Norman Gray
2 December 2001

Theory and Modelling Resources Cookbook

Abstract

This cookbook is intended to assemble references to resources likely to be of interest to theorists and modellers. It's not a collection of standard recipes, but instead a repository of brief introductions to facilities. It includes references to sources of authoritative information, including those Starlink documents most likely to be of interest to theorists.

Although the topics are chosen for their relevance to theoretical work, a good proportion of the information should be of interest to all of the astronomical computing community.

Document information

Document date	2 December 2001
Last revised	10 March 2003
Version number	2
Distribution ID	2-5

Contents

1 Introduction

Theory work, unlike observational work, does not have a clear set of requirements for applications programs. There is no instrument data to reduce, no observations to plan, and all the observational fuss of calibration, flat-fielding, dark frames, image centroids can be avoided. All we need is a fast machine, and a compiler we can trust.

Well, not quite all. This cookbook is intended to be useful at and after the point when you start to wrestle with the computing details of your scientific problem. It's not a collection of standard recipes, but instead a repository of brief introductions to facilities you may not know existed, or didn't know how to get started on. It includes references to sources of authoritative information, including those Starlink documents most likely to be of interest to theorists. It doesn't try to teach you all of anything, but aims to give you enough information to decide if the topic is useful, and if the included references are worth pursuing.

Although the topics are chosen for their relevance to theoretical work (and for the purposes of this text, I'm taking the term to include all who develop their own modelling codes), a good proportion of the information should be of interest to all of the astronomical computing community.

1.1 Call for contributions

For such a wide ranging project, I cannot hope to have given an ideally just account of every topic. Please send comments, corrections, and expansions either to me at `norman@astro.gla.ac.uk` or to the Starlink software librarian at `ussc@star.rl.ac.uk`. I'll maintain a web page for the cookbook at `<http://www.astro.gla.ac.uk/users/norman/star/sc13/>`. Updates to the cookbook and its example programs can be found there.

I am particularly interested in comments on the following topics:

- I'd like to include a section on Monte Carlo simulations, but don't feel qualified to comment myself. Can anyone give any pointers?

- The cookbook is reasonably well-stocked with material on computation, but rather thin on more detailed astrophysical topics (*Section 4*). It does have a section on Stellar Atmosphere models (*Section 4.1*); what else should be here?

- Do I need a section on parallelization, beyond the very brief (indeed evasive!) mention in *Section 2.5.2*?

2 Computing

Everyone should start with the SUG, which should be available at your site, and is on the web at `<http://star-www.rl.ac.uk/star/docs/sug.htx/sug.html>`.

2.1 Unix guides

Before you can do anything, you need to make friends with the machine. The slogan to remember here is 'unix *is* user-friendly, it's just picky about who its friends are'. Keep calm, breath deeply, and make friends with a guru.

Your first source of information might be SUN/145, Starlink's Unix introduction. This covers the basics of logging on, moving around, issuing commands, creating files, and starting programming. It also includes references to other Starlink documents which can provide more detailed help on various aspects.

There are very good online introductions to Unix at `<http://unixhelp.ed.ac.uk/>` and `<http:/ /star-www.maps.susx.ac.uk/help/4ltrwrd/unixman.html>`

There are numerous books on Unix. Two which seem to be at the right level are *Unix Shells by Example* [quigley] and *Unix in a Nutshell* [nutshell]. Both cover the Bourne shell (`sh`) and the C shell (`csh`), plus other utilities such as `sed` and `awk` (see *Section 2.4.6.1*). By the way, let me put in a plug for `bash` as a usable shell, as it includes all the best bits of the Bourne and C shells. The only disadvantage of bash is that Starlink has, to some extent, standardised on `csh`, so that setup scripts, for example, are designed to work with `csh` alone; this is not often a real problem, since these scripts are generally simple enough that you can duplicate their effects 'by hand'.

You will occasionally see references to unix manual pages followed by a number. This indicates which section of the manual the documentation can be found in (section 1 is normal user commands, section 3 is standard library calls, section 5 is file formats, and so on). If you see a reference to `sed(1)`, for example, you'd read the manual page online with the command `man sed`. A useful variant of the `man` command is `apropos`, for example `apropos find`. This searches the list of man-pages, and lists all those commands which include a particular word in their short description.

2.2 Editors

One of the aspects of working on Unix which you'll have to deal with pretty early in your contact with it, is which editor to use. An editor is the tool you use to create your documents or programs. It's the tool you'll use more than any other.

It's very much a personal decision, which editor you use, and when you're feeling understimulated, you can discuss the matter with your officemates (don't get blood on the machines, though, and remember to wash and dry your thumbscrews carefully after use -- you'll need them again when you discuss programming languages -- see *Section 2.4*).

SUN/170, *Editors on Unix*, is an overview of some of the available options, listing their good and bad points. In addition to this, Brian McIlwrath wrote an overview of available text editors in the September 1998 (issue 21) edition of the Starlink Bulletin[1].

---

[1]`http://star-www.rl.ac.uk/bulletin/98sep/node15.html`

Emacs is many people's favourite (and mine). You can do just about anything in emacs -- it's a hugely productive environment -- but you may get cricks in your fingers from the odd key combinations. Emacs itself can help you up the long learning curve: there's a very good tutorial built-in to emacs, and available by typing `C-h C-t` (that's control-H, control-T), plus extensive documentation behind `C-h C-i`. Leave emacs with `C-x C-c` (obvious, really).

Vi is another favourite. It is indeed very powerful, but rather more of an acquired taste. An advantage of vi is that it's small, and therefore quick to start up -- this makes it useful if you want to make a quick change to a file, or make similar changes to a number of files, and it makes it useful as a editor for Pine, or any other mail program. An even bigger advantage of vi, however, is that it's available on every Unix system since the Ark, so if you can use vi, you can work on whichever Unix system you find yourself. The main disadvantage is that it is exceedingly unintuitive. There's quite a lot of help available for vi (it needs it, after all), but not, oddly enough, in the manual page, which describes in elaborate detail how to start up vi, but not what to do once you succeed. You'll find a good introduction to vi in Chapter 6 of the *Solaris 2.6 Advanced Users' Guide*, and in Chapter 2 of the old *SunOS 4 Documentation Tools* manual, and probably in the corresponding manual for any other Unix you use (don't worry about these being out of date, by the way, vi doesn't change much...). Online introductions to, and references for, vi include the the unixhelp manual[2], vi101[3], and even the vi-lovers home page[4]!

If you want to leave vi without saving what you've (accidentally?) typed, you can do so by typing `:q!` (if it beeps at you, or if those characters simply appear in the typing buffer on screen, try pressing escape once or twice first). By the way, it's pronounced 'vee-eye', not 'vye': pronounce it the wrong way and you'll lose all your credibility (of course, if you know the correct way to pronounce it, you'll lose all credibility with a different set of folk -- your choice).

For VMS fans, there's `jed`, which is a simple and fairly sane editor which includes an emulation of the EDT editor of old. It's available on Starlink systems, and documented in SUN/168.

Finally, there's `pico`, the editor used internally by the pine mailer. You can't go wrong with this one, but I'd imagine it could be a little painful if you're writing much code with it. Along the same lines, the `textedit` editor which comes with OpenWindows really isn't up to much. It looks pretty, but has zero support for programming. You'll be using an editor *a lot*, so it's worth investing time to learn to use a powerful one to its full extent.

2.3 Numerical analysis

Anyone writing numerical code needs *some* awareness of numerical analysis, to avoid burning up CPU time with a hideously inappropriate and inaccurate algorithm.

By far the most accessible introduction to numerical methods is Numerical Recipes[5] [nr], which comes in different editions, containing code in C, Fortran and Fortran 90. Use the second edition: the first has a significant number of bugs.

A large part of the book's popularity stems from the fact that its authors are scientists rather than numerical analysts, so that they are more concerned with producing reliable results than

---

[2]`http://unixhelp.ed.ac.uk/vi/ref.html`
[3]`http://www.devshed.com/Server_Side/Administration/Vi101/Vi101/page1.html`
[4]`http://www.thomer.com/thomer/vi/vi.html`
[5]`http://www.nr.com`

with a point of view which sometimes appears to see efficiency, unshakable robustness and algorithmic elegance as ends in themselves. For further discussion, and some caveats, see *Section 2.3.1*.

My advice is to use *Numerical Recipes* for your numerical programming until it runs out of steam on your particular problem. Follow the references in there, or look at the booklist[6] in the on-line Numerical Analysis FAQ. Also (perhaps unexpectedly), I suggest you take a look at the Usenet newsgroup `comp.lang.fortran`, even if you don't actually use Fortran. This is one of those happy few Usenet newsgroups with a high signal-to-noise ratio, and listening in on this can be profitable when the conversation turns to general numerical analysis. A similar resource is the JISCmail comp-fortran-90[7] list.

To support more specialised numerical computing, refer to the libraries section below, *Section 4.2*.

2.3.1 Numerical Recipes

Numerical Recipes[8] [nr] does not claim to be a numerical analysis textbook, and it makes a point of noting that its authors are (astro-)physicists and engineers rather than analysts, and so share the motivations and impatience of the book's intended audience. The declared premise of the NR authors is that you will come to grief one way or the other if you use numerical routines you do not understand. They attempt to give you enough mathematical detail that you understand the routines they present, in enough depth that you can diagnose problems when they occur, and make more sophisticated choices about replacements when the NR routines run out of steam. Problems *will* occur because the routines are not written to be bullet-proof, and if you use them thoughtlessly you can break them without much difficulty. Also, the routines will likely prove inadequate if you have a very demanding application which needs a more efficient or more specialised routine than the ones available here.

That is, the NR library is not filled with magic bullets, and if you try to use its contents as such, the only thing you'll shoot is your foot. However, NAG and SLATEC don't supply magic bullets either (though 'any sufficiently advanced library is indistinguishable from magic to the unsophisticated programmer', as Arthur C Clarke didn't quite say). You might use the NR routines successfully for years, but if and when they fail you, you'll use the more sophisticated substitute all the better because you'll understand *why* the simpler original was inadequate.

It is a consequence of the book's aims that the routines will not necessarily be the most intricately and obscurely efficient. Efficiency matters a lot if you are running some huge hydro code taking months of Cray time, but I would claim that if your code takes less than a week of wall-time to run, then the efficiency gains from using opaque library routines is simply not worth the cost in debugging time. No matter how robustly the library routine is written, you will be able to abuse it -- you will manage to break it somehow -- and when that happens your only recourse is to work through pages of Fortran IV trying to find the overflowing total, or the case you hadn't realised was marginal.

If you need very high efficiency, then take a course on numerical analysis and spend time understanding the subtleties of the library routines. Otherwise, read NR carefully (there are

---

[6]`http://www.mathcom.com/corpdir/techinfo.mdir/scifaq/q165.html`
[7]`http://www.jiscmail.ac.uk/lists/COMP-FORTRAN-90.html`
[8]`http://www.nr.com`

sometimes important caveats in the text), customise the routines to your problem, cross check the results you obtain, and keep alert. If you have an aversion to documentation, use NAG or another library: using the NR routines as a black box is a numerical recipe for disaster.

