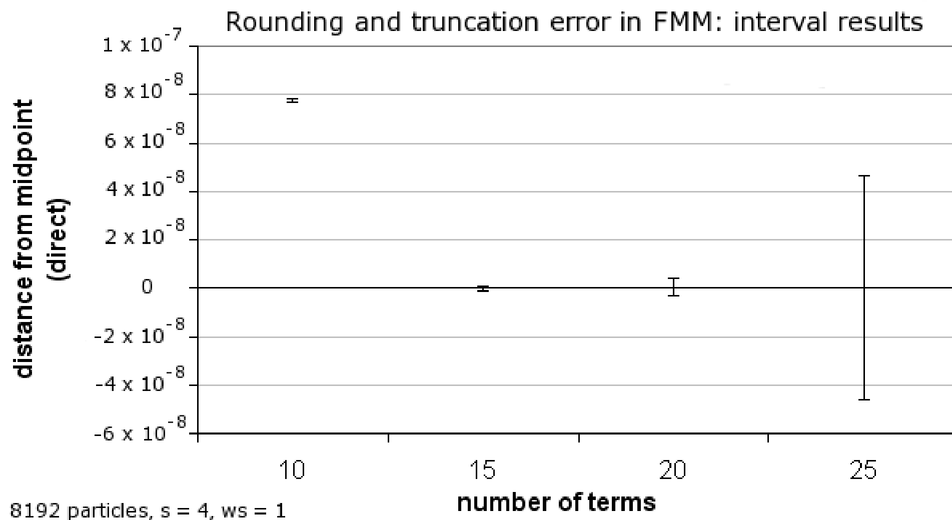
The *known* truncation error, as measured by the distance by which the FMM interval result falls outside the interval result from direct evaluation using compensated summation (see § 4.2.5), decreases rapidly with an increased number of terms, or if the well-separatedness parameter ws is increased from 1 to 2. Above a certain number of terms, the truncation error is zero because the interval result from the FMM completely contains the interval result from direct evaluation.

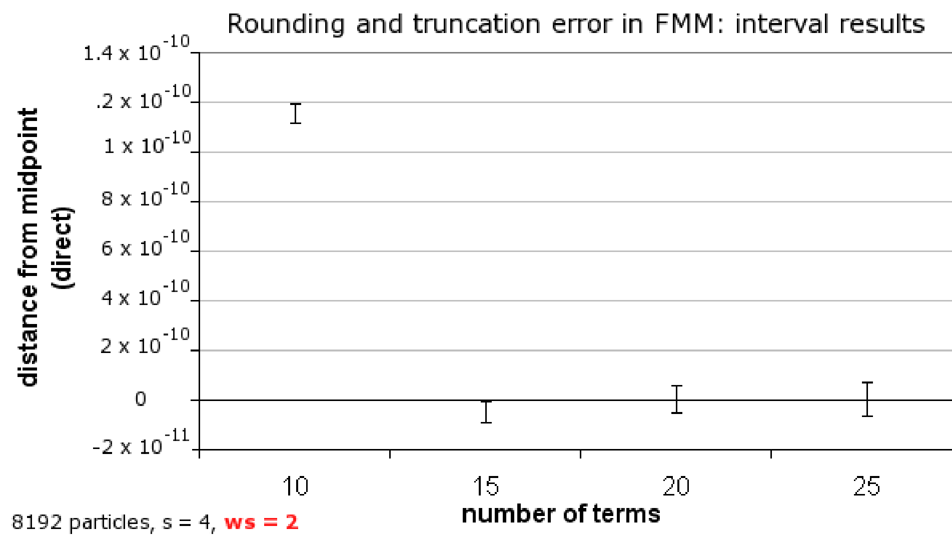
The interval width increases as the number of terms in the multipole and local expansions is increased. If the known truncation error is already zero, then increasing the number of terms only widens the interval width of the FMM result, thus making the result more uncertain, and no more accurate.

4.3.3 Rounding and truncation error in the FMM

Because the FMM is an approximate method, numerical analysts are naturally inclined to concentrate on the approximation errors introduced through the series truncations. Truncation error is introduced by two operations:

- the truncation of multipole expansions at p terms; and





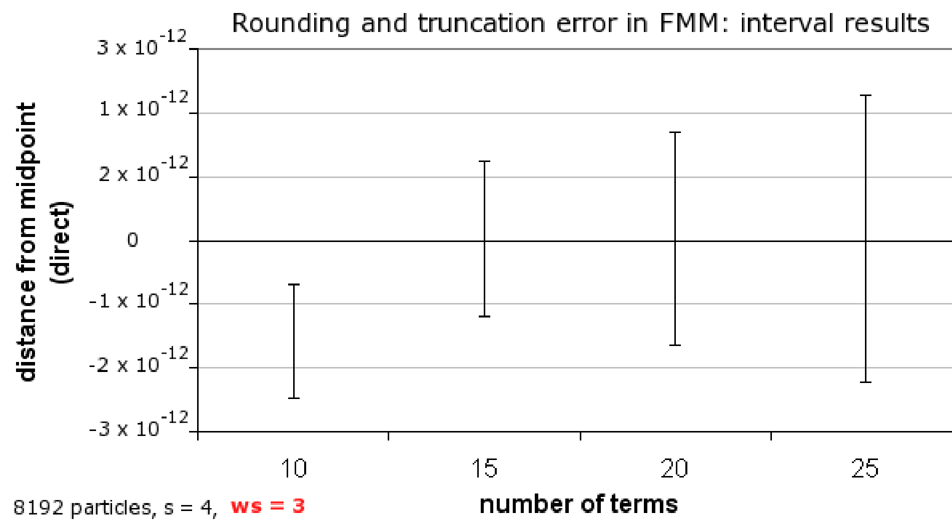


Figure 4.11: Potential energy for a system of charged particles: interval results from FMM compared to results from direct evaluation for different numbers of tiers and multipole terms

- the transformation of multipole expansions into local expansions, which are also truncated at p terms.

The second source of error does not exist for the $\mathcal{O}(n \log n)$ scheme, which does not use local expansions. This reduces the number of terms required for a given level of accuracy, at an increased computational cost. [10].

The results from the interval FMM code suggest that as the number of terms used is increased, the rounding error in the result may also increase. For small numbers of terms, this rounding error is less than the truncation error due to the approximations in the algorithm. For larger numbers of terms, the rounding error may be greater than the truncation error, meaning that adding further terms becomes detrimental to the accuracy of the method.

These results suggest that for guaranteed accuracy, increasing the well-separatedness is preferred over increasing the number of terms. The main reason for this result is probably that increasing the well-separatedness also increases the number of near interactions that are evaluated directly. The main drawback to increasing the well-separatedness is the performance penalty; a four-fold increase in cost from $ws = 1$ to $ws = 2$ and a doubling in cost from $ws = 2$ to $ws = 3$.

The effect of changing the different parameters p, ws, s on truncation and rounding error is shown qualitatively in Table 4.2. The table also includes the effect on execution time, as a reminder that reducing error must come at a cost.

	Effect of increasing parameter on		
parameter	truncation	rounding	execution time
number of terms (p)	-	+	+
well-separatedness (ws)	-	-	+
number of tiers (s)	-	+	+

Table 4.2: Truncation error, rounding error and execution time in the FMM: relationship to different parameters

4.3.4 Qualifications on these results

The interval bounds calculated by this implementation are likely to be pessimistic, particularly due to the wrapping effect on complex intervals. Therefore it is likely that more terms can be "safely" used in the FMM than are guaranteed by this analysis to increase accuracy.

Further work is required to determine whether the interval results can be made narrower while maintaining a mathematically equivalent algorithm.

All locations and charges were represented by real numbers (as opposed to intervals), that is, these values were taken to be exact. In a real-world calculation, these values are likely to be uncertain, and they may therefore be represented as intervals. Further work is needed to determine how input uncertainty propagates through the calculation.

4.3.5 Performance of the interval fast multipole method

The main motivation for using the fast multipole method is, of course, speed. It is therefore gratifying to find that the interval version of the FMM is not significantly slower than the floating-point version. The execution times for a potential calculation for a three-tier FMM are given in Table 4.3, with respect to different numbers of terms and well-separatedness. The interval calculations took 3.5–9 times longer than the floating-point calculations.

		execution time (s)	
p	ws	floating-point	interval
10	1	22.0	79.6
10	2	30.5	192
10	3	38.7	339
25	1	79.1	542
25	2	164	1280
25	3	209	1730

Table 4.3: Execution time for 3-tier FMM calculations of potential energy of a system of 16384 particles: floating-point and interval times vs. number of terms p and well-separatedness ws

These results are only for a small system, for which it is more appropriate to use direct calculation. Additionally, the floating-point code has not been heavily optimised. However, the interval code follows the floating-point code very closely (with the exception of the basic arithmetic operations and trigonometric functions), so the ratio between floating-point and interval times is what might be expected even in optimised code.

The results suggest that it may be feasible to perform interval FMM calculations for large systems. The fast performance of the interval FMM is attributable to the use of hand-optimised interval operations that were developed in Chapter 3.

4.4 Conclusions

This demonstration has shown that interval analysis enables guaranteed error bounds to be calculated, with reasonably little effort, that may be good enough for some practical applications. For real world problems, it is likely that the guaranteed bounds on rounding error will be insignificant compared to measurement error. Thus it may be possible, using intervals, to support a choice to ignore rounding error as a source of incorrect results.

The interval results obtained are usually wider than an optimal bound, because of the dependency problem. This may limit their usefulness for some applications. More effort is required to obtain narrow bounds, and the required techniques may be different for each problem.

Comparison of the results from direct and FMM evaluation suggest that “approximate” methods may in some cases be more accurate than “exact” methods, where

exact evaluation is more seriously affected by rounding error. If the approximation reduces the number of floating-point operations necessary to calculate the result, it may also reduce the number of rounding errors.

Chapter 5

Discussion

You may be dissatisfied with the approximate view of nature
that physics tries to get at.
The attempt is to ever increase the accuracy of the approximation.

Richard Feynman
Lecture on *Characteristics of Force*
Caltech, 7 November 1961

Physical discoveries in the future
are a matter of the sixth decimal place.

Albert A. Michelson
dedication of the Ryerson Physical Laboratory,
University of Chicago, 1884

The work presented in this thesis was focused on two potential problems that may limit the feasibility of applying interval analysis in scientific computation. The first is the performance of interval computation relative to floating-point computation. The second is the extent to which the interval results are actually useful, in the sense that they provide information about the uncertainty of results that could not easily be obtained by another method. These problems will be discussed in turn.

5.1 Performance of interval computation

Most software support for interval computation is in the form of libraries: all interval operations are compiled to library calls. The benchmark results in Chapter 3 show that such an implementation is unlikely to result in optimal performance. A better approach is to compile the basic interval arithmetic operations to inline assembly code. For this reason, the implementers of filib also provide a cut-down macro version in which interval macros are translated into sequences of inline C++ statements by the preprocessor. [37]

The best performance, however, is likely to be achieved through intrinsic compiler support for interval types. Schulte et al. [51] have demonstrated that an interval-enhanced GNU Fortran compiler enables great performance improvements over interval library support. Sun's implementation in the SunStudio suite is well-placed to

move from its current approach (compiling to assembly procedure calls) to compiling interval operations to inline assembly code.

Recently, many researchers have contributed towards the development of interval support in the popular Boost C++ library (see § A.3). While the performance of this implementation is not likely to be optimal, the intention is to provide a 'reference standard' for the implementation of intervals. The availability of a standard – de facto or otherwise – would greatly assist the accessibility of interval computation to a wider audience. Accepted standards should be incorporated into compiler support in gcc and other compilers.

Interval computation is currently expensive in comparison with floating-point computation, largely because the interval community is still commercially and culturally in a fledgling stage. Hardware and software vendors (including the open source community) are unlikely to devote much effort to improve performance of interval computation without significant demand from a broad user community. At the same time, most users will find it impractical to apply interval computation until performance improves. Walster [57] refers to this as the *commercial funding feedback loop*.

One element that is needed to close the loop is the development of 'killer interval apps': interval analysis applied to find solutions to problems that cannot be successfully approached in other ways. A small number of these have already been demonstrated for global optimization problems (see § 2.3.5.1), however, this is only a small area of computing. A much broader use of interval techniques is in error analysis and calculation of verified results, as demonstrated in Chapter 4. To significantly increase the interval user community, more demonstrations of this kind will be required.

5.2 Usefulness of interval results

The results in Chapter 4 demonstrate that interval analysis can be used to quantify the rounding error in a given computation. It is easy to perform naïve interval analysis by replacing real numbers with intervals in the code. Running the code for certain inputs gives some indication of the stability of the algorithm for those inputs: if the interval widths are acceptably narrow, then the rounding error is verified to be within acceptable bounds.

It is also possible to perform a kind of perturbation analysis by representing inputs as intervals of non-zero width. By examining the effect of different input widths on the width of the result, it is possible to explore the extent to which uncertainty in each of the inputs influences the uncertainty in the final result.

However, the fact that an interval result width is large doesn't necessarily mean that the algorithm is numerically unstable. In particular, where an algorithm is affected by the dependency problem (see § 2.3.4), the interval result will be much wider than the possible range of the corresponding floating-point result. In some cases, it may be possible to manually rearrange the algorithm to reduce dependencies. In other cases (for example the direct evaluation of potentials discussed in § 4.2), such rearrangement may not be possible.

A special case of the dependency problem involves the wrapping effect in calculations involving multiplication and division of complex intervals. Relatively little has been published on this problem, but it may be a serious barrier to achieving narrow width intervals in many scientific codes. One suggestion for reducing wrapping is to represent complex numbers as discs, rather than rectangles, in the complex plane (see § A.4). It is not apparent that this representation is appropriate for all complex values. It may be necessary to analyse each complex quantity to determine which representation is more appropriate; choosing a representation that does not correspond to the underlying physics of a given quantity may invalidate the interval results. It is clear that more work is needed on complex interval analysis before it will gain widespread acceptance as a tool for reliable computing.

Chapter 6

Conclusion

> If you've made up your mind to test a theory, or you want to explain some idea, you should always decide to publish it whichever way it comes out. If we only publish results of a certain kind, we can make the argument look good. We must publish *both* kinds of results.
>
> **Richard Feynman**
> "Cargo Cult Science"
> Caltech Commencement Address 1974

The primary aim of this work was to demonstrate interval techniques that can feasibly be applied to scientific computing; this aim has been achieved. The results for the example application in Chapter 4 provide useful information about rounding error in a variety of computational methods, that could not have easily been obtained in another way.

6.1 Contributions

In order to support this demonstration, some new tools and algorithms were developed.

- A collection of hand-optimised interval subroutines was created and used in the benchamrks and example applications. The techniques used in the creation of these subroutines indicate that it should be possible for Sun to greatly improve the performance of interval code compiled with the SunStudio compilers. (Chapter 3)

- A faster method of interval multiplication was proposed for the UltraSPARC version of the SunStudio compilers, based on conditional move statements. This method is approximately 30% faster than the branching method currently used by the SunStudio compilers. (§ 3.4.6.2)

- Additional instructions were proposed for UltraSPARC and similar architectures that could further improve the performance of interval multiplication. (§ 3.4.7)

- A method of accurate summation using accumulators was developed. This method enables the summation of a set of floating-point or interval values to an accuracy near machine precision. (§ 4.2.4)
- The floating-point compensated sum was extended to intervals. It was proved that the interval results from this method always enclose the correct result. It was also proved that the interval widths from this method are no wider than those from standard summation. Experimental results show that this method greatly reduces the interval widths in typical applications. (§ 4.2.5)
- An interval Fast Multipole Method was implemented for potential evaluations. This is believed to be the first time interval analysis has been used in conjunction with this method. The interval results from this implementation were used to explore the balance between rounding and truncation error in the method. It was concluded that, as rounding error increases with the number of terms used in the method, there may be limits on the accuracy that can be achieved with this method. (§ 4.3)

6.2 Further work

6.2.1 Interval tools

The results for the UltraSPARC show that interval multiplication is faster (and more predictable) if it is implemented using conditional move statements, rather than using branch statements. Interval arithmetic implementations for other in-order architectures should be reevaluated in the light of these results.

The accumulator sum was shown to greatly increase the accuracy of summation in experimental results. A rigorous numerical analysis of this method has not been performed; such an analysis would aim to give an a priori bound on the error from this method, given certain properties of the summands.

The value of the interval compensated sum has been demonstrated. It may be possible to create interval versions of other algorithms for accurate operations, such as the *accurate sum and dot product* due to Ogita, Rump and Oishi [49].

6.2.2 Fast multipole method

Possible extensions to the error analysis of the Fast Multipole Method are almost unlimited.

Many improvements to the algorithm analysed in Chapter 4 are possible. Greengard and Rokhlin [24] convert the multipole expansion into six directional plane-wave expansion, which allows the use of diagonal (as opposed to spherical) translation operators, thereby reducing the cost of translation from $\mathcal{O}(p^4)$ to $\mathcal{O}(p^3)$ for a multipole expansion of p terms. Cheng, Greengard and Rokhlin [13] give an adaptive algorithm which does not further subdivide boxes at a given level that contain less than a certain number of charges. Both these techniques are used to improve performance, but

it would be interesting to see what effect they have on the balance between truncation and rounding error.

The use of calculations on complex intervals needs to be investigated more closely, to see if this use causes wrapping in the interval FMM results. It may be possible to use an alternative representation, such as complex discs, to reduce wrapping.

6.3 Conclusion

The time is ripe for the wider application of interval methods to scientific computing. While there are still some issues around computational performance and the ease with which interval analysis can be usefully applied, these issues can be overcome. The work presented in this thesis goes some way towards this end.

Walster [57] predicts a paradigm shift in computing that will see interval methods become the standard in many fields. If he is right, we may hope that the embarrassments and disasters caused by floating-point error will become a thing of the past.

Appendix A

Software tools for interval computation

Several interval tools were mentioned in this thesis; these are listed below, with text and/or website references. A comprehensive list of interval and related software is maintained by Vladik Kreinovich at
http://www.cs.utep.edu/interval-comp/intsoft.html.

A.1 Sun compilers

Sun Microsystems includes support for interval arithmetic in its C++ and Fortran compilers in the SunStudio [4; 5]. More information on Sun's interval support can be found at
http://developers.sun.com/

A.2 filib

Von Gudenberg et al. at the University of Wuppertal have created the C++ interval library filib[38]. Krämer and Bantle[35] describe how forward error analysis was used to improve the efficiency of evaluation of interval functions.