Despite NR's popularity, it has its critics. These are typically that the NR routines do not use the most efficient modern algorithms, and that they sometimes go wrong in borderline situations. While this may be true (and will be more true of the first edition than the second), it is largely a consequence of NR's declared intention of being accessible and intelligible. Without making the point explicit, as far as I can see, the NR authors privilege intelligibility over high efficiency, and practicality over unabusable robustness; I don't believe it's entirely fair, therefore, to criticise them for not being ultra-efficient and bulletproof. There is a collection of criticisms of the books by W Van Snyder at JPL, at <http://math.jpl.nasa.gov/nr/>, along with some suggestions for alternatives; there is a rebuttal from the NR authors at <http://www.nr.com/bug-rebutt.html>.

From the NR home pages, you can browse and print out chapters from the books, but you can't download the source code. Do remember that the NR code is copyrighted and *not* free; this may affect your ability to redistribute code which uses it.

2.3.2 Floating point representation

As with numerical analysis, the intricacies of how floating-point numbers are represented, and the quirks of their representation on different platforms, are a maelstrom into which you can fall, dazed, confused and unproductive. However (and again as with numerical analysis), if your codes become elaborate enough, then you are going to have to take the plunge.

Even if you have no plans to venture into the deep, there is a minimum awareness of the problems which can help you write more robust code.

What follows here is rather detailed, but it does, I believe, represent the majority of what you might need to know about floating point representation; there is a good alternative introduction in *Numerical Recipes* [nr], and an excellent and detailed introduction in chapter 2 of Sun's *Numerical Computation Guide* [sunncg] (appendix E of the same book claims to be 'What every computer scientist should know about floating point arithmetic', and is an edited reprint of [goldberg91]. It makes interesting reading, but it's probably more than many natural scientists need to know). My account here is not intended to supplant either of these sources. Below, I'll talk exclusively of IEEE base-2 floating point numbers, as defined in IEEE standard IEEE-754[9]; there are other standards, but they're now of historical interest only, as just about all modern machines other than VAXes and older Crays use IEEE. Most of what I say concerning accuracy will be about single precision; exactly the same issues arise with double precision, but you can sweep them under the carpet for longer.

2.3.2.1 Endianness of floating-point numbers   The IEEE standard leaves it open to chip manufacturers to decide the order in memory of the four or eight bytes of a floating point number (the same is true of integers, though they're not part of the IEEE spec). This is denoted by the endian-ness (or 'bytesex') of the machine. Alpha and Intel chips are little-endian; Sparcs, the Motorola 68k family used in Macs, and Motorola PowerPC chips, are big-endian (there's also a 'PGP-endian' ordering, but that really isn't likely to trouble anyone any more). This generally does not matter if you stick to a single platform (other than in the context of programs such as

---

[9]http://grouper.ieee.org/groups/754/

those described in *Appendix A.1* and *Appendix A.2*), but it makes a difference if you want to share raw data files (see *Section 2.5.2.3*) between architectures.

2.3.2.2 Accuracy   The central insight is that *you cannot represent an infinite number of reals in a finite number of bits without approximation*. It is a consequence of this that the result of any calculation will lose precision when it is represented as a floating-point bit pattern, and that the extent and importance of this loss of precision *depends on the operands*. Both of these seem obvious by themselves, but the consequences can be sufficiently non-obvious to trip you up.

A non-special IEEE floating-point single-precision longword represents a number

$$(-1)^s \times 1.m \times 2^{e-127}, \tag{1}$$

where s (the sign), m (the mantissa or significand) and e (the exponent) are base-2 numbers represented in 1, 8 and 23 bits respectively, and $0 < e < 255$; the offset 127 is called the *bias*. These are encoded into the 32 bits of the longword as shown in table Table 1.

	s	e	m
$1 =$	0	0 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$2 =$	0	1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$3 =$	0	1 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$2\epsilon =$	0	0 1 1 0 1 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$1 + 2\epsilon =$	0	0 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
$b = 1.000220 =$	0	0 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1 1 0 0 1 1
$(b-1) = 2.197266 \times 10^{-4} =$	0	0 1 1 1 0 0 1 0	1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0

Table 1: IEEE floating-point representations of numbers. This table is adapted from table 2--3 in [sunncg] and figure 1.3.1 in [nr], which have fuller discussions of the issues here. For discussion of b and $b-1$ see the text.

Firstly, it is important to distinguish between the *range* and the *accuracy* of floating-point numbers. The smallest and largest positive numbers which can be *represented* by normal IEEE single-precision numbers are $1.0_2 \times 2^{1-127} \approx 1.175 \times 10^{-38}$ and $1.1 \ldots 1_2 \times 2^{254-127} \approx 3.403 \times 10^{38}$, but this is very different from the accuracy to which these numbers are represented. These extreme values differ from the next representable ones up and down by $1 \times 2^{-23} \times 2^{-126} \approx 1.401 \times 10^{-45}$ and $1 \times 2^{-23} \times 2^{127} \approx 2.028 \times 10^{31}$ respectively, so that any number which differs from them by less than that amount is unrepresentable. This accuracy limitation is expressed by the *machine epsilon*, $\epsilon = 2^{-24} \approx 5.960 \times 10^{-8}$ (for IEEE single-precision), which is the maximum relative error incurred when a real number is represented in floating-point. It is a consequence of this that ϵ is the smallest number such that $1 \oplus 2\epsilon \neq 1$, where the operator $\oplus$ represents addition of floating-point numbers.[10]

Another way of thinking about this is as follows. When two floating-point numbers are added, the exponent of the smaller one must be increased to match that of the larger, and the mantissa (or significand), m, must be shifted rightwards to compensate. If the two numbers differ greatly in magnitude, then the smaller will lose significance in this operation, and if it is less than a

---

[10]This is a slightly different convention from that used in, for example, [nr], which defines ϵ to be the smallest number for which $1 \oplus \epsilon \neq 1$.

factor 2ϵ of the larger one, the shift will push the number right off the end of the mantissa, and contribute nothing to the sum. You can easily confirm that $2^{24} \oplus 1.0 = 2^{24} = 16777216.0$.

The immediate practical consequence of this is that if, in the bowels of some calculation, you are adding up many small numbers (doing an inner product for very large vectors, for example), then the final total may be grossly inaccurate if you've fallen into this trap. Similarly, if you are subtracting numbers which are almost equal, perhaps in the course of debiasing or removing some DC offset, the subtraction may allow all or most of the leading accurate digits to cancel, leaving the result to be determined by digits possibly heavily contaminated by roundoff. [RW]

Consider the simple calculation $ab - ac$, with $a \approx 1.000244$, $b \approx 1.000220$ and $c \approx 1.000269$. Here, the rounding in the two multiplications conspires to give a difference which has a huge relative error of 2.07×10^{-3}. We can, however, address the problem by rewriting the calculation in ways which are equivalent for real arithmetic, but inequivalent for machine arithmetic. If we do the subtraction before the multiplication, and calculate $a \otimes (b \ominus c)$, then we retain a little precision in the subtraction and end up with a relative error of 9.765625×10^{-4}. Finally, if we instead calculate $a \otimes ((b - 1) \ominus (c - 1))$, then we do the subtraction with a full 23 bits of precision (compare Table 1), and lose as little as possible of this in the final multiplication, and end up with a relative error of only 2.532525×10^{-7}. This is clearly a rather synthetic example, but it is not at all unrealistic, as it is equally clearly closely related to the common problem of calculating the discriminant $b^2 - 4ac$ of the quadratic formula, when the quadratic is nearly degenerate.

These problems are demonstrated in the short program `fpdemo.c` in *Appendix A.3*. Note that the numbers $a = 1 + 1/2^{12}$, $b = 1 + (1-)/2^{12}$ and $c = 1 + (1+)/2^{12}$ are (a) specifically chosen to demonstrate the effect, and (b) are calculated within the program, rather than being initialised from the decimal representations 1.000244, and so on. If the latter were not true, then the improvement in relative error would disappear, since all precision would have been lost in this initialisation. If you wish to explore the representation of floating-point numbers on your machine, you can do so using the example program `fpp.c` in *Appendix A.2*.

These problems are unfortunately both easy to run into, and hard to notice if you're not in the habit of looking for them whilst writing your code. A brute-force way around them is to do sensitive parts of your calculation in double precision. If you are doing the calculation in double precision and still running into the problem, then you will have to rearrange the calculation somehow, to contain the loss of precision.[11] Note, however, that a compiler's optimizer can frustrate your efforts here, if you've given it a sufficiently high optimization level that it starts reorganising your code: in general $(a \oplus b) \oplus c \neq a \oplus (b \oplus c)$, and $a \ominus b \oplus (b \ominus c) \neq a \ominus c$, but a high optimization level may instruct the compiler to ignore this fact, which can be disastrous for the accuracy of your code. Moral: make sure you know what the various optimization levels actually do before you invoke them.

As alluded to above, it is generally possible to sweep accuracy problems under the carpet by using double-precision floats. These occupy eight bytes of storage rather than four, and the analogue of Eqn. (1) is

$$(-1)^s \times 1.m \times 2^{e-1023},$$

where s, m and e are respectively 1, 52 and 11 bits long, and the bias is 1023. Thus the smallest and largest normal positive double-precision IEEE numbers are $1.0_2 \times 2^{1-1023} \approx 2.225 \times 10^{-308}$

---

[11]For example, appendix E of [sunncg] displays 'Kahan's summation formula', which allows you to perform a large sum without losing precision in the manner described above.

and $1.1 \ldots 1_2 \times 2^{2046-1023} \approx 1.798 \times 10^{308}$, and the machine epsilon for double precision is $2^{-53} \approx 1.11 \times 10^{-16}$.

On some platforms such as Solaris and Compaq, with the manufacturer's compilers, it is possible to use quadruple precision. You should not use double or quadruple precision automatically, however, for a variety of practical and principled reasons.

Firstly, if your program already suffers from rather prodigal use of memory (ie, has very large arrays), then the doubling in the size of real arrays will only make the problem worse and, by probably increasing the amount of swapping, might make your program significantly slower.

Secondly, although there are some issues to do with the relative speed of single- and double-precision calculations, these are probably not significant enough to be worth worrying about in all but the most long-running codes, and in any case would generally be swept up by the compiler (I'd welcome any correction or amplification on this point).[12]

Thirdly, if your results change if you switch to double-precision, then there is an accuracy problem in your code, which would probably bear some investigation. If there is some part of your code which genuinely requires the extra precision -- say because you have to add numbers of hugely different scale in accumulating a large inner product -- then there is not much you can do about it, and that part of your code, at least, must have the extra precision. Failing such an explanation, however, you can regard such behaviour as a miner's canary, indicating *some* potential problem with your code, which you might benefit from investigating further.