The filib source code and documentation is maintained by Jürgen Wolff von Gudenberg at
http://www2.informatik.uni-wuerzburg.de/staff/wvg/Public/.

A.3 Boost interval arithmetic library

The popular free Boost library project released an interval library in late 2004. Work is currently underway to have the boost implementation accepted as a standard in the C++ Standards Committee's Library Technical Report. The library is maintained with documentation at
http://www.boost.org/libs/numeric/interval/

A.4 Interval arithmetic in Maple

Markus Grimmer has implemented a interval arithmetic package for Maple based on previous work by Corless, Connell, Geulig and Krämer. The package includes an implementation of complex interval arithmetic using the complex disc representation. It can be downloaded from `http://www.maplesoft.com/`.

Appendix B

Mathematics for the fast multipole method

In fact, the glory of mathematics is that
you don't know what you're talking about.

Richard Feynman
Lecture on *Characteristics of Force*
Caltech, 7 November 1961

This section details the mathematics of the fast multipole method for the Laplace equation used in Chapter 4. Some knowledge of the expansions and operators of the method is useful in understanding the relationship between truncation and rounding error in FMM calculations. The discussion draws heavily on Cheng, Greengard and Rokhlin [13] and Beatson and Greengard [10].

B.1 Potential functions

Potential or *harmonic* functions satisfy the Laplace equation

$$\nabla^2 \Phi = \frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} + \frac{\partial^2 \Phi}{\partial z^2} = 0 \tag{B.1}$$

This may be written in spherical coordinates r, θ, ϕ as

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2 \frac{\partial \Phi}{\partial r}\right) + \frac{1}{r^2 \sin\theta}\frac{\partial}{\partial \theta}\left(\sin\theta \frac{\partial \Phi}{\partial \theta}\right) + \frac{1}{r^2 \sin^2\theta}\frac{\partial^2 \Phi}{\partial \phi^2} = 0 \tag{B.2}$$

The standard solution by separation of variables yields a series

$$\Phi = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \left(L_n^m r^n + \frac{M_n^m}{r^{n+1}} \right) Y_n^m(\theta, \phi) \tag{B.3}$$

in which $r^n \cdot Y_n^m(\theta, \phi)$ are the spherical harmonics of degree n and L_n^m, M_n^m are the moments of expansion. The spherical harmonics are defined in terms of the associated

Legendre functions P_n^m (Equation B.17) as follows:

$$Y_n^m(\theta,\phi) = \sqrt{\frac{2n+1}{4\pi}\frac{(n-|m|)!}{(n+|m|)!}} \cdot P_n^{|m|}(\cos\theta)e^{im\phi} \tag{B.4}$$

B.2 Multipole and local expansions

The potential function Φ may be separated into two parts: the multipole and local expansions.

The multipole expansion gives the potential at points far from the origin, due to a set of charges within a sphere centred at the origin. Given a set of charges q_i : $i = 1, 2, \ldots, N$ located at points $\mathbf{a}_i = (\rho_i, \alpha_i, \beta_i) : \rho_i < a$, the potential at any point $\mathbf{r} = (r, \theta, \phi) : r > a$ is given by

$$\Phi(\mathbf{r}) = \sum_{n=0}^{\infty}\sum_{m=-n}^{n} \frac{M_n^m}{r^{n+1}} \cdot Y_n^m(\theta,\phi) \tag{B.5}$$

where the multipole moments of expansion are

$$M_n^m = \sum_{i=1}^{N} q_i \cdot \rho_i^n \cdot Y_n^{-m}(\alpha_i,\beta_i) \tag{B.6}$$

The local expansion is the converse of the multipole expansion; it gives the potential local to the origin, due to a set of charges located far from the origin. The potential at a point $\mathbf{r} : (r, \theta, \phi)$ within a sphere of radius a centred at the origin, due to a set of charges $q_i : i = 1, 2, \ldots, N$ located at points $(\rho_i, \alpha_i, \beta_i) : \rho_i > a$ is given by

$$\Phi(\mathbf{r}) = \sum_{j=0}^{\infty}\sum_{k=-j}^{j} L_j^k \cdot Y_j^k(\theta,\phi) \cdot r^j \tag{B.7}$$

where the local moments of expansion are

$$L_j^k = \sum_{i=1}^{N} q_i \cdot \frac{Y_j^{-k}(\alpha_i,\beta_i)}{\rho_i^{j+1}} \tag{B.8}$$

Both multipole and local expansions may be summed around a common centre.

B.3 Translation of multipole expansion

Suppose that the potential due to a set of charges located within a sphere centred at (ρ, α, β) is given by a expansion of the form of Equation B.5 with multipole moments

O_n^m. This expansion may be translated to an expansion centred at the origin:

$$\Phi(\mathbf{r}) = \sum_{j=0}^{\infty} \sum_{k=-j}^{j} \frac{M_j^k}{r^{j+1}} \cdot Y_j^k(\theta, \phi) \tag{B.9}$$

where the moments of the translated expansion are given by

$$M_j^k = \sum_{n=0}^{j} \sum_{m=-n}^{n} \frac{O_{j-n}^{k-m} \cdot i^{|k|-|m|-|k-m|} \cdot A_n^m \cdot A_{j-n}^{k-m} \cdot \rho^n \cdot Y_n^{-m}(\alpha, \beta)}{A_j^k} \tag{B.10}$$

with A_n^m defined by

$$A_n^m = \frac{(-1)^n}{\sqrt{(n-m)! \cdot (n+m)!}} \tag{B.11}$$

B.4 Transformation of multipole to local expansion

A multipole expansion for a group of distant charges may be converted to a local expansion about the origin.

Given a set of charges within a sphere of radius a centred at (ρ, α, β) where $\rho > (c+1)a : c > 1$, there is a multipole expansion with moments O_n^m that converges within a sphere of radius a centred at the origin. This may be converted to a local expansion for any point within the sphere:

$$\Phi(\mathbf{r}) = \sum_{j=0}^{\infty} \sum_{k=-j}^{j} L_j^k \cdot Y_j^k(\theta, \phi) \cdot r^j \tag{B.12}$$

where the moments of expansion L_j^k are given by

$$L_j^k = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \frac{O_n^m \cdot i^{|k-m|-|k|-|m|} \cdot A_n^m \cdot A_j^k \cdot Y_{j+n}^{m-k}(\alpha, \beta)}{(-1)^n A_{j+n}^{m-k} \cdot \rho^{j+n+1}} \tag{B.13}$$

and A_n^m etc. are defined as per Equation B.11.

B.5 Translation of local expansion

Suppose a local expansion around $\mathbf{a} : (\rho, \alpha, \beta)$ that gives the potential at a point $\mathbf{r}$ where where $\mathbf{r} - \mathbf{a} = (r', \theta', \phi')$:

$$\Phi(\mathbf{r}) = \sum_{n=0}^{p} \sum_{m=-n}^{n} O_n^m \cdot Y_n^m(\theta', \phi') \cdot r'^n \tag{B.14}$$

this may be translated to a local expansion around $\mathbf{r} : (r, \theta, \phi)$ as follows:

$$\Phi(\mathbf{r}) = \sum_{j=0}^{p} \sum_{k=-j}^{j} L_j^k \cdot Y_j^k(\theta, \phi) \cdot r^j \tag{B.15}$$

B.6 Associated Legendre functions

The Legendre polynomials $P_n(x)$ are defined by Rodrigues' formula

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} (x^2 - 1)^n \tag{B.16}$$

The associated Legendre functions are defined on these by

$$P_n^m(x) = (-1)^m (1 - x^2)^{\frac{m}{2}} \frac{d^m}{dx^m} P_n(x) \tag{B.17}$$

B.7 Truncation error bounds for the fast multipole method

The approximation errors in the FMM allow the calculation of rigorous *a priori* error bounds, dependent on the parameters to the method.

B.7.1 Error in a truncated multipole expansion

The truncation of a multipole expansion at p terms introduces an error in the potential that is bounded by:

$$\left| \Phi(\mathbf{r}) - \sum_{n=0}^{p} \sum_{m=-n}^{n} \frac{M_n^m}{r^{n+1}} \cdot Y_n^m(\theta, \phi) \right| \leq \left(\frac{Q}{r - a} \right) \left(\frac{a}{r} \right)^{p+1} \tag{B.18}$$

where Q is the total magnitude of charge:

$$Q = \sum_{i=1}^{N} |q_i| \tag{B.19}$$

B.7.2 Error due to transformation of multipole to local expansion

Equation B.12 allows the conversion of a multipole expansion centred at (ρ, α, β) where $\rho > (c+1)a : c > 1$ to a local expansion within a sphere of radius a centred at the origin. For this operation, the error due to truncation of the local expansion at p terms is bounded by:

$$\left| \Phi(\mathbf{r}) - \sum_{j=0}^{p} \sum_{k=-j}^{j} L_j^k \cdot Y_j^k(\theta, \phi) \cdot r^j \right| \leq \left(\frac{Q}{ca - a} \right) \left(\frac{1}{c} \right)^{p+1} \tag{B.20}$$

Appendix C

Source code for the interval fast multipole method

C.1 Source file: fmmgrid.F95

This is the harness used to run the FMM code for different parameters for the same system of particles, in comparison with direct evaluation.

```
program fmmgrid
  use fmm

  integer, allocatable :: terms(:), nlow(:), ws(:)
  integer :: nlevel
  character*100 :: input_buffer

  call get_params()
  call particle_setup()

  call direct_energy()

  do i = 1, size(terms)
     np = terms(i)
     do j = 1, size(nlow)
        N0 = nlow(j)
        do k = 1, size(ws)
           iws = ws(k)
           call fmm_setup(nlevel)
           call fmm3d(nlevel)
           call fmm_cleanup()
           call flush(6) ! flush standard output
        end do
     end do
  end do

  call particle_cleanup()

contains
  subroutine get_params()
    if (iargc() < 1) then
       write(*,*) 'usage: fmmgrid natom [terms] [N0] [well-spaced]'
```

```
      stop
   end if

   print *, '-------------------------------------------------------'
   print *, '             Coulomb potential calculation '
   print *, '-------------------------------------------------------'

   !     first command line argument natom = number of particles
   call getarg(1, input_buffer)
   read(input_buffer, *) natom

   print *, 'Atoms:', natom

   ! Default parameters
   allocate(ws(3))
   ws = (/ 1, 2, 3 /)
   allocate(nlow(3))
   nlow = (/ 10, 50, 100 /)
   allocate(terms(4))
   terms = (/ 10, 15, 20, 25 /)

   if (iargc() > 1) then
      ! second argument nlow = number of atoms per lowest level box
      deallocate(nlow)
      allocate(nlow(1))
      call getarg(2, input_buffer)
      read(input_buffer, *) nlow(1)
      if (iargc() > 2) then
         ! third argument terms = number of terms in multipole expansion
         deallocate(terms)
         allocate(terms(1))
         call getarg(3, input_buffer)
         read(input_buffer, *) terms(1)
         if (iargc() > 3) then
            !     fourth argument ws = well-spaced distance
            deallocate(ws)
            allocate(ws(1))
            call getarg(4, input_buffer)
            read(input_buffer, *) ws(1)
         end if
      end if
   end if

   print*, 'nlow:', nlow
   print*, 'terms:', terms
   print*, 'ws:', ws
 end subroutine get_params

end program fmmgrid
```