Fourthly, remember that some algorithms are designed to work with single precision, and will no longer work properly if you search-and-replace `double` for `float` or `real*8` for `real*4`. As just one example, one of the Numerical Recipes algorithms (`svdcmp`) includes a convergence test of, essentially, `if (val+err==val)`. Whatever the merits or otherwise of this as a test, it is a cheap way of establishing whether `err` has reached machine precision, which will not work in the expected way if `val` and `err` are double precision.

For further details on floating point representation, see, for example [hauser96], and informal but very useful discussions (both available on the web) by William Kahan [kahan96] and Jim Demmel [demmel]. The former is one of the authors of the IEEE-754 standard for floating-point numbers.

2.3.2.3 Other floating-point topics   As well as defining the format in which normal floating-point numbers are stored, the IEEE-754 standard includes the definitions for several other (classes of) special values.

When the exponent e (of a single-precision number) is zero, the longword is taken to represent the number

$$(-1)^s \times 0.m \times 2^{-126} \tag{2}$$

(compare Eqn. (1)). Unlike the usual floating-point numbers, which have an implied leading 1 in the significand and 23 bits of precision, and which are referred to as 'normalised', these have an implied leading 0 and less than 23 bits of precision. These are 'denormal' or 'subnormal' numbers, and are the result of an underflow, such as dividing the smallest normalised number by

---

[12]There used to be an issue here in that all of K&R C's floating-point operations were defined to be done in double-precision -- this made things easy for compiler writers, at the expense of runtime. This is no longer true in ANSI C.

two. This behaviour is known as 'gradual underflow', and allows the precision in a calculation
to degrade gracefully when it underflows. It is distinct from 'Store-0 underflow' common before
the IEEE standard, in which any expression which underflowed was replaced by zero. This was,
and remains, one of the more contentious parts of the IEEE standard. Be warned that older
Crays, which use their own floating-point format, have a Store-0 underflow policy, and that the
Alpha chip, although it generally implements IEEE floats, has a Store-0 underflow as its default
mode, and will neither produce, nor accept as input, denormalised numbers.

If the significand as well as the exponent is zero, the longword represents the number $(-1)^s \times 0$.
Note that the IEEE zero is *signed*, which allows $1/(+0)$ and $1/(-0)$ to be positive and negative
infinity; this can be important when doing calculations near branch points.

If the exponent is 255 and the significand is zero, the longword represents the value $(-1)^s \times \infty$. That is, infinity has a special value, distinct from the largest possible normalised number.
Positive infinity is generated by, for example $1/0$, or $\log 0$, where infinity is the mathematically
correct, but otherwise unrepresentable, value of a calculation.

If the exponent is 255 and the significand is non-zero, the longword is *Not a Number*, usually
represented by the string 'NaN'. A NaN is the result of an operation on invalid operands, such
as $0/0$ or $\log(-1)$. NaNs have the properties that any operation which has a NaN as an operand
has a NaN as its result; and any comparison on a NaN, such as $<$ or `==`, evaluates to False,
including `NaN==NaN`. The only exception is that, when `x` is a NaN, then `x != x` is true. Why
would you want to use such a peculiar number? Generally you don't, and its appearance in
a calculation is an error, which is why processors can be set to dump core when a NaN or an
Infinity is produced. However, it can be useful to turn off this behaviour if it is not off by default,
and rather than elaborately avoid producing a NaN at each stage, make a check once at the
end of a calculation, possibly invoking an alternative (possibly more expensive or special-cased)
algorithm if any NaNs are found.

Different compilers handle exceptional values in different ways; see Table 2. Of the three Starlink
platforms, only the Alpha traps on exceptional values by default.

Compiler	traps enabled by default	see	suppress with
Sun cc & f77	none	`-ftrap`	
Alpha cc	overflow, divide-by-zero, invalid operation	-fptm, -ieee	-ieee
Alpha f77	overflow, divide-by-zero, invalid operation	-fpe	-fpe1
gcc	none	`-mfp-trap-mode` (Alpha-gcc)	

Table 2: Summary of compiler IEEE traps

We have only scratched the surface of a rather intricate topic here. Both Suns and Alphas have
extensive `f77` and `cc` man-pages, which hint at the broad range of floating-point options available
when compiling and linking your code. See *Section 2.5.2.4* for discussion of the efficiency
tradeoffs when using IEEE exceptional values.

2.4 Programming languages

> I speak Spanish to God, Italian to women, French to men, and German to my horse.
> *Emperor Charles V* (attr.)

There is no One True Programming Language, nor even One True Compromise.

The language you use to write code is as much a function of your background, environment and personal tastes, as it is a function of technical merit. At some level or another, all programming languages are equivalent, but just as you'd look askance at anyone who wrote a stellar atmosphere code in TeX[13], it's clear that some languages are more suited for some tasks than others.

The obvious issue at this point is the choice between Fortran and C. Let me state first of all that, *for general scientific computing*, I do not believe the difference between the two is substantial enough, or the advantages of each are unqualified enough, that it warrants abandoning the language you're comfortable with and learning the other.

The languages' respective strengths only become significant when you are deep within their 'native territories': for Fortran this territory is that of numerically intensive codes with week-long runtimes, for C it is intricate manipulation of data structures. Away from these extremes, the choice is at the level of which to choose for a particular application, given that you know both, and an ideal application might be said to be one with a Fortran heart whirring away in a C harness.

Though Fortran and C are both simple languages, Fortran is simple in ways that an optimizing compiler can exploit. This means that for a numerically intensive application, where processing speed is very important, Fortran is the only suitable language.

Why is this? The reason can be illustrated by the two languages' loop semantics. Fortran's loop is very simple:

```
do 10, i=1,n
C  process array element a(i)
10 continue
```

The equivalent construction in C is

```
for (i=0; i<n; i++) {
      /* process array element a[i] */
    }
```

The difference is what the two language compilers can say about the loop. The Fortran compiler can know that the loop will be performed exactly n times, since neither `i` nor `n` may be changed within the loop; also the array `a(i)` cannot overlap with any other array within the loop body. On the other hand, in C, the loop `for (init; test; increment) { body }` is defined to be equivalent to the loop `init; while (test) { body; increment; }`, so that both `i` and `n` could change within the loop, and since `a` could be a pointer, it could point *anywhere*. The Fortran compiler could rewrite the first loop as

```
do 10, i=1,n,4
C     process array element a(i)
C     process array element a(i+1)
C     process array element a(i+2)
C     process array element a(i+3)
   10 continue
```

---

[13]Not impossible. Since someone has already done the hard work of implementing a BASIC interpreter in TeX (honestly!), you'd simply have to port your code to BASIC and let it rip.

cutting the number end-of-loop tests by a factor of four (this is a simple example of 'loop unrolling', and presumes that n is divisible by 4). The loop semantics could also make the elements of the loop candidates for parallelization, if the compiler and hardware support that[14]. The C compiler has to do a lot more investigative work before it can make any similar rearrangements.

Even if there were no such fundamental differences, the fact would remain that Fortran vendors have always sold their products on the strength of their optimizers, so that it is both easier *and* profitable for Fortran compilers to produce very fast code.

Though C's loop semantics are troublesome, it is really the pointer type which causes a lot of the trouble -- since a pointer is little more than an address, the compiler can have little clue what the programmer is aiming to do with it, and so has little choice but to make a literal translation of the C code into assembler.

The pointer type is redeemed, however, because it is this which allows C to code complicated data structures so naturally: not only aggregate types, but linked lists, queues, and all the other ways of holding, reading, parsing and generally juggling data, so beloved of Computer Science 1.

Fortran 90/95 goes some way towards addressing Fortran's weaknesses by introducing pointers (though the fact that they're living in a Fortran code doesn't make them any less troublesome to an optimizer), as well as proper data structures and some modularisation. At the same time as Fortran does that, however, C++ uses object-orientation to further enhance C's advantages in data manipulation, going so far as to make functions auxiliary features of data structures.

For general discussions of the issues involved in high-performance computing, see the book *High Performance Computing* [dowd].

Starlink ensures that C, C++, Fortran 77 and Fortran 90/95 compilers are available at all sites.

2.4.1 Fortran 77

Fortran 77 is probably the dominant current Fortran dialect, replacing earlier standards such as Fortran IV and Fortran 66[15]. There are several dialects of Fortran, containing various vendors' language extensions, but the only extensions which are usually portable are those in VAX Fortran, which includes the `enddo` statement for terminating do-loops, and the `%val()` function which is necessary to intermix Fortran and C code (see *Section 2.5.4*).

The standards document for Fortran 77[16] is not exactly bedtime reading, but can be quite useful when you have forgotten details of syntax, especially to make sure what you are writing is correct rather than just allowed by the compiler you are using at the time. As an introduction to Fortarn 77, I've heard good things about [metcalf85]. [MBT]

An important deficiency in Fortran is its lack of any standard way of dynamically allocating memory (as opposed to allocating arrays to be a fixed size at compile time). The Starlink CNF

---

[14]Note that, with pipelining, RISC chips can typically support some degree of on-chip parallelization, even for a single CPU.

[15]A statement like that can't be made without some qualification. Depending how you added it up, you could probably make a case that old Fortran dialects probably have more code actually running on CPUs, since many heavily-used libraries were written a long time ago.

[16]`http://www.fortran.com/fortran/F77_std/rjcnf.html`

library, SUN/209 is intended to make this reasonably easy, and includes a brief discussion of the underlying mechanism. This is rather a can of worms, but the essential technique is to write a bit of C which obtains a pointer to a block of memory via a call `malloc`, return that pointer to the Fortran program as an integer, then use `%VAL()` to supply that integer as an argument to a function which is expecting an array. This is non-standard (and not likely to become standard, now that Fortran 90 includes its own mechanisms for dynamic memory allocation), but it is a well-established technique and therefore probably more portable than you have any right to expect.

There is a large number of introductions to Fortran, but not, I believe, a single preeminent one. The *Starlink application programming standard*, SGP/16, is a collection of programming style guidelines for Fortran, and includes further references.

There is a reasonable amount of online information on Fortran, which is well-covered at the 'Fortran Market' (`<http://www.fortran.com/fortran/info.html>`), which includes several Fortran FAQs.

2.4.2 Fortran 90/95

Almost immediately after Fortran77 was standardised, work began on its successor. Although this project was named Fortran 8X, the work took long enough that the final standard was named Fortran 90.

The Fortran standard is maintained by ISO committee JTC1/SC22/WG5[17], and the current version of the standard is ISO/IEC 1539-1 : 1997[18].

Fortran 90 was an attempt to respond to the numerous developments in language design, seen since Fortran's last standardisation in the sixties and seventies. In contrast to Fortran 77, which aimed to standardise existing practice, Fortran 90 was an attempt to push forward the development of Fortran as a language. A summary of the new features (adapted from [metcalf96]) is:

- Free format source code form (no more column-counting).