C.2 Source file: fmm.F95

This is the 'guts' of the code; all setup and evaluation for the FMM is implemented here.

```
module fmm
  use randomizer

#include "inttypes.F95"

  private multipole3d, translate, local, nearest, translate_expansion, &
       & Olm, Mlm, Plm, polar, box_centre, box_idx, direct_system, direct_pair

  type polar_coords
     interval(8) :: r, theta, phi
  end type

  integer, parameter :: mxlevel=6
  integer :: iws ! well separated boxes are at least [iws] boxes apart
  integer :: natom ! total number of particles
  logical(1), allocatable :: box_empty(:) ! = .false. if this box is occupied
  integer, allocatable :: nhost(:,:)
  real(8), allocatable :: achg(:), scoor(:,:) ! charges and locations
  interval(8), allocatable :: Ei_pairwise(:)
  interval(8) :: E_pairwise
  real(8) :: scale(3,mxlevel) ! box scale on each side below highest level
  real(8) :: dmax(3), dmin(3)
  real(8) :: tsta(7)
  integer :: level_idx(0:mxlevel), np, N0, mxbox, naj
  integer :: mxbox ! total number of boxes below highest level
  integer :: np ! number of terms in multipole and local expansions
  integer :: expnlen ! expansion length = (np+1)**2

contains

#include "intfuncs.F95"

  subroutine particle_setup()
    implicit none
    integer :: ia, i

    allocate(Ei_pairwise(natom))
    allocate(nhost(3,natom))
    allocate(achg(natom))
    allocate(scoor(3,natom))

    ! Set atoms' positions and charges.
    call fill_array(scoor)
    achg(:) = 1d0 / natom

    dmin = 100.0
    dmax = -100.0
    do ia = 1, natom
       dmin = min(dmin, scoor(:,ia))
       dmax = max(dmax, scoor(:,ia))
    end do

    do i=1,3
       dmin(i) = floor(dmin(i))
```

```
      dmax(i) = ceiling(dmax(i))
   end do
 end subroutine particle_setup

 subroutine particle_cleanup()
   deallocate(Ei_pairwise)
   deallocate(nhost)
   deallocate(achg)
   deallocate(scoor)
 end subroutine particle_cleanup

 subroutine fmm_setup(nlevel)
   implicit none
   integer, intent(inout) :: nlevel
   integer :: i, ma, ndxbox

   print *, '--------------------------------------------------------'
   print *, '                   Fast Multipole Method '
   print *, '--------------------------------------------------------'
   tsta(:) = 0.0d0
   call cpu_time(tsta(1))
   !     Number of levels in octree for FMM
   nlevel = int(log(real(natom/N0))/log(8d0)) + 1
   if (nlevel.lt.2)  nlevel=2
   print '(a9,i3,4x,a30,f6.1)', 'levels:', nlevel, &
     & 'atoms per box, lowest level:', natom/real(8**nlevel)

   mxbox = 8*(1-8**nlevel)/(-7)+1
   allocate(box_empty(mxbox))

   expnlen = (np+1)**2
   print '(a9,i3,4x,a30,i4)', 'terms:', np, 'well-spaced separation:', iws

   level_idx(0) = 1
   do i = 1, nlevel
      level_idx(i) = level_idx(i-1) + 8**(i-1)
      scale(:,i) = (dmax - dmin) / 2**i
   end do

   box_empty(:) = .true.
   do ma = 1, natom
      nhost(:,ma) = scoor(:,ma) / scale(:,nlevel)
      ndxbox = level_idx(nlevel) + box_idx(nhost(:,ma), nlevel)
      box_empty(ndxbox) = .false.
   end do
 end subroutine fmm_setup

 subroutine fmm_cleanup()
   implicit none
   deallocate(box_empty)
 end subroutine fmm_cleanup

 subroutine fmm3d(nlevel)
   implicit none
```

```
  integer :: nlevel
  type(icomplex) :: ci(expnlen,mxbox)
  type(icomplex) :: cli(expnlen,mxbox)

  if (2**nlevel > iws+1) then
     ! can only do FMM if some boxes are well-spaced

     call cpu_time(tsta(2))
     !     Multipole expansion in the lowest level.
     call multipole3d(ci,nlevel)

     call cpu_time(tsta(3))
     !     Multipole expansion transfer from child to parent box.
     call translate(ci,nlevel)

     call cpu_time(tsta(4))
     !     Local expansion for each level.
     call local(ci,cli,nlevel)

     call cpu_time(tsta(5))
     !     Output far field potential energy & error check.
     call fmm_energy(cli,nlevel)

     call print_times()
  else
     print*, 'Can''t do FMM with', nlevel, 'levels for well-spaced =', iws
  end if
end subroutine fmm3d

!     Perform multipole expansion on highest level
subroutine multipole3d(ci,nlevel)
  integer :: nlevel
  type(icomplex) :: ci(expnlen,mxbox), cylo(0:np,-np:np)
  interval(8) :: v(3)

  ci(:,:) = icmplx(0.0d0, 0.0d0)
  do ia=1,natom
     ndxbox = level_idx(nlevel) + box_idx(nhost(:,ia), nlevel)
     v = box_centre(nhost(:,ia), nlevel) - scoor(:,ia)
     cylo = Olm(achg(ia),v,np)
     inm = 0
     do n=0,np
        do m=-n,n
           inm=inm+1
           ci(inm,ndxbox) = ci(inm,ndxbox) + cylo(n,m)
        end do
     end do
  end do
end subroutine multipole3d

!     Translate multipole expansions from children and combine
!     to form expansions for parent boxes
subroutine translate(ci,nlevel)
  type(icomplex) :: ci(expnlen,mxbox), cylo(0:np,-np:np)
```

```
  interval(8) :: v(3)
  type(icomplex) :: cylm(0:np,-np:np)
  integer :: child_coords(3), parent_coords(3)
  integer :: child_idx, parent_idx

  do il=(nlevel-1),2,-1
     !         print *, 'Now translate from level', il+1, 'to level', il
     do icx=0,2**(il+1)-1
        do icy=0,2**(il+1)-1
           do icz=0,2**(il+1)-1
              child_coords(1) = icx
              child_coords(2) = icy
              child_coords(3) = icz
              child_idx = level_idx(il+1) &
                   & + box_idx(child_coords, il+1)
              if (.not.box_empty(child_idx)) then
                 parent_coords = child_coords / 2
                 parent_idx = level_idx(il) &
                      & + box_idx(parent_coords, il)
                 box_empty(parent_idx) = .false.
                 v = box_centre(parent_coords, il) &
                      & - box_centre(child_coords, il+1)
                 !     relation between parent box and its child!
                 cylo = Olm(1.0d0, v, np)
                 inm = 0
                 do jt=0,np
                    do kt=-jt,jt
                       inm = inm + 1
                       ido = 0
                       do lt=0,jt
                          do mt=-lt,lt
                             ido = ido + 1
                             if (abs(kt-mt).le.(jt-lt)) then
                                ci(inm, parent_idx) = &
                                     & ci(inm, parent_idx) &
                                     & + ci(ido, child_idx) &
                                     & * cylo(jt-lt, kt-mt)
                             end if
                          end do
                       end do
                    end do
                 end do

              end if
           end do
        end do
     end do
  end do
end subroutine translate

!     perform multipole to local expansion
subroutine local(ci,cli,nlevel)
  integer :: nlevel
  type(icomplex) :: ci(expnlen,mxbox)
```

```
  type(icomplex) :: cli(expnlen,mxbox)
  type(icomplex) :: cylm(0:np,-np:np)
  interval(8) :: v(3)
  real(8) :: t1, t2
  integer :: local_coords(3), parent_coords(3), target_coords(3), shift(3)
  integer :: last_box, begin_coords(3), end_coords(3)

  cli(:,:) = icmplx(0.0d0, 0.0d0)
  do il=2,nlevel
     iinter=0
     call cpu_time(t1)

     last_box = 2**il-1

     !     Each box cycle
     do ix=0,last_box
        do iy=0,last_box
           do iz=0,last_box
              local_coords = (/ ix, iy, iz /)
              ndxbox = level_idx(il) + box_idx(local_coords, il)
              parent_coords = local_coords / 2
              if (.not.box_empty(ndxbox)) then
                 call nearest(il, last_box, ndxbox, local_coords, &
                   & parent_coords, iinter, ci, cli)

                 !     Local expansion translation from parent to child box
                 ndxpbox = level_idx(il-1) + box_idx(parent_coords, il-1)
                 v = box_centre(local_coords, il) - box_centre(parent_coords, il-1)
                 call translate_expansion(v, cli(:,ndxbox), cli(:,ndxpbox))

              end if
           end do
        end do
     end do

     call cpu_time(t2)
     print '(a,f10.3,a,i8,a)', ' Used ',t2-t1, ' for ', iinter, ' interactions'
  end do
end subroutine local

subroutine nearest(il, last_box, ndxbox, local_coords, parent_coords, iinter, ci, cli)
  integer, intent(in) :: il, last_box, local_coords(3), parent_coords(3)
  integer, intent(inout) :: iinter
  type(icomplex), intent(inout) :: ci(expnlen,mxbox)
  type(icomplex), intent(inout) :: cli(expnlen,mxbox)
  type(icomplex) :: cylm(0:np,-np:np)
  interval(8) :: v(3)
  integer:: target_coords(3), shift(3)
  integer :: begin_coords(3), end_coords(3)

  begin_coords = max((parent_coords-iws)*2, 0)
  end_coords = min((parent_coords+iws)*2+1, last_box)
  !     Iteration list cycle
  do ixT=begin_coords(1),end_coords(1)
```

```
      do iyT=begin_coords(2),end_coords(2)
         do izT=begin_coords(3),end_coords(3)
            target_coords = (/ ixT, iyT, izT /)
            ndxtbox=level_idx(il) &
                 & + box_idx(target_coords, il)
            if (.not.box_empty(ndxtbox)) then
               shift = target_coords - local_coords
               if (any(abs(shift).gt.iws)) then
                  v = shift * scale(:,il)
                  iinter=iinter+1
                  cylm = Mlm(v,np)
                  inm=0
                  do jt=0,np
                     do kt=-jt,jt
                        inm=inm+1
                        ido=0
                        do nt=0,(np-jt)
                           do mt=-nt,nt
                              ido = ido + 1
                              cli(inm,ndxbox) = &
                                   & cli(inm,ndxbox) + &
                                   & ci(ido,ndxtbox) * &
                                   & cylm(jt+nt,mt+kt)
                           end do
                        end do
                     end do
                  end do
               end if
            end if
         end do
      end do
    end do

  end subroutine nearest

  ! get potential of each particle, and total of system
  subroutine direct_energy()
    implicit none
    real(8) :: ts(2)
    real(8) :: widdirect
    widdirect = 0.0d0
    print *, '--------------------------------------------------------'
    print *, '                    Direct Calculation '
    print *, '--------------------------------------------------------'

    call cpu_time(ts(1))

    call direct_system(widdirect)

    call cpu_time(ts(2))