- A means for the language to evolve by labelling some features as 'obsolescent'.

- Array operations (for example, `X(1:N)=R(1:N)*COS(A(1:N))`).

- Pointers.

- Improved facilities for numerical computation including a set of numeric enquiry functions.

- User-defined derived data types composed of arbitrary data structures and operations upon those structures.

- Facilities for defining collections called 'modules', useful for global data definitions and for procedure libraries. These support a safe method of encapsulating derived data types. Operator overloading and prototyping.

- Requirements on a compiler to detect the use of constructs that do not conform to the syntax of the language or are obsolescent.

---

[17]`http://www.nag.co.uk/sc22wg5/`
[18]`http://www.nag.co.uk/sc22wg5/IS1539-1_1997.html`

- New control constructs such as the `select case` construct, an `exit` and a new form of the `do`.

- The ability to write internal prodedures and recursive procedures, and to call procedures with optional and keyword arguments.

- Dynamic storage (automatic arrays, allocatable arrays, and pointers).

- Improvements to the input-output facilities, including handling partial records and a standardized `namelist` facility.

- Many new intrinsic procedures, (date, precision, arrays, ...)

Fortran 90 is backwards compatible with Fortran 77, so that every *strictly* conformant Fortran 77 program is also a valid Fortran 90 program. However, many of Fortran 77's odder features are strongly deprecated, and may disappear in the next revision of Fortran, due to appear sometime in the next decade. Fortran 95 is a minor revision of Fortran 90.

There is an increasing number of books on Fortran 90/95; for a booklist, refer to the Fortran Market at the URL above. I am, of course, reluctant to recommend books I have not used myself, so I will simply note that I have heard good things about [metcalf96], and that the first author was prominent in the negotiations concerning the development of the Fortran 90 standard.

If you plan to use Fortran 90/95, but avoid the deprecated parts of Fortran 77 altogether, you might be interested in F[19], which is a subset of Fortran 90 with all the deprecated features removed. You can obtain an F compiler from Imagine1, including a free educational version for Linux.

Several Fortran compilers were reviewed in a short report[20] of January 1997 by the high performance computing project at Liverpool.

It is occasionally necessary to produce code in a mixture of Fortran and C. See *Section 2.5.4*.

2.4.3 C

There are hundreds of books on C, but the only *essential* one is the Kernighan and Ritchie book [kr] (also known as just 'K&R'). This is a very short book, by the authors of the language, which combines a tutorial introduction with a detailed reference to the language and its standard libraries. I believe it's the only C book you'll ever need.

The book's compression makes it possible, and even advisable, to read it from beginning to end. It avoids the bloat found in many other C books by not teaching you how to program, by never saying anything twice, and by never attempting the empty reassurance that 'it's all easy, really'; its examples often illustrate more than one point at a time, and culminate in a sample implementation of the library function `malloc`. Science textbooks don't talk down to you; I don't see why computer texts should be excused for doing so.

It follows from this that K&R is not necessarily the ideal recommendation for someone new to programming, who might need something a little more comprehensive. My recommendation in

---

[19]`http://www.swcp.com/~walt/imagine1/`
[20]`http://www.liv.ac.uk/HPC/FortranCompilerStudyHTML/FortranCompilerStudyHTML.html`

that case is to find an introductory C (or even general programming) book which covers what you want without irritating you, and use that up to the point where you feel comfortable with K&R.

C allows the programmer a considerable degree of freedom in how an algorithm is expressed. The fact that this freedom is easily abused makes style guides more common for C than for other languages. Starlink has adopted *The Elements of C Programming Style* by Jay Renade and Alan Nash as its principal C programming standard, as well as producing a compact collection of programming style suggestions in SGP/4.

One way of making your life easier in this respect is to be consistent about using current standard ANSI-C. See *Section 2.5.5.*

The C language as originally designed by Kernighan and Ritchie was standardised as ANSI-C in 1990 (K&R 2nd edition describes the latter). This standard is currently being revised by an ISO working group with the memorable name ISO/IEC JTC1/SC22/WG14 -- C[21]. One of the motivations to the work is to provide better support for floating point calculations in C, such as defining a standard interface to the IEEE floating point exceptions.

The excellent C FAQ[22] contains *substantially* more than you realised you wanted to know about C. It contains detailed discussions of both common and arcane problems.

2.4.4 C++ and object orientation

C++ is an object-oriented version of C, and was finally standardised in 1997. Compiler makers have tracked the developing standards, so that recent C++ compilers should conform pretty closely to the standard even if they formally pre-date it.

Object-orientation is the notion that functions are *attached to* data, rather than simply operating on them. For example, if a C program were to declare a data type `Array`, and initialise a variable of that type with `Array a;` then one can imagine a function to return the determinant of the array, declared as `float determinant (Array a);`. In C++, the 'function' to obtain the determinant could be declared as *part of the data type*, and obtained via the expression `a.determinant()`. There are two main points to this. Firstly, the so-called 'member function' `determinant` can have privileged access to the internal representation of the data type `Array`, so that other parts of the program need not, and may not, manipulate that representation, erroneously or otherwise. Secondly, and consequently, the representation and matching member functions can be *changed* with the guarantee that the rest of the program will be unable to tell the difference. This is characterised in the remark that new programs have always been able to use old code (in libraries), but object-oriented approaches mean that old programs can use new code (when an implementation changes, while the interface remains the same).

Of course, both of these points are to some extent true of traditional programming languages -- indeed I'd doubt that there's anything you can do in C++ which you couldn't do with some ingenuity in C -- but the point is that they are much easier, and much more natural, in C++.

It may be clear at this point that C++ is not a beginner's language. The result of grafting a high-level abstraction onto portable assembler is a language with far more syntax than is healthy. Do not feel you need to learn C before C++-- they are closely enough related that the

---

[21]`http://std.dkuug.dk/JTC1/SC22/WG14/`
[22]`http://www.eskimo.com/%7Escs/C-faq/top.html`

differences can be confusing. My feeling, however, is that you should consider learning Javafirst if you have the time: there is relatively little that needs to be unlearned going from Java to C++, and Java's simplicity makes it easier to come to grips with object-orientation, without drowning in a sea of punctuation.

The excellent C++ FAQ[23] includes book recommendations. The canonical book for C++, with the same status K&R has for C, is *The C++ Programming Language* [stroustrup], by the language's author. However, I find Stroustrup's book rather irritating: it's rather badly organised, and the index is *dreadful.*

Note, by the way, that the C++ compiler on Suns is named `CC`, on Alphas named `cxx`, and on Linux named both `c++` and `g++`.

2.4.5 Java

Java was developed by Sun and at present (mid-2002) eclipsed only by XML as the current Big Thing. Source code is compiled to machine-independent bytecode, which is then either interpreted, or compiled to machine code on the fly, by a Java Virtual Machine (JVM) on a particular platform. The main interest to astronomy might be in developing machine- and architecture-independent interfaces either to codes or archives. It's also, I think, of use as a stepping-stone to C++, since its object-oriented features are less obscured by syntax than they are in C++. A very good textbook, written (you won't be surprised to guess) by two of the language's authors, is *The Java Programming Language* [arnold98]; once you've mastered the basics, there's a good deal of helpful advice in *Effective Java* [bloch01]. There are numerous resources at Sun's Java site at `<http://java.sun.com>`.

Java is a mixture of an interpreted and a compiled language. Java source code is compiled by the Java compiler into bytecodes, which are then interpreted by a Java Virtual Machine. These bytecodes are low-level, but architecture-independent so that, in principle, only the virtual machine and its runtime need to be ported to a new machine, whereas the code should be completely portable. This does not work quite as well in practice as it does in principle, but it does mean that Java is not too far off the ideal of run-anywhere code.

Since Java bytecodes are ultimately interpreted, Java programs have the potential to be rather inefficient. However, just-in-time compilers (JIT), which analyse running code and optimize the most heavily-used parts, and similar developments, should help the situation improve in the future. This is possible because JITs have access to both the program source code and the running program, and so can combine the features of both an optimizer and a profiler, and thus optimize more aggressively than a traditional compiler could. This, combined with Java's good built-in support for different platforms and for resource-discovery, means that, possibly somewhat surprisingly, Java has been suggested as a suitable language for high-end codes. The Java Grande Forum[24] is concerned with investigating and promoting developments in this area.

Note that it is relatively easy to write slow Java (for example, adding Strings is a *lot* slower than using a StringBuffer; input and output streams are not (extensively) buffered, so if you have a lot of I/O, you would be well advised to use `Buffered{Input,Output}Stream` objects wrapping your I/O objects; small-object creation is extensively and increasingly optimised, but it's still not dirt-cheap, so look out for that in a small loop). That means that using a profiler

---

[23]`http://www.parashift.com/c++-faq-lite/`
[24]`http://www.javagrande.org/`

(see *Section 2.5.1*) is particularly important. However, with Java, that's easy, because there's one built in to the JVM. Give the option `-Xrunhprof:help` to find out how to invoke it. For example:

```
java
          -Xrunhprof:cpu=samples,file=myprog.hprof
          myclass
```

There's quite a lot of information in here (and the file format is under development), but as with all profilers, you'll see a list of the number of times various methods were called: the ones at the top of the list are the ones where the program spent most of its time, so work out why, and concentrate on making them faster. Ignore the rest.

Java is still developing. At present (mid-2002) Java 1.3.1 is long in the tooth but stable. Version 1.4 is in beta, and on the point of being released properly; it includes a few changes to the language, most noticeably the inclusion of an `assert` construct.

2.4.6 Other languages

C and Fortran cover most of the bases for scientific computing, but there are one or two others which come in useful occasionally.

2.4.6.1 awk and sed   Often, you can find yourself performing some repetitive editing task, for example massaging data into a form which a program can conveniently read. Such tasks can conveniently, and reliably, be done by programs such as `awk` and `sed`. Neither of these utilities is as well-known as it should be, as they can save a great deal of tedious and error-prone effort.

`sed` is a version of the *very* simple editor `ed`, which is specialised for performing edits on a stream of text. For example, the following rather elaborate `sed` script prints all the section headings from a LaTeX document:

```
sed -n
          's/^\\\(sub\)*section{\(.*\)}.*$/\2/p'
          sc13.tex
```

This may look like gibberish, but it is simpler than it looks. The option `-n` instructs `sed` not to print out input lines, which it does by default. The `sed` expression in quotes calls the `s` command: whenever the 'regular expression' between the first pair of slashes matches, the `s` command replaces it with the expression between the second pair and, because the `s` command is suffixed with a `p`, prints out the modified line. The regular expression matches lines which start with a backslash, have zero or more occurrences of the string 'sub', which is followed by the string '`section{`', then any sequence of characters, followed by a } then any characters, ending at the end of the line. The caret `^` matches the beginning of a line, the backslash is a special character, so that it must be 'escaped' by prefixing it with another backslash, `\\`, the grouping operators are `\(` and `\)`, the asterisk indicates that the previous (bracketed) expression may be present zero or more times, the dot matches any character, and the dollar matches the end of the line. As well as grouping, the parentheses save what they match, and the expression `\2` in

the replacement text refers to what the second pair of parentheses matched, namely the text between the curly braces. The overall result is that the matched string (the whole of the line) is replaced by the contents of the braces and then, because of the `p` suffix, printed.