    print *, 'Total energy:', E_pairwise
    print '(a46,es20.14)', 'mid(E_pairwise):', mid(E_pairwise)
    print '(a46,es10.4)', 'wid(E_pairwise):', wid(E_pairwise)
```

```
  print '(a46,es10.4)', 'max(wid(Ei_pairwise)):', widdirect
  print '(52x,a8,f10.3)', 'Time:', ts(2) - ts(1)
end subroutine direct_energy

subroutine direct_system(widdirect)
  real(8), intent(inout) :: widdirect
  interval(8) :: Eij
  real(8) :: ss, ys, errs, ts, si, yi, erri, ti

  E_pairwise = 0d0
  Ei_pairwise(:) = 0d0
  widdirect = 0d0

  si = -0d0
  ss = 0d0
  erri = 0d0
  errs = 0d0
  do i = 1, natom
     do j = i+1, natom
        Eij = direct_pair(scoor(:,i), scoor(:,j), &
             & achg(i), achg(j))
        Ei_pairwise(i) = Ei_pairwise(i) + Eij
        Ei_pairwise(j) = Ei_pairwise(j) + Eij
        widdirect = max(widdirect, wid(Eij))
        call round_up
        yi = inf(Eij) - erri
        ys = sup(Eij) + errs
        ti = si
        ts = ss
        si = si - yi
        ss = ss + ys
        erri = (ti - si) - yi
        errs = (ts - ss) + ys
        call round_nearest
     enddo
  enddo
  call round_up
  si = si * 1d0
  ss = ss * 1d0
  call round_nearest
  E_pairwise = interval(-si, ss)
end subroutine direct_system

pure function direct_pair(r1, r2, q1, q2)
  real(8), intent(in) :: r1(3), r2(3), q1, q2
  interval(8) :: d, direct_pair

  d = sqrt(sum((r1 - r2)**2))
  direct_pair = q1 * q2 / d

end function direct_pair

! Collect real and multipole potential energy
subroutine fmm_energy(cli,nlevel)
```

```
implicit none
type(icomplex) :: cli(expnlen,mxbox)
integer :: nlevel, dist(3), i, j, k, inm, ndxbox
type(icomplex) :: cylo(0:np,-np:np), ctstla
interval(8) :: v(3)
interval(8) :: Eij_direct, Ei_near
interval(8) :: Ei_FMM, E_FMM
real(8) :: errmax, widfmm
real(8) :: timestamp(3)
real(8) :: ss, ys, errs, ts, si, yi, erri, ti

print *, '--- Results -------------------'

errmax = epsilon(errmax)
widfmm = 0.0d0
E_FMM = 0.0d0

do i=1,natom
   call cpu_time(timestamp(1))

   si = -0d0
   ss = 0d0
   erri = 0d0
   errs = 0d0
   do j=1,natom
      if (j.ne.i) then
         if (all(abs(nhost(:,j)-nhost(:,i)).le.iws)) then
            ! Direct nearfield - used for FMM
            Eij_direct = direct_pair(scoor(:,i), scoor(:,j), &
                 & achg(i), achg(j))
            call round_up
            yi = inf(Eij_direct) - erri
            ys = sup(Eij_direct) + errs
            ti = si
            ts = ss
            si = si - yi
            ss = ss + ys
            erri = (ti - si) - yi
            errs = (ts - ss) + ys
            call round_nearest
         end if
      end if
   end do
   Ei_near = interval(-si, ss)

   call cpu_time(timestamp(2))

   !     FMM farfield value
   ctstla = icmplx(0.0d0,0.0d0)
   ndxbox = level_idx(nlevel) + box_idx(nhost(:,i), nlevel)
   v = scoor(:,i) - box_centre(nhost(:,i), nlevel)
   cylo = Olm(achg(i), v, np)
   inm = 0
   do j=0,np
```

```
        do k=-j,j
           inm=inm+1
           ctstla=ctstla+cli(inm,ndxbox)*cylo(j,k)
        end do
     end do

     Ei_FMM = dreal(ctstla) ! farfield only

     Ei_FMM = Ei_FMM + Ei_near ! total
     widfmm = max(widfmm, wid(Ei_FMM))
     E_FMM = E_FMM + Ei_FMM

     call cpu_time(timestamp(3))

     tsta(6)=tsta(6)+timestamp(3)-timestamp(2)
     tsta(7)=tsta(7)+timestamp(2)-timestamp(1)

     errmax = max(errmax, abs(mid(Ei_pairwise(i)) - mid(Ei_FMM)))
  end do

  ! energy has been overestimated by factor of 2
  E_FMM = E_FMM / 2d0

  print *, 'Total energy:', E_FMM
  print *, '--- Errors --------------------'
  print '(a45,es10.4)', ' max(wid(Ei_FMM)):', widfmm
  print '(a45,es10.4)', ' wid(E_FMM):', wid(E_FMM)
  print '(a45,es10.4)', ' max(abs(mid(Ei_FMM) - mid(Ei_pairwise))):', errmax
  print '(a45,es10.4)', ' abs(mid(E_FMM) - mid(E_pairwise)):', &
    & abs(mid(E_FMM) - mid(E_pairwise))

end subroutine fmm_energy

! Perform local expansion translation from parent to child box
subroutine translate_expansion(v, child_expansion, parent_expansion)
  interval(8), intent(in) :: v(3)
  type(icomplex), intent(in) :: parent_expansion(expnlen)
  type(icomplex), intent(inout) :: child_expansion(expnlen)
  type(icomplex) :: cylo(0:np,-np:np)

  cylo = Olm(1.0d0, v, np)
  inm = 0
  do jt=0,np
     do kt=-jt,jt
        inm = inm+1
        do lt=jt,np
           do mt=kt-(lt-jt), kt+(lt-jt)
              ido = (lt**2)+lt+mt+1
              child_expansion(inm) = child_expansion(inm) + &
                   & parent_expansion(ido) * cylo(lt-jt,mt-kt)
           end do
        end do
     end do
  end do
```

```
end subroutine translate_expansion

! Calculate the multipole-like term O_{lm} (with m.ge.0) for a point v(3).
pure function Olm(q, v, p)
  implicit none
  real(8), intent(in) :: q
  interval(8), intent(in) :: v(3)
  integer, intent(in) :: p
  type(icomplex):: Olm(0:p,-p:p)
  integer :: l, m
  interval(8) :: il, ilm, pplm(0:p,0:p), rfac
  type(polar_coords) :: v_pole
  type(icomplex) :: phifac0, phifac

  v_pole = polar(v)
  pplm = Plm(cos(v_pole%theta),p)
  Olm(:,:) = icmplx(0d0,0d0)
  phifac0%rl = sint(cos(-v_pole%phi))
  phifac0%im = sint(sin(-v_pole%phi))
  rfac = 1d0
  il=1d0
  do l = 0, p
     il = il*max(l,1)
     ilm = il
     phifac = icmplx(1d0,0d0)
     Olm(l,0) = q*rfac*pplm(l,0)*phifac/ilm
     do m = 1, l
        ilm = ilm*(l+m)
        phifac = phifac * phifac0
        Olm(l,m) = q*rfac*pplm(l,m)*phifac/ilm
     end do
     do m = -l, -1
        Olm(l,m) = dconjg(Olm(l,-m)) * dble(2*mod(-m+1,2)-1)
     end do
     rfac = rfac * v_pole%r
  end do
end function Olm

! Calculate the Taylor-like term M_{lm} (with m.ge.0) for a single point v(3).
pure function Mlm(v, p)
  implicit none
  interval(8), intent(in) :: v(3)
  integer, intent(in) :: p
  type(icomplex) :: Mlm(0:p,-p:p)
  integer :: l, m
  interval(8) :: il, ilm, pplm(0:p,0:p), rfac, rfac0
  type(polar_coords) :: v_pole
  type(icomplex) :: phifac0, phifac

  v_pole = polar(v)
  pplm = Plm(cos(v_pole%theta),p)
  rfac0 = 1d0 / v_pole%r
  Mlm(:,:) = icmplx(0d0,0d0)
  phifac0%rl = sint(cos(v_pole%phi))
```

```
  phifac0%im = sint(sin(v_pole%phi))
  rfac = rfac0
  il = 1d0
  do l = 0, p
     il = il*max(l,1)
     ilm = il
     phifac=icmplx(1d0,0d0)
     Mlm(l,0) = rfac*pplm(l,0)*ilm*phifac
     do m = 1, l
        ilm = ilm/(l+1-m)
        phifac = phifac*phifac0
        Mlm(l,m) = rfac*pplm(l,m)*ilm*phifac
     end do
     do m = -l, -1
        Mlm(l,m) = dconjg(Mlm(l,-m)) * dble(2*mod(-m+1,2)-1)
     end do
     rfac = rfac*rfac0
  end do
end function Mlm

!     convert from cartesian coords v(3) to polar (r,theta,phi)
pure function polar(v)
  implicit none
  interval(8), intent(in) :: v(3)
  type(polar_coords) :: polar
  interval(8) :: rxy, r2, rxy2
  real(8), parameter :: safety = 1d0-1d-15
  interval(8), parameter :: pi = [3.14159265358979323d0, 3.14159265358979324d0]

  rxy2 = v(1)**2 + v(2)**2
  r2 = rxy2 + v(3)**2

  polar%r = sqrt(r2)
  if (rxy2.eq.0d0) then
     if (v(3).cge.0d0) then
        polar%theta = 0d0
     else
        polar%theta = pi
     end if
     polar%phi = 0d0
  else
     rxy = sqrt(rxy2)
     polar%theta = acos(v(3)/polar%r*safety)
     polar%phi = acos(v(1)/rxy*safety)
     if (v(2).clt.0d0) then
        polar%phi = 2*pi - polar%phi
     end if
  end if
end function polar

! Calculate associated Legendre polynomials P_{lm}(x) up to l=p (m.ge.0)
pure function Plm(x, p)
  implicit none
  interval(8), intent(in) :: x
```

```
  integer, intent(in) :: p
  interval(8) :: Plm(0:p,0:p)
  integer :: i, m, l
  interval(8) :: somx2
  real(8) :: fact

  Plm(0,0) = 1d0
  somx2 = sqrt((1d0-x)*(1d0+x))
  fact = 1d0
  do i = 1, p
     Plm(i,i) = -Plm(i-1,i-1)*fact*somx2
     Plm(i,i-1) = x*fact*Plm(i-1,i-1)
     fact = fact + 2d0
  end do
  do m = 0, p-2
     do l = m+2, p
        Plm(l,m) = (x*(2*l-1)*Plm(l-1,m)-(l+m-1)*Plm(l-2,m))/(l-m)
     end do
  end do
end function Plm

!     get the one-dimensional index of a box given its 3D
!     indices (coords) in a given level of the octree (nlevel)
!     this is used to index the 1D arrays box_empty, ci and cli
pure function box_idx(coords, nlevel)
  implicit none
  integer, intent(in) :: coords(3), nlevel
  integer :: box_idx, ndivs

  ndivs = 2**nlevel
  box_idx = coords(1) &
       & + (coords(2) * ndivs) &
       & + (coords(3) * (ndivs**2))
end function box_idx

!     get the 3D coordinates of the box centre given its
!     3D indices (coords) in a given level of the octree (nlevel)
pure function box_centre(coords, nlevel)
  implicit none
  integer, intent(in) :: coords(3), nlevel
  interval(8) :: box_centre(3)

  box_centre = (coords + 0.5d0) * scale(:,nlevel)
end function box_centre

! get the a priori error bound for the FMM evaluation of potential of
! a single particle, given:
! Q: total charge of system
! d: box dimension at lowest level - each side is 2d
! p: number of terms in multipole expansions
! iws: number of boxes separating 'well spaced' boxes

pure function multipole_error(Q, d, p, iws)
  real(8), intent(in) :: Q, d
```

```
    integer, intent(in) :: p, iws
    real(8) :: multipole_error
    real(8) :: max_dist_ratio ! max ratio near:far
    real(8) :: truncation_error ! error due to truncation of multipoles

    max_dist_ratio = (Q / (((1 + 2*iws) - sqrt(3d0)) * d))
    truncation_error = (sqrt(3d0) / (1 + 2*iws)) ** (p+1)
    multipole_error = max_dist_ratio * truncation_error
  end function multipole_error


  pure function local_error(Q, d, p, iws)
    real(8), intent(in) :: Q, d
    integer, intent(in) :: p, iws
    real(8) :: local_error
    real(8) :: max_dist_ratio ! max ratio near:far
    real(8) :: truncation_error ! error due to truncation of Taylor series

    max_dist_ratio = (Q / (((2 + 2*iws) - 2d0*sqrt(3d0)) * d))
    truncation_error = (2d0*sqrt(3d0) / (2 + 2*iws)) ** (p+1)
    local_error = max_dist_ratio * truncation_error
  end function local_error

  ! Perform time scaling
  subroutine print_times()
    implicit none

    print *, '--- Times ---------------------'
    print('(6(a10,2x))'), 'Setup', 'Multipole', 'Mltpl Trn', &
         & 'Local Trn', 'Far Eval', 'Total FMM'
    print ('(6(f10.3,2x))'), tsta(2)-tsta(1), tsta(3)-tsta(2), &
         & tsta(4)-tsta(3), tsta(5)-tsta(4), &
         & tsta(6), tsta(5)-tsta(1)+tsta(7)
  end subroutine print_times
end module fmm
```