Another useful tool is `awk`, named after its designers Aho, Weinberger and Kernighan. Like `sed`, it works through a text file, executing scraps of code whenever a line matches some condition. Before it does anything with a line, it breaks it into fields, separated by whitespace by default. Consider the following example[25]

```
ps u | sed 1d | \
    awk '{print $4, $0; totmem+=$4}; END {printf "total memory: %f\n", totmem}'
```

This (not terribly useful) line generates a process listing using `ps`, uses `sed` to delete the first line (that is, it executes the command `d` on line number 1), and then passes the result through the `awk` program contained in quotes. On every line, this prints field number 4 (the `%MEM` column in the listing) and field 0 (which is `awk`-speak for the whole input line), and adds the value of the fourth field to a running total; on the line matching the pattern `END` -- that is, the pseudo-line at the end of the file -- `awk` prints out the accumulated total.

You won't typically generate expressions as complicated as these on the fly (at least, not until you get *really* good). This example is intended to suggest that you can, in aliases or in scripts, perform quite complicated transformations of text files. For further details you could look at the `sed` or `awk` man-pages, which are complete but very compressed, or work through a tutorial in your system's printed documentation. There are several guides to `sed` and `awk`, but you might be best off, initially, using an advanced introduction to Unix, such as [quigley] or [nutshell]. The canonical documentation for regular expressions is on the `ed(1)` manual page.

2.4.6.2 Perl   Perl is a general-purpose scripting language. It started off as a text-reformatting facility, rather like a super-`awk`, but it has now grown to the point where it really is a programming language in its own right, capable of supporting quite substantial projects. Perl programmers can call on a huge range of supporting code, collected at the Comprehensive Perl Archive Network, CPAN[26], to do everything from internet programming to database access. Perl's expressive power makes it ideal for rapid development of all sorts of complex systems -- some huge proportion of the web's CGI scripts, for example, are written in Perl. Unfortunately, the flexibility of Perl's syntax make it quite possible to write spaghetti, the like of which we have not seen since Fortran IV dropped out of fashion.

The Perl manual pages are reasonably clear. O'Reilly publishes a good book on Perl, written by Larry Wall, its author [wall]. This is a good reference book, but [schwartz97] is possibly a better tutorial introduction. Perl regular expressions are *slightly* different from the ones used by `sed` and friends -- see the `perlre` manual page.

Perl is a semi-interpreted language. Somewhat like Java, when the Perl interpreter first processes your Perl script, it compiles it to an internal code, which it then proceeds to interpret. This means that Perl programs have a relatively long startup time, but run reasonably efficiently after that. This is not a big issue in most applications.

---

[25]There are two versions of `ps` on Suns -- this example assumes you are using the `/usr/ucb/ps` version.

[26]`http://www.cpan.org`

The current (end-2001) version of Perl is 5.6 or thereabouts. Perl 6 will be a significant step in the evolution of the language: it's in the offing, but still some way away.

2.5 Code topics

Several of the topics mentioned here are discussed at greater length in the Sun Fortran User's Guide [sunf77], which is of interest even if you program only in C.

2.5.1 Profiling

Before you start doing any optimization at all, check which parts of your code are slowing the machine down -- there's no point in tweaking, for example, a one-time initialisation routine which takes only a tiny fraction of the program's runtime. You do this by using a profiler.

Different compilers will invoke a profiler (presuming they have one) in different ways. The Sun and Digital Fortran compilers include profiling code if you give the option -pg to the f77 command. Compile *and* link the program with this option (if you wish, you can compile only those modules you want to profile with the -pg option, but you must include the option in the final link command), and then run it. This run will create a file gmon.out in your current directory. Then you can run gprof. Taking as example the programs in *Appendix A.4*, we can build and profile them as follows:

```
% f77 -pg -o timewaster p1.f p2.f p3.f
p1.f:
MAIN silly:
p2.f:
mkidentity:
p3.f:
determinant:
Linking:
% timewaster
1.00000
% ls
gmon.out        p1.o            p2.o            p3.o
p1.f            p2.f            p3.f            timewaster*
% gprof timewaster >timewaster.gprof
```

The output file timewaster.gprof is many lines long, even though the program is so short! The file contains a great deal of information, but buried amongst it is the following display (from the Sun profiler, trimmed):

```
called/total          parents
index  %time     self descendents  called+self     name      index
called/total          children


0.05        0.64       1/1             main [2]
[1]     97.2    0.05        0.64        1          MAIN_ [1]
0.64        0.00  100000/100000        mkidentity_ [4]
```

```
       0.00          0.00        1/1             __s_wsle_nv [167]
       0.00          0.00        1/1             determinant_ [8]
       0.00          0.00        1/1             __do_l_out [157]
       0.00          0.00        1/1             __e_wsle [158]
       -------------------------------------------------
       0.64          0.00  100000/100000         MAIN_ [1]
       4]      90.1    0.64          0.00  100000              mkidentity_ [4]
```

This indicates that the subroutine `mkidentity` was called 100000 times from the main program, and that 90.1% of the time -- a total of 0.64 seconds -- was spent in that routine. Were this a real project, this would indicate that this routine would be a good one to examine for speedups.

This example, and a further one using the `tcov` utility, were taken from [sunf77], which might be consulted for further details. `pixie` is the corresponding utility on Compaqs -- see its man-page for details [RW].

`gprof` is not specific to Sun, but is available for other architectures, and other compilers (including `gcc`) as well.

2.5.2 Optimization

Just say no! Or, if you need authority:

> *[Donald Knuth, in Literate Programming]*Premature optimization is the root of all evil.

The first thing to ask yourself, when you are considering performance enhancements on your code is 'do I really need to do this?'. You should only attempt optimizations once you have either identified a particular part of your code which needs improving, or else you *know* that you can write the efficient version of the code correctly. If the 'efficient' version is unnatural, then the savings in runtime you achieve by reordering the code have a good chance of being wiped out by the time spent debugging a convoluted juggling act. Within reason, prefer simplicity and robustness to efficiency[27] -- get your code working correctly and believably, and only then start worrying about speed. That way, if you later establish that you need to work on part of the code, you at least have a trusted set of results to check the new version's against. Another way of putting this is that it's easier to optimize correct code than it is to correct optimized code.

If you're writing a code which will run for days or more, then it might be worthwhile investing a significant effort in tuning your code for performance. Doing this properly might involve you learning more about your machine's architecture than you might enjoy, and as as result will make your program less portable to the next generation of whizz-bang hardware, or even the next compiler.

It is generally worthwhile to use hand-optimized routines, if they are available for your compiler. Where this can pay particularly good dividends is in the case of linear algebra, where the speed of a calculation can be significantly improved ('a factor of several') by routines carefully written to access memory in an optimal way. There is a standard interface to such routines called the BLAS (Basic Linear Algebra Subroutines), which your compiler documentation may mention.

---

[27]However, there is *no* excuse for Bubble Sort!

Although it won't be as fast as a machine-specific implementation, the free BLAS implementation at Netlib[28] (UK mirror[29]) should speed your codes up significantly. [MBT,RW]

The first lesson to learn about optimization is that the compiler can probably do it better than you can.

Most compilers will have a `-On` option, to allow you to set how aggressive the compiler's optimization will be. The GNU gcc compiler, for example, allows optimization levels 1, 2 and 3, and Solaris Fortran has five levels, which progressively enable more techniques for improving your program's execution speed. Compaq compilers have a '`-tune host`' option, to produce code optimized for a particular machine. [MBT]

If your program starts behaving oddly, at some point you should start worrying about errant optimization, if you have that turned on. Optimization works by the compiler recognising patterns in your code, and possibly rearranging the output assembly language to take advantage of these. If it gets this wrong (because there's a bug), if your program relies on a particular floating-point behaviour, or if you're doing something sufficiently weird with your programming language that you manage to confuse the compiler, then you could trick it into doing the wrong thing.

Specifically, if you have occasion to worry about the order in which expressions are evaluated -- for example to conserve precision, as described in *Section 2.3.2.2* -- then you should be very wary of optimization, as one of the first things an optimizer *might* do is to reorder expressions which are associative mathematically but not computationally. If you have some code like this, it might be best to isolate a single routine in a module of its own and compile it separately with its own appropriate optimization level.

Parallelization is another type of optimization. Just say no several times to this, but if you have an application which really needs it, then buy a book and some aspirins. See *Section 2.5.2.7*.

Although you should generally leave optimization to the compiler, there are some things you can do to at least avoid getting in its way.

2.5.2.1 Avoid using the `register` keyword in C   This is, historically, supposed to suggest to a compiler that a particular variable might be better stored in a register than in main memory, though the actual semantics defined in the C standard is that the keyword is an assertion that the program nowhere takes the address of the variable so qualified. Such an assertion means, amongst other things, that the variable will never be visible from any other functions, which might trigger other optimizations. [SG]

You do not need to use this keyword to actually suggest to the compiler that it store the variable in a register, since the compiler might well do this anyway, and have a better idea of which (non-obvious, or temporary) variables to choose than you have.

2.5.2.2 Walk through arrays in the correct order   A multi-dimensional array is necessarily stored in memory as a linear array. If you have to process this entire array, it will be fastest if you try to do so in the order in which the array is stored. Fortran arrays are stored with the leftmost index varying fastest. That is, the Fortran array `integer i(2,2)` will be stored in memory in the order `i(1,1)`, `i(2,1)`, `i(1,2)` and `i(2,2)`

---

[28]`http://www.netlib.org/blas/`
[29]`http://sunsite.doc.ic.ac.uk/packages/netlib/blas/index.html`

Take, for example the Fortran array `integer ia(1000,1000)`. If you run through this array with the first index changing fastest, as in the following fragment

```
do 10,
            j=1,1000 do 20, i=1,1000 call some_calc (ia(i,j)) 20
            continue 10 continue
```

then you will move through the array in the 'natural' order. If, however, you process it in the other order, with the second index changing fastest, then when you move from `ia(i,j)` to `ia(i,j+1)`, the computer will have to jump 1000 places forward in the stored array, each time round the loop. This would not matter *too* much if the entire array were stored in physical memory, but this will not be the case for any sizable matrix, so that to find the appropriate array element, the machine may have to reload part of its memory from a saved version on disk. Such input is relatively slow, and if the calculation is laid out in such a way that this happens repeatedly in an inner loop, which is executed many times by an outer loop, then there can be a significant impact on your program's performance.