C.3 Source file: inttypes.F95

This file defines the handwritten interval and complex interval types, along with allowable operators on these types. It is included by a preprocessor statement (rather than being compiled to a library object) to allow the compiler to inline interval operations as necessary.

```
  type sinterval
    real(kind=8) :: inf
    real(kind=8) :: sup
 end type sinterval

  type icomplex
     type(sinterval) :: rl, im
  end type icomplex

  interface
     elemental subroutine round_up
```

```
     end subroutine round_up
     elemental subroutine round_down
     end subroutine round_down
     elemental subroutine round_nearest
     end subroutine round_nearest
  end interface

  interface operator(+)
     module procedure add_ss
     module procedure add_yy
  end interface

  interface operator(*)
     module procedure multiply_ds
     module procedure multiply_sd
     module procedure multiply_ss
     module procedure multiply_yd
     module procedure multiply_iy
     module procedure multiply_yi
     module procedure multiply_yy
     module procedure multiply_yz
     module procedure multiply_zi
  end interface

  interface operator(/)
     module procedure divide_yi
  end interface

  interface operator(-)
     module procedure subtract_ss
     module procedure negate_s
     module procedure negate_y
  end interface
```

C.4 Source file: intfuncs.F95

This file contains the implementation of functions of handwritten interval (`sinterval`) and complex interval types. Most of these functions are hand-optimised using the techniques developed in Chapter 3.

```
elemental function add_ss(a, b)
  type(sinterval), intent(in) :: a, b
  type(sinterval) :: add_ss

  call round_down
  add_ss%inf = a%inf + b%inf
  call round_up
  add_ss%sup = a%sup + b%sup
  call round_nearest
end function add_ss

elemental function add_yy(a, b)
  type(icomplex), intent(in) :: a, b
  type(icomplex) :: add_yy
```

```
  add_yy%rl = a%rl + b%rl
  add_yy%im = a%im + b%im
end function add_yy

elemental function subtract_ss(a, b)
  type(sinterval), intent(in) :: a, b
  type(sinterval) :: subtract_ss

  call round_down
  subtract_ss%inf = a%inf - b%inf
  call round_up
  subtract_ss%sup = a%sup - b%sup
  call round_nearest
end function subtract_ss

elemental function negate_s(a)
  type(sinterval), intent(in) :: a
  type(sinterval) :: negate_s

  negate_s%inf = -a%inf
  negate_s%sup = -a%sup
end function negate_s

elemental function negate_y(a)
  type(icomplex), intent(in) :: a
  type(icomplex) :: negate_y

  negate_y%rl = -a%rl
  negate_y%im = -a%im
end function negate_y

elemental function multiply_ds(a, b)
  real(8), intent(in) :: a
  type(sinterval), intent(in) :: b
  type(sinterval) :: multiply_ds

  call round_down
  multiply_ds%inf = a * b%inf
  call round_up
  multiply_ds%sup = a * b%sup
  call round_nearest
end function multiply_ds

elemental function multiply_sd(a, b)
  type(sinterval), intent(in) :: a
  real(8), intent(in) :: b
  type(sinterval) :: multiply_sd

  call round_down
  multiply_sd%inf = a%inf * b
  call round_up
  multiply_sd%sup = a%sup * b
  call round_nearest
end function multiply_sd
```

```
elemental function multiply_ss(a, b)
  type(sinterval), intent(in) :: a, b
  real(8) :: bin, bsn
  type(sinterval) :: multiply_ss

  bin = -(b%inf)
  bsn = -(b%sup)

  call round_up
  multiply_ss%inf = a%sup * bsn
  multiply_ss%sup = a%inf * b%inf
  multiply_ss%inf = max(multiply_ss%inf, a%inf * bin)
  multiply_ss%sup = max(multiply_ss%sup, a%inf * b%sup)
  multiply_ss%inf = max(multiply_ss%inf, a%inf * bsn)
  multiply_ss%sup = max(multiply_ss%sup, a%sup * b%inf)
  multiply_ss%inf = max(multiply_ss%inf, a%sup * bin)
  multiply_ss%sup = max(multiply_ss%sup, a%sup * b%sup)
  call round_nearest
  multiply_ss%inf = -multiply_ss%inf
end function multiply_ss

elemental function multiply_yd(a, b)
  type(icomplex), intent(in) :: a
  real(8), intent(in) :: b
  type(icomplex) :: multiply_yd

  multiply_yd%rl = a%rl * b
  multiply_yd%im = a%im * b
end function multiply_yd

elemental function multiply_iy(a, b)
  interval(8), intent(in) :: a
  type(icomplex), intent(in) :: b
  type(icomplex) :: multiply_iy
  type(sinterval) :: temp

  temp%inf = inf(a)
  temp%sup = sup(a)

  multiply_iy%rl = temp * b%rl
  multiply_iy%im = temp * b%im
end function multiply_iy

elemental function multiply_yi(a, b)
  type(icomplex), intent(in) :: a
  interval(8), intent(in) :: b
  type(icomplex) :: multiply_yi
  type(sinterval) :: temp

  temp%inf = inf(b)
  temp%sup = sup(b)

  multiply_yi%rl = a%rl * temp
```

```
  multiply_yi%im = a%im * temp
end function multiply_yi

elemental function multiply_yy(a, b)
  type(icomplex), intent(in) :: a, b
  type(icomplex) :: multiply_yy

  multiply_yy%rl = a%rl * b%rl - a%im * b%im
  multiply_yy%im = a%im * b%rl + a%rl * b%im
end function multiply_yy

elemental function multiply_yz(a, b)
  type(icomplex), intent(in) :: a
  complex(8), intent(in) :: b
  type(icomplex) :: multiply_yz


  multiply_yz%rl = a%rl * real(b) - a%im * aimag(b)
  multiply_yz%im = a%im * real(b) + a%rl * aimag(b)
end function multiply_yz

elemental function multiply_zi(a, b)
  complex(8), intent(in) :: a
  interval(8), intent(in) :: b
  type(icomplex) :: multiply_zi
  type(sinterval) :: temp

  temp%inf = inf(b)
  temp%sup = sup(b)

  multiply_zi%rl = real(a) * temp
  multiply_zi%im = aimag(a) * temp
end function multiply_zi

elemental function divide_yi(a, b)
  type(icomplex), intent(in) :: a
  interval(8), intent(in) :: b
  type(icomplex) :: divide_yi
  interval :: temp, res

  temp = interval(a%rl%inf, a%rl%sup)
  res = temp / b
  divide_yi%rl%inf = inf(res)
  divide_yi%rl%sup = sup(res)

  temp = interval(a%im%inf, a%im%sup)
  res = temp / b
  divide_yi%im%inf = inf(res)
  divide_yi%im%sup = sup(res)
end function divide_yi

elemental function sint(a)
  interval(8), intent(in) :: a
  type(sinterval) :: sint
```

```
  sint%inf = inf(a)
  sint%sup = sup(a)
end function sint

elemental function expi(cmp)
  type(icomplex), intent(in) :: cmp
  type(icomplex) :: expi
  complex(8) :: itemp, stemp
  real(8) :: blah

  itemp = dcmplx(cmp%rl%inf, cmp%im%inf)
  stemp = dcmplx(cmp%rl%sup, cmp%im%sup)

  itemp = exp(itemp)
  stemp = exp(stemp)

  expi%rl%inf = real(itemp)
  expi%rl%sup = real(stemp)

  expi%im%inf = aimag(itemp)
  expi%im%sup = aimag(stemp)
end function expi

elemental function icmplx(rl, im)
  real(8), intent(in) :: rl, im
  type(icomplex) :: icmplx
  icmplx%rl%inf = rl
  icmplx%rl%sup = rl
  icmplx%im%inf = im
  icmplx%im%sup = im
end function icmplx

elemental function dreal(cmp)
  type(icomplex), intent(in) :: cmp
  interval(8) :: dreal
  dreal = interval(cmp%rl%inf, cmp%rl%sup)
end function dreal

elemental function dconjg(cmp)
  type(icomplex), intent(in) :: cmp
  type(icomplex) :: dconjg
  dconjg%rl = cmp%rl
  dconjg%im = -cmp%im
end function dconjg
```

C.5 Source file: randomizer.F95

This file is used to generate repeatable pseudo-random particle distributions. It is also used in the direct evaluation examples in §4.2.

```
module randomizer
  real(kind=8), parameter :: RMIN = 0d0, RMAX = 3d0
contains
```

```
  subroutine fill_array(arr)
    real(8) :: arr(:,:)

    call set_seed()

    do i = 1, size(arr(1,:))
       do j = 1, size(arr(:,1))
          call rand(arr(j,i))
       end do
    end do
  end subroutine fill_array

  subroutine set_seed()
    integer, dimension(2) :: seed
    seed(1) = 1239
    seed(2) = 9869
    call random_seed(put=seed)
  end subroutine set_seed

  subroutine rand(result)
    !     get a random number between RMIN and RMAX
    real(kind=8) :: result
    real(kind=8) :: rnd
    call random_number(rnd)
    result = rnd * (RMAX - RMIN)
  end subroutine rand
end module
```

Bibliography

[1] Patriot Missile defense – software problem led to system failure at Dhahran, Saudi Arabia. Tech. Rep. B-247094, United States General Accounting Office, Feb 1992.

[2] VIS^{TM}Instruction Set User's Manual. Tech. Rep. 805-1394-03, Sun Microsystems, Inc., May 2001.

[3] UltraSPARC III Cu User's Manual. Tech. rep., Sun Microsystems, Inc., Jan 2004. v2.2.1.

[4] C++ interval arithmetic programming reference, Sun Studio 10. Tech. Rep. 819-0505-10, Sun Microsystems, Inc., Jan 2005.

[5] Fortran 95 interval arithmetic programming reference, Sun Studio 10. Tech. Rep. 819-0503-10, Sun Microsystems, Inc., Jan 2005.

[6] IA-32 Intel Architecture Software Developer's Manual. Tech. Rep. 253665-015, Intel Corporation, Apr 2005.

[7] ADAMS, E., AND KULISCH, U. *Scientific Computing with Automatic Result Verification*. Academic Press, San Diego, CA, 1993.

[8] ALT, R., AND MARKOV, S. On the algebraic properties of stochastic arithmetic: comparison to interval arithmetic. In *Scientific Computing, Validated Numerics, Interval Methods: Proceedings of SCAN 2000 and Interval 2000* (New York, NY, 2000), Kluwer Academic/Plenum Publishers, pp. 331–341.

[9] ANDRADE, M. V. A., COMBA, J. L. D., AND STOLFI, J. Affine arithmetic. In *INTERVAL'94* (St-Petersburg, Russia, 1994).

[10] BEATSON, R. K., AND GREENGARD, L. *A short course on fast multipole methods*. Wavelets, Multilevel Methods and Elliptic PDEs. Oxford University Press, 1997, pp. 1–37.