This system of swapping memory back and forth to disk is known as 'virtual memory', and the machine's discovery that the memory it needs is swapped out is known as a 'page fault', and is one of the important statistics about the run of your program. You can obtain such statistics in copious detail by using the compiler's profiler (see *Section 2.5.1*), or quickly by using the `time` command (see your system's `time` man-page for details -- not all versions provide this information by default).

Of course, it's not really as simple as just that. Even when the entire array *is* stored in physical memory, or when you have rearranged your program to minimise the number of page faults, it can be important not to hop about in arrays too much. Modern architectures will typically have a small amount (modern PCs around half a Mb, biggish servers around 8 Mb) of memory called 'cache' which has much faster access time (typically an order of magnitude or more) than the bulk of the RAM. The relationship between this and core RAM is strongly analogous to the relationship between core RAM and disk, in that cache-lines (which are like memory pages, but typically only a few words long) get swapped in and out of it when one word from the line is required. The upshot of that is that it's important to avoid hopping about over distances much smaller than an I/O page since, even if you have no page faults, cache faults make a huge difference. And of course it's not as simple as that either -- sometimes there are two levels of cache, some architectures implement some sort of prefetching, and so on. Except with some rather sophisticated profilers (which generally have to be supported by a processor architecture which keeps a record of these things) it's not really possible to see how many cache faults you're getting, except by writing the code more efficiently and seeing what the difference is. [MBT]

Unlike Fortran, C arrays are stored with the *rightmost* index increasing fastest, so that the array `int i[2][2]` will be stored as `i[0][0]`, `i[0][1]`, `i[1][0]` and `i[1][1]`.

2.5.2.3 I/O   Input and output are slow, and the faster processors become, the bigger is the cost of I/O relative to CPU cycles. That means you should think twice before writing out intermediate results, for reuse later in the same calculation or another. For example, it's quite easy to fall into the trap of 'saving time' by reading in a precalculated table of values, without realising that it can take more time to read the file than it would take to recalculate the table

from scratch each time. Remember that computers don't get bored, and don't mind doing the same calculation repeatedly.

If you do need to save values for later use, and don't mind doing it only semi-portably, you can save time and precision by using raw I/O rather than formatted ASCII [RW]. A C example is:

```
FILE *ofile; float arr[10];

/* set arr[i] to have sensible values */

ofile = fopen ("output-c.dat", "w");
fwrite ((void*)arr, sizeof(arr[0]), 10, ofile);
fclose (ofile);
```

Note that we carefully cast the variable `arr` to type `void*`, and that we could replace this by `(void*)&arr[0]` if we thought it was clearer. Note also the minor trick of using `sizeof(arr[0])` rather than the more obvious `sizeof(float)`; they are equivalent, but the first will remain correct even if we change our mind about the size of the elements of the array `arr`. You would read the contents of this file in using the function `fread`.

The `fwrite` and `fread` functions are not portable in general, since they deal with floating point numbers merely as bundles of bits, and pays no attention to how these are interpreted. This means that such files are not portable between machines which interpret these bits differently, so that files written on a little-endian machine (see *Section 2.3.2*) will be gibberish if read on a big-endian machine, and vice-versa. However, C's raw I/O is *semi*-portable, inasmuch as the `fwrite` function above does nothing other than copy 10×4 bytes (in this case) from memory to disk; `fread` similarly copies bytes from disk to memory without any interpretation.

The corresponding Fortran example is

```
      real arr(10)

C     set arr() to have sensible values
      open (unit=99,file='output-f.dat',form='unformatted')
      write (99) arr
      close (99)
```

Note that Fortran unformatted output is not portable in any way. Unlike C raw I/O, the Fortran compiler is free to store the information on disk in any way it likes, and the resulting file will not, in general, have 10×4 bytes in it. That is Fortran unformatted I/O is machine- *and compiler*-specific, and the file `output-f.dat` in this example will only be readable in general using a Fortran program built using the same compiler.

2.5.2.4 Use NaN and Infinity   As described in *Section 2.3.2.3*, any operation which has a `NaN` as input, produces a `NaN` as its result. Similarly, the special values positive and negative `Infinity` are legitimate numbers which can appear in a calculation. Consider the following code fragment:

```
      do 10, i=-5000,5000
            do 20, j=-5000,5000
                if (j .ne. 0) then
                    t(i,j) = real(i)/real(j)
                else
                    t(i,j) = 0.0
                endif
    20      continue
    10 continue
```

The `if` test is to avoid a division-by-zero error. If this loop were buried deep within a hierarchy of other loops, then the test could be the source of a significant fraction of the runtime. It can, however be omitted, allowing the matrix `t(i,j)` to include numerous `Infinity`s and one `NaN` (due to 0/0). Because both of these values are permissable floating-point operands, they can be allowed to percolate through the rest of the calculation without elaborate further tests, to be checked only at the end. This is possible only if the floating-point environment is set up *not* to generate signals on floating-point exceptions (see also *Section 2.3.2.3*).

Note that calculations involving the exceptional values tend to run more slowly than those using normal values, so if your calculation produces a significant number of exceptional values -- like the artificial example above -- a switch to IEEE semantics might not produce a speed win overall. Note also that the expression `(a.lt.b)` is *not* equivalent to `.not.(a.gt.b)`, since both `a.lt.b` and `a.gt.b` are false when one of the variables is an IEEE exceptional value: this could produce hard-to-find bugs if you were not alert to this when you were writing the code. [SG]

2.5.2.5 Remove debugging options It may or may not be obvious that debugging and profiling options, as discussed in *Section 2.5.3* and *Section 2.5.1*, will both slow your program down (especially in the latter case), and make it bigger. You should compile your program, or at least the numerically intensive parts of it, without them when you are not debugging it. [MBT]

2.5.2.6 Choose a fast compiler Random points:

- This is a case where free software is not necessarily better. Both Sun and Compaq expend resources on making their compilers as efficient as possible. If speed is important to your application, you're probably best off using one of these compilers.

- On the Alpha, it's worth while using the Fortran 90 compiler even for Fortran 77 code [SG].

2.5.2.7 Further reading Take a look at the Sun Numerical Computation Guide [sunncg], particularly the end of chapter 5, and at the collection of papers at <http://sunsite.doc.ic.ac.uk/sun/Papers/SunPerfOv+references/>, particularly the helpful paper about compiler switches: `you_and_your_compiler` (this is a generally useful collection of Sun papers, by the way).

If you *do* need to spend significant effort tuning your code, then you should consult a textbook on the matter. A good one is *High Performance Computing* [dowd]. This reference touches on parallelizing your code, a topic which I have deemed sufficiently arcane not to be discussed at all in this introductory guide.

2.5.3 Debugging

One approach to debugging is the brute-force method: simply scatter tracing statements through your code and watch them fill your screen. There are some types of debugging for which this is ideal -- for example, graphing a trace of intermediate values may reveal that an algorithm is showing signs of instability -- but it can be very cumbersome.

Better, in many cases, is to use a debugger. A debugger allows you to roam through your code, stopping in troublesome functions, examining data, and stepping through your code line-by-line. There is a great deal you can do with a debugger, but they are not often the easiest tools to master. However, there is a good deal you can do armed only with experience of the foothills of the debugger's learning curve.

I will describe the Sun debugger `dbx`, here. The use of the GNU debugger, `gdb`, and the `dbx` debugger on Digital Unix, are broadly similar.

First, compile `crash.c`, listed in *Appendix A.5*, in the usual way (`cc -o crash crash.c`), run it, and watch it crash when it tries to dereference a zero pointer. This leaves a `core` file in the current directory[30]. You can obtain rudimentary post-mortem information from this core file with `dbx`, as follows:

```
% dbx crash core
[...]
program terminated by signal SEGV (no mapping at the fault address)
(dbx) where
=>[1] bananas(0x0, 0x1, 0xef691338, 0x10074, 0x2, 0xeffff8d0), at 0x10870
[2] main(0x1, 0xeffff94c, 0xeffff954, 0x20800, 0x1, 0x0), at 0x108fc
(dbx) quit
```

The `where` command to `dbx` shows where the program was when it crashed (the `[...]` shows where I have omitted some unenlightening chatter from the debugger).

The other information `dbx` gives you is less useful -- the debugger doesn't know enough about your program to be more helpful. You can, however, tell the compiler to pass on more information to the debugger when it processes your code; do this with the `-g` flag to the compiler, as in `cc -g -o crash crash.c`. If you compile several modules to produce your executable, you'll need to give this option to the compilation of each module you wish to debug. You'll also have to give this option at the link stage on SunOS4, but not on Solaris or for `gdb` on Linux.

If we run `crash` again now, and look at the core file, we see that the debugger can provide us with more information.

```
% dbx crash core
[...]
program terminated by signal SEGV (no mapping at the fault address)
```

---

[30]If you don't get a core file, it might be that you have your shell set to prevent it -- possibly as a (reasonable) precaution to avoid filling up filespace with 'useless' core files. The command `ulimit -c` (`sh`-type shells only) will show the maximum size of core file: setting this to zero inhibits creating core files, and setting it to a very large number or to `unlimited` allows core files to be created. On `csh`-type shells, the corresponding command is `limit coredumpsize unlimited`

```
          Current function is bananas
          12              printf ("banana split: %d\n", *zp);
          (dbx) where
          =>[1] bananas(i = 0), line 12 in "crash.c"
          [2] main(), line 22 in "crash.c"
          (dbx) print i
          i = 0
          (dbx) print buf
          buf = "banana number 0
          "
          (dbx) print *zp
          dbx: reference through nil pointer
          (dbx) print zp
          zp = (nil)
          (dbx) quit
```

Note that we can examine the values of variables in scope at the point where execution stopped, and that dbx knows enough about C (or Fortran) to know the type of variables it is asked to print. Printing the value of the pointer zp shows us why our program has crashed.

The debugger is not only useful for such post-mortem diagnosis. It is also (for some people, primarily) used for investigating a program's behaviour when it is running normally, albeit with bugs we wish to track down. For example, start the debugger and tell it to stop when it enters the function bananas:

```
          % dbx crash
          [...]
          (dbx) stop in bananas
          (2) stop in bananas
```

You can set multiple breakpoints (so called) not only in functions, but also at line numbers, or linenumbers within files (using (dbx) stop at crash.c:12). Then tell the program to start running:

```
          (dbx) run
          Running: crash
          (process id 29707)
          Entering the bananas function: 1
          stopped in bananas at line 6 in file "crash.c"
          6       sprintf (buf, "banana number %d\n", i);
          (dbx) next
          stopped in bananas at line 7 in file "crash.c"
          7       if (i != 0)
          (dbx) print i
          i = 1
```

Note that, as well as output from the debugger itself, we also see output from the running program. If the program required input from the user, it would receive it as normal. The program stops at the first line of code within the bananas function. We can move one line forward in the code, verify that the parameter i has its expected value, then tell the program to resume execution.

```
(dbx) cont
banana number 1
No bananas left!
stopped in bananas at line 6 in file "crash.c"
6        sprintf (buf, "banana number %d\n", i);
(dbx) print i
i = 0
(dbx) cont
signal SEGV (no mapping at the fault address) in bananas
at line 12 in file "crash.c"
12           printf ("banana split: %d\n", *zp);
(dbx) quit
```

The Solaris debugger can be used through a GUI: give the command `debugger` to start this up. The Digital GUI debugger is `ladebug`. There is a GUI interface to `gdb`, called `xxgdb`.