[11] BIRDIE, T. R., AND SURANA, K. S. The use of interval analysis in hydrologic systems. *Reliable Computing 4* (1998), 269–281.

[12] BOHLENDER, G., CORDES, D., KNÖFEL, A., KULISCH, U., LOHNER, R., AND WALTER, W. *Proposal for accurate floating-point vector arithmetic*. Scientific Computing with Automatic Result Verification. Academic Press, San Diego, CA, 1993, pp. 87–102.

[13] CHENG, H., GREENGARD, L., AND ROKHLIN, V. A fast adaptive multipole algorithm in three dimensions. *J. Computational Physics 155* (1999), 468–498.

[14] CHIRIAEV, D., AND WALSTER, G. W. Fortran 77 interval arithmetic specification. Tech. rep., Marquette University, May 1998.

[15] COPPERSMITH, D., AND WINOGRAD, S. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the Nineteenth Annual ACM Conference on Theory of Computing* (New York, NY, 1987), ACM Press, pp. 1–6.

[16] DONGARRA, J. J., CROZ, J. D., HAMMARLING, S., AND HANSON, R. J. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software 14*, 1 (Mar 1988), 1–17.

[17] FEFFERMAN, C. L., AND SECO, L. A. Aperiodicity of the Hamiltonian flow in the Thomas-Fermi potential. *Revista Matemática Iberoamericana 9*, 3 (1993), 409–551.

[18] FEFFERMAN, C. L., AND SECO, L. A. *Interval Arithmetic in Quantum Mechanics*, vol. 3 of *Applied Optimization*. Kluwer Academic Publishers, Dordrecht, 1996, pp. 145–167.

[19] FELDSTEIN, A., AND GOODMAN, R. Convergence estimates for the distribution of trailing digits. *J. ACM 23*, 2 (1976), 287–297.

[20] GARG, R. P., AND SHARAPOV, I. *Techniques for Optimizing Applications: High Performance Computing*. Sun Blueprints. Sun Microsystems Press, Prentice Hall Inc., Upper Saddle River, NJ, 2001.

[21] GILL, S. A process for the step-by-step integration of differential equations in an automatic digital computing machine. *Proceedings of the Cambridge Philosophical Society 47* (1951), 96–108.

[22] GILLIES, G. T. The Newtonian gravitational constant: recent measurements and related studies. *Reports on Progress in Physics 60* (1997), 151–225.

[23] GREENGARD, L., AND ROKHLIN, V. A fast algorithm for particle simulations. *J. Computational Physics 73*, 2 (1987), 325–348.

[24] GREENGARD, L., AND ROKHLIN, V. A new version of the fast multipole method for the Laplace equation in three dimensions. *Acta Numerica 6* (1997), 229–269.

[25] GUSTAFSON, J. ASCI should require intervals. Email correspondence, May 1999.

[26] HALES, T. Kepler conjecture. Email correspondence, Aug 1998.

[27] HANSEN, E. R. *Global optimization using interval analysis*. Marcel Dekker, Inc., New York, NY, 1992.

[28] HANSEN, E. R. Sharpness in interval computations. *Reliable Computing 3* (1997), 17–29.

[29] HASS, J., HUTCHINGS, M., AND SCHAFLY, R. The Double Bubble conjecture. *Electronic Research Announcements of the American Mathematical Society 1*, 3 (1995), 98–102.

[30] HIGHAM, N. J. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2002.

[31] HOEFKENS, J., BERZ, M., AND MAKINO, K. Controlling the wrapping effect in the solution of ODEs for asteroids. *Reliable Computing 8* (2003), 21–41.

[32] HOLZMANN, O., LANG, B., AND SCHÜTT, H. Newton's constant of gravitation and verified numerical quadrature. *Reliable Computing 2*, 3 (1996), 229–239.

[33] IEEE COMPUTER SOCIETY STANDARDS COMMITTEE. WORKING GROUP OF THE MICROPROCESSOR STANDARDS SUBCOMMITTEE, AND AMERICAN NATIONAL STANDARDS INSTITUTE. *IEEE standard for binary floating-point arithmetic*. ANSI/IEEE Std 754-1985. IEEE Computer Society Press, Silver Spring, MD, 1985.

[34] KAHAN, W. Pracniques: further remarks on reducing truncation errors. *Communications of the ACM 8*, 1 (1965), 40.

[35] KRÄMER, W., AND BANTLE, A. Automatic forward error analysis for floating point algorithms. *Reliable Computing 7* (2001), 321–340.

[36] LAWSON, C. L., HANSON, R. J., KINCAID, D. R., AND KROGH, F. T. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software 5*, 3 (Sep 1979), 308–323.

[37] LERCH, M., TISCHLER, G., AND VON GUDENBERG, J. W. filib++ - interval library specification and reference manual. Tech. Rep. 279, Lehrstuhl für Informatik II, Universität Würzburg, 2001.

[38] LERCH, M., TISCHLER, G., VON GUDENBERG, J. W., HOFSCHUSTER, W., AND KRÄMER, W. The interval library filib++ 2.0: Design, features and sample programs. Tech. rep., Bergische Universität Wuppertal, 2001.

[39] LIN, Y., AND STADTHERR, M. A. Advances in interval methods for deterministic global optimization in chemical engineering. *J. Global Optimization 29*, 3 (2004), 281–296.

[40] MAKINO, K., AND BERZ, M. Efficient control of the dependency problem based on Taylor model methods. *Reliable Computing 5* (Feb 1999), 3–12.

[41] MALCOLM, M. A. On accurate floating-point summation. *Communications of the ACM 14*, 11 (1971), 731–736.

[42] MCNAMEE, J. M. A comparison of methods for accurate summation. *ACM SIGSAM Bulletin 38*, 1 (Mar 2004), 1–7.

[43] MIEHE, D., KRÄMER, W., AND HOFSCHUSTER, W. Visualization of resulting sets coming from multiplication and division of complex intervals. Tech. Rep. Preprint, Bergische Universität Wuppertal, 2002/2003.

[44] MØLLER, O. Quasi double-precision in floating point addition. *BIT 5* (1965), 37–50.

[45] MOORE, R. E. Automatic error analysis in digital computation. Tech. Rep. LMSD-48421, Lockheed Missiles and Space Company, Jan 1959.

[46] MOORE, R. E. *Interval Analysis*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1966.

[47] NEUMAIER, A. Grand challenges and scientific standards in interval analysis. *Reliable Computing 8* (2002), 313–320.

[48] NEUMAIER, A. Taylor forms–use and limits. *Reliable Computing 9* (2003), 43–79.

[49] OGITA, T., RUMP, S. M., AND OISHI, S. Accurate sum and dot product. *SIAM J. Scientific Computing 26*, 6 (2005), 1955–1988.

[50] ROKNE, J., AND LANCASTER, P. Complex interval arithmetic. *Communications of the ACM 14*, 2 (1971), 111–112.

[51] SCHULTE, M. J., ZELOV, V., AKKAŞ, A., AND BURLEY, J. C. The interval-enhanced GNU Fortran compiler. *Reliable Computing 5* (1999), 311–322.

[52] STADTHERR, M. A., SCHNEPPER, C. A., AND BRENNECKE, J. F. Robust phase stability analysis using interval methods. In *AIChE Symp. Ser.* (1995), vol. 91, American Institute of Chemical Engineers, pp. 356–359.

[53] STINE, J. E., AND SCHULTE, M. J. A combined interval and floating point multiplier. In *Proceedings of the 8th Great Lakes Symposium on VLSI* (Los Alamitos, CA, 1998), no. 98TB100222, IEEE Computing Society, pp. 208–215.

[54] STRASSEN, V. Gaussian elimination is not optimal. *Numerische Mathematik 13* (1969), 354–356.

[55] SUNAGA, T. Theory of interval algebra and its application to numerical analysis. *RAAG Memoirs 2* (1958), 29–46.

[56] VIGNES, J. *An efficient stochastic method for round-off error analysis*, vol. L.N.C.S. 235 of *Accurate Scientific Computations*. Springer, 1985, pp. 185–205.

[57] WALSTER, G. W. The future of intervals: Keynote address. In *Scientific Computing, Validated Numerics, Interval Methods: Proceedings of SCAN 2000 and Interval 2000* (New York, NY, 2000), Kluwer Academic/Plenum Publishers, pp. 1–15.

[58] WARMUS, M. Calculus of approximations. *Bulletin de l'Academie Polonaise de Sciences 4*, 5 (1956), 253–257.

[59] WATANABE, Y., YAMAMOTO, N., NAKAO, M. T., AND NISHIDA, T. A numerical verification of nontrivial solutions for the heat convection problem. *J. Mathematical Fluid Mechanics 6* (2004), 1–20.

[60] WEAVER, D. L., AND GERMOND, T. *The SPARC Architecture Manual (Version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, 1994.

[61] WHITE, C. A., AND HEAD-GORDON, M. Derivation and efficient implementation of the fast multipole method. *J. Chemical Physics 101*, 8 (Oct 1994), 6593–6605.

[62] WOLFE, J. M. Reducing truncation errors by programming. *Communications of the ACM 7*, 6 (1964), 355–356.

[63] YANG, R. 3D fast multipole method code for evaluating the potential of a system of point charges. Fortran code, Jan 2005.