In general, you can't debug optimized code (that is, code compiled with the option `-O`), because the optimizer may rearrange lines or remove redundant variables. However, both `dbx` and `gdb` do have support for debugging code with mild optimization, though there are some things, such as stepping from line to line, you might not be able to do.

This introduction has merely scratched the surface of what you can do with the debugger, by showing you the bare minimum of commands you need to navigate around your running program. There is a great deal more information available in Sun's debugging manual [sundebug], or in `gdb`'s info pages (`info gdb`).

2.5.4 Intermixing Fortran and C

Because there are so many reliable subroutine libraries written in Fortran, you will sometimes need to call a Fortran routine from C. Likewise, you might need to call a low-level C routine from a Fortran program.

For a detailed overview of such 'mixed language programming', see SUN/209, *CNF and F77 Mixed Language Programming*, which gives a detailed introduction to calling each language from the other, as well as a set of C macros to help support this. I will not duplicate the contents of that guide, but instead give a very compressed introduction to the problem. This might be enough to get you going. There is a further discussion of the issues, and a compressed description of the solutions, at the Cambridge High Performance Computing Facility (HPCF), at <http://www.hpcf.cam.ac.uk/mixed.html>. For a Sun-specific discussion, see Chapter 12 of [sunf77].

The biggest difference between C and Fortran is that 'C is call-by-value, Fortran is call-by-reference'. What that means is that when a C function is called, it receives the *values* of its arguments, so that any changes to them disappear when the function finishes, but when a Fortran function is called, it receives a *reference to* its arguments, so they can be altered easily within the function. The consequence of this is that when you call a Fortran function from C, you should pass arguments using C's address-of operator,`&`, and when you call a C function from Fortran, you will typically need to pass it the *value* of the variable using the `%val()` function (this is a non-standard VAX Fortran extension, but one which is now so ubiquitous that it's safe to use). These remarks apply to unstructured types such as characters, integers and floats

-- arrays and strings present other problems, as described below. It follows from what I've said that if a C function is declared as `void subr (int *p)`, it's expecting (the value of) a pointer to an integer, so that this could be called in the 'normal' way from fortran: `call subr (ia)`, where `ia` is an integer variable.

See *Appendix A.6* for example programs.

2.5.4.1 Arrays   Fortran's arrays are simple: an array of any dimension is just a list of locations in memory, stored in the order `a(1,1)`, `a(2,1)`, and so on (see *Section 2.5.2.2*); when a Fortran function is given an array argument, what it actually receives is a pointer to the 'top-left' element. If you're calling Fortran from C, you simply have to be aware of the switch on order, and then pass `&a[0][0]`.

If you have a C routine with a one-dimensional array argument (either `void func (int a[])` or `void func (int *a)`), and you want to call it from Fortran, you can call it simply by giving the array name as an argument: `call func (a)`.

Passing a multi-dimensional array *to* a C function is potentially problematic. However, you'll almost never need to do that, because Fortran very rarely needs to invoke C to do a numerical calculation. If the C declaration is `func (int a[][3])`, for example (that is, an array of three-element arrays), then a Fortran array `integer a(3,$n$)` could be passed simply, as `call func (a)`. If, on the other hand, the C declaration were `func (int **a)` (that is, a pointer to pointer to integer, with the actual array elements separately allocated), then the above Fortran array could *not* be passed as shown (and no, smarty, `call func (%loc(a))` wouldn't work, even though it's the right type). If you *do* need to call such a C function, you'll probably have to provide a suitable C wrapper for it, and call that from the Fortran code. C's array/pointer syntax is elegant, but not ideally suited for numerical work[31].

2.5.4.2 Strings   C strings are simple objects: they are by definition arrays of characters, with the end of the string marked by a zero byte. The internal structure of Fortran strings is not defined, so that they could be stored in whichever way is most convenient to the author of the compiler; typically, however, they are an array of characters with a length encoded with them. The practical upshot of this is that you simply cannot pass strings back and forth between Fortran and C code in a *portable* way, and it is fortunate that you rarely need to do this. On those occasions when you do need such a facility, you can use a library of conversion routines such as those described in SUN/209.

2.5.4.3 Compiling and linking   When a Fortran compiler produces object code, it *typically* adds an underscore to the end of each function name. That is, the subroutine:

```
subroutine func1 (i)
      integer i
      call func2 (i)
      end
```

---

[31]A little known factoid for C enthusiasts: did you know that C's array reference syntax is commutative, since `a[i]` is defined to be equivalent to `*(a+i)` and is thus equal to `i[a]`? This means that `a[3]` is the fourth element of the array `a`, and so is `3[a]`! Bizarrely enough, the latter is a legitimate array reference, but one you're probably best not including in your own code.

will produce object code with an external symbol `func1_` calling a subroutine named `func2_`. You must be aware of this when you compile Fortran code which is to be linked with C functions. There are two ways of dealing with this.

First, you can call this subroutine from C by calling it with the correct name:

```
int i1;
extern void func1_ (int *i);
/* call it */
func1_ (&i1);
```

So far so good. The problem arises when you want to call C from the Fortran function, since the Fortran function will expect to link against a function with a trailing underscore. If the C function is written by you, then you could provide this, but if it is a library routine, you will have to tell the compiler not to add the underscore to the external name when it generates object code. For Sun's `f77` compiler, you do this with the compiler option `-ext_names=plain`, for the GNU `g77` compiler it is with the `-fno-underscoring` option, and for `f77` on the Alpha, it is `-assume nounderscore` [RW]. Note that this will apply to all function names in that module. Sun's compiler also allows you to declare that a function is in a C module, using a pragma, but this is obviously non-portable, and so is not recommended. On this subject, [RW] points out that to get access to main() via f77 on decs, you need to set `-nofor_main` at link time.

You should, in general, use the Fortran compiler to link the object files into an executable. This calls the linker with all the correct Fortran libraries. It is of course possible to do the same with the C compiler, but requires a much more elaborate call.

2.5.5 Compilers, and other stray remarks on code

Using Fortran's `implicit none` keyword is a Good Idea. The tiny amount of typing time you save by relying on Fortran's automatic declaration of variables, is more than likely wiped out by the debugging time spent clearing up silly problems `implicit none` would have avoided from the beginning.

Similarly, in C, use ANSI-C rather than older 'K&R' C, and use prototypes religiously. That is, if you have a module `myfuncs.c`, then create a header file `myfuncs.h` containing the prototypes of the functions within it, and include it (`#include "myfuncs.h"`) both in any modules which use those functions and inside `myfuncs.c` as well. That way, if you change the interface to any functions in that module, the compiler will prompt you to change the calls to that function *everywhere* that it is used. This way, the compiler can help you avoid a whole class of embarrassingly silly problems.

Make your code portable. Unless you are specifically targetting your code at a particular (supercomputer) processor, or magic parallelizing compiler, you will save yourself time in the long run by not making assumptions about the machine or compiler environment. Especially given the imminent arrival of 64-bit machines, you are probably not doing yourself any favours by, for example, assuming that integers are four bytes. Probably more pertinently, exploiting compiler-specific features might prevent you running your code on some new faster architecture. If you do decide to use some non-portable features, you can make the future porting effort easier by having second or third thoughts about precisely how to use them, and by isolating them into a few subroutines, rather than scattering them throughout your code. It follows that....

Compiler Warnings Are Your Friend. Try compiling your codes with compiler warnings switched on (with the option `-Wall` on GNU compilers, `+w2` on Sun Workshop compilers, and `-w0` on Compaqs). You might be surprised at the number of peculiarities and non-standard features this exposes, each of which is a potential portability problem for the future. Don't treat compiler warnings as irrelvant nagging -- each of them is, to some extent, an error you have made, which has the possibility of turning round and biting you later. I would particularly advise this if you develop using GNU compilers, as these seem particularly liberal about standards: `gcc` in particular seems happy to take any old nonsense, say 'I know what you mean', and produce code from it -- in my experience it usually guesses my intentions correctly, but I don't want to rely on it. You can also adjust the strictness of the C compiler's conformance to the ANSI-C standard by using compiler options (Sun: `-Xa` and `-Xc`; DEC: `-std1`; GNU: `-ansi` and `-pedantic`). Having said all this, I don't want to overstate the importance of compiler warnings: the main plank of this advice is to spend time now to save time and aggravation later, but this tradeoff is unlikely to be in your favour if you spend time removing every last anomaly from your code.

2.6 Link farms

At the end of each section of this cookbook, I'll include a collection of web-based resources you can use to search for further information. These will typically be either FAQs (lists of 'frequently asked questions') or 'link farms' (thematic collections of links with little further detail).

2.6.1 Unix documentation and standards

- 'POSIX[32] is the term for a suite of applications program interface standards to provide for the portability of source code applications where operating systems services are required. POSIX is based on the UNIX (Registered trademark administrated by the Open Group) Operating System, and is the basis for the Single UNIX Specification from The Open Group.' [IEEE PASC (Portable Application Standards Committee)]

- This is currently (2001-06-20) undergoing revision by the Austin Group[33]

- The POSIX standard is

  1. 'IEEE Std 1003.1 Standard for Information technology -- Portable Operating Systems Interface (POSIX) -- Part 1: System Interface [C language binding]', which is identical to 'ISO/IEC JTC1 IS 9945-1 Standard for Information technology -- Portable Operating Systems Interface (POSIX) -- Part 1: System Interface [C language binding]'

  2. 'IEEE Std 1003.2 Standard for Information technology -- Portable Operating Systems Interface (POSIX) -- Part 2: Shell & Utilities', which is identical to 'ISO/IEC JTC1 IS 9945-2 Standard for Information technology -- Portable Operating Systems Interface (POSIX) -- Part 2: Shell & Utilities'.

  ...and these cost money (lots[34]) to buy on paper.

---

[32] http://www.pasc.org/abstracts/posix.htm
[33] http://www.opengroup.org/austin
[34] http://standards.ieee.org/catalog/olis/licenses/licenses.html

- Single Unix specification[35], from the Open Group (other publications[36]). My understanding from the PASC document above is that Single Unix is broader than POSIX, and covers more of the environment (bit vague, here). It feeds into the POSIX/ISO standardisation process, but is not a formal standard itself. Single Unix does, however, have the great advantage of being available online.

- The link with The Unix System[37] is, I fear, rather obscure to me, but it appears that the latter is the public face of the Open Group's labours.

`<http://www.geek-girl.com/unix.html>`: The 'Unix Reference Desk'. This is a very useful collection of pointers to unix documentation.

`<http://www.faqs.org/faqs/unix-faq/faq/>`: the General Unix FAQ, full of arcana. This is one of several FAQs which cover Unix and related issues; for some others, look at `<http://www.faqs.org/faqs/by-newsgroup/comp/comp.unix.questions.html>`

`<http://uk.yahoo.com/Computers_and_Internet/Software/Text_Editors/vi/>`: Collection of links to vi documentation at Yahoo.

---

[35]`http://www.opengroup.org/onlinepubs/007908799/`
[36]`http://www.opengroup.org/publications/catalog/web.htm`
[37]`http://www.UNIX-systems.org/`

3 Theory support

3.1 Computer algebra

Starlink provides access to computer algebra by supporting the Maple package. You might have access to Maple on your own Starlink node, but if not, you may use it on the machine `star.rl.ac.uk`, if you have an account there. If you are a Starlink user, you should apply for an account by mailing `star@star.rl.ac.uk`.

Maple allows you to enter mathematical expressions using a fairly natural syntax, substitute into them, simplify them, differentiate and (with limits) integrate them, and finally go on to graph them. As an added bonus, you can also produce output in C, Fortran and LaTeX.

For example, consider the following example.

```
star:nxg> maple
     |\^/|      Maple V Release 4 (Rutherford Appleton Laboratory)
._|\|   |/|_. Copyright (c) 1981-1996 by Waterloo Maple Inc. All rights
 \  MAPLE  /  reserved. Maple and Maple V are registered trademarks of
 <____ ____>  Waterloo Maple Inc.
      |        Type ? for help.
> gi := amp * exp(-(xparam^2/sa^2)/2);
                                             2
                                        xparam
                      gi := amp exp(- 1/2 -------)
                                             2
                                          sa
```

We start up Maple, and enter an expression for a gaussian. Maple makes an attempt to display the result intelligibly. If we had ended the expression with a colon rather than a semicolon, Maple would have suppressed the display. Note that undefined variables represent themselves, and that Maple knows that `exp` is the exponential function, so that it knows, for example, how to differentiate it.

Now define the variable `xparam`, and redisplay `gi`.

```
> xparam:= cos(theta)*(xc-x0);
      xparam := cos(theta) (xc - x0)

> gi;
                                        2           2
                              cos(theta)  (xc - x0)
                  amp exp(- 1/2 ---------------------)
                                          2
                                        sa
```

Then differentiate the gaussian, and assign the result to `gid`. The result is something you're happy not to have had to work out yourself.

```
> gid := diff (gi, theta);

                                                              2          2
                                                      cos(theta)  (xc - x0)
                 2                              exp(- 1/2 ---------------------)
       amp cos(theta) (xc - x0)  sin(theta)                        2
                                                                  sa
gid := -------------------------------------------------------------------------
                                           2
                                          sa
```

If that the purpose of this was to do a calculation somewhere, you might want to code this expression in Fortran. Doing this by hand would be error-prone, but Maple can produce output in Fortran as well as this 'prettyprinted' style.

```
> fortran (gid,optimized);
     t1 = cos(theta)
     t4 = (xc-x0)**2
     t6 = sa**2
     t7 = 1/t6
     t10 = t1**2
     t15 = amp*t1*t4*t7*sin(theta)*exp(-t10*t4*t7/2)
```

The `optimized` argument tells Maple to try to produce Fortran code without repeated subexpressions. You can save this to a file with the expression `fortran (gid, filename='gaussian.f', optimized);` You can produce output in C as well, though because the identifier`C` is potentially such a common one, you must explicity load the C library first.

```
> readlib(C):
> C([gf=gid],optimized);
     t1 = cos(theta);
     t4 = pow(xc-x0,2.0);
     t6 = sa*sa;
     t7 = 1/t6;
     t10 = t1*t1;
     gf = amp*t1*t4*t7*sin(theta)*exp(-t10*t4*t7/2);
```

There are two things to note here. The first is that we have renamed the expression `gid` on the fly. The second is that the expression for `t4` is not the most efficient -- it is very bad to use the `pow()` function for raising expressions to small integer powers: much better would be `t4a=xc-x0; t4=t4a*t4a;`, as has happened automatically for `t10`.

You can also produce results in LaTeX

```
> latex(gid);
{\it amp}\,\cos(\theta)\left ({\it xc}-{\it x0}\right )^{2}\sin(\theta
){e^{-1/2\,{\frac {\left (\cos(\theta)\right )^{2}\left ({\it xc}-{
\it x0}\right )^{2}}{{{\it sa}}^{2}}}}}{{\it sa}}^{-2}
```

Maple has done the correct thing with the cosine and sine functions, and with the θ variable, and it has got *all* the braces matching correctly, but it has expressed the exponential as a simple e-to-the-power which will look rather ugly (as well, the exponential should be written with \mathrm{e}).

Leave Maple by giving the command `quit`.

SUN/107 provides an introduction to Maple, and SGP/47 is a comparison of Maple and Mathematica. Also, the Maple manual and tutorial are very clear. There is help within Maple (type `?intro`), and this gives enough information to get you going. Maple's web pages are at `<http:/ /www.maplesoft.com/>`, but they don't currently (December 1998) give a lot of tutorial help. See also the example Maple program in *Appendix A.7.*

There's also a GUI for maple, which you can invoke with `xmaple`.

As a final point, don't fall into the common trap of thinking that because you've produced your result using computer algebra, it must be right. This is as false of computer algebra as it is of numerical programming -- be inventive in thinking of cross-checks.

3.2 Data visualisation

Starlink supports two data visualisation packages, IDL and DX, but for relatively simple graphing of simple results, gnuplot will probably produce acceptable results in rather less time.

The package SM, which is a descendent of Mongo, has numerous adherents, but I don't propose to discuss it, partly because I've never used it, partly because it's not part of Starlink's base set and so is not available at all sites, but mostly because if gnuplot runs out of steam, you might as well go straight to IDL, which is easily available through Starlink.

SG/8, *An Introduction to Visualisation Software for Astronomy*, is an overview of visualisation systems, which mentions both IDL and DX.

3.2.1 gnuplot

Gnuplot is valuable because it's so simple -- easy things can be done easily. If you want to graph an equation or plot some data, and produce postscript output, then you'll probably do it faster, from a standing start, than someone using one of the beefier packages. Its weaknesses are that it can't easily do very complicated or fancy visualisations (that is, it doesn't try to take over the world), and it deals naturally only with ASCII data in columns. It is scriptable, but I wouldn't fancy programming anything complicated with it.

Start it up with the simple command `gnuplot`. Once it's started, you can give it commands as simple as

```
gnuplot> plot sin(x)
```

to produce a plot of the sine function with default ranges, or you can give it a range and specify a line type as follows

```
gnuplot> plot [x=0:3.14] sin(x)*cos(x**2)**2 with impulses
```

Gnuplot can also graph data files. The example file `gausssine.dat` consists of two columns of 64×64 values. It can be plotted in gnuplot with the commands:

```
gnuplot> set output 'gausssine-gnuplot.eps'
gnuplot> set terminal postscript eps
Terminal type set to 'postscript'
Options are 'eps monochrome dashed "Helvetica" 14'
gnuplot> splot 'gausssine.dat' using 1 with lines
gnuplot> set terminal x11    # set the terminal type back to the (default) X
gnuplot> set output          # close the output file
```

This produces the EPS file shown in Figure 1. The file has a blank line after each block of 64 numbers. Gnuplot interprets this as a signal that this is the end of a 'row' of a matrix (a standard gnuplot gotcha is that is must be a *completely* blank line, with no whitespace). You can plot the surface represented by the second column, with the clause `using 2` to `splot`. Gnuplot can read more generally formatted data, but that's already stepping towards advanced usage.

Figure 1: File gausssine.dat, displayed with gnuplot

There is good help within gnuplot, available by typing `help`.

3.2.2 IDL

IDL is much more powerful and flexible than gnuplot, and has a correspondingly longer learning curve. It's never been accused of being elegant, but with only a bit of headbanging, I've always managed to get it to do what I wanted (I've always seen it as reminiscent of Fortran in this respect).

Several missions have used IDL as the language in which they have written their data-analysis software, and Starlink is currently experimenting with providing IDL interfaces to important Starlink applications, which is possible because IDL can link to codes written in languages such as Fortran or C. IDL is moving towards being a core facility at Starlink sites.

IDL displays data in arrays as part of a generic set of array manipulations. For example, the data in the file `gausssine.dat` was produced by the following sequence of IDL commands:

```
x=(findgen(64)-32)^2
d=fltarr(64,64)
for i=0,63 do d(i,*)=sqrt(x+(i-32)^2)
a3=exp(-(d/15)^2)
s=sin(findgen(64)/63*5*3.141592657)
s2=s#s
m=s2*a3
```

The function `findgen(N)` returns a float array (indexed from 0 to $N-1$) in which the value of each element is the same as its index; performing an arithmetic operation such as subtraction or exponentiation on an array performs it on each element of the array; `fltarr` declares an array of floats of the specified dimensions; the array reference `d(i,*)` refers to the entire `i`'th row of

d; the operator `#` forms the direct product of the two vectors. The result of this is to set d to be an array where each element is the euclidean distance from element `(32,32)`[38].

Once we have the data in the array `m`, we can produce a surface plot with `surface,m`, and then go on to annotate it, rotate it, shade it, contour it, with the large collection of options and parameters to the `surface` command. We can produce PostScript output with the following sequence of commands:

```
!p.font=0              ; use postscript fonts rather than IDL outlines
  set_plot, 'ps'       ; use the postscript output driver
  device, /encap, filename='gausssine-idl.eps'
                       ; produce encapsulated postscript
  surface, m
  device, /close       ; close the file
```

This produces the EPS file shown in Figure 2. A minor, but persistent, irritation with IDL is that, although its input and output facilities are as flexible as, say, Fortran's (which they closely resemble), it doesn't come with a function for dealing with the common case of data printed out in columns (the only case gnuplot naturally deals with). Fortunately, such a function is not only easy to write, but a useful example, too. See *Appendix A.8*.

Figure 2: File gausssine.dat, displayed with IDL

IDL comes with rather good manuals, the reference parts of which are available on-line by typing `?` at the IDL prompt.

See appendix B of SUN/55 for a description of how to import data in the Starlink NDF format into IDL.

3.2.3 DX

IBM's Data Explorer is also available on some Starlink machines. I don't have personal experience of it, but there is a Starlink manual for it in SUN/203, *SX and DX -- IBM data explorer for data visualisation* and a Starlink DX cookbook in SC/2.

In his overview of visualisation systems in SG/8, Clive Davenhall says of DX:

> IBM Data Explorer (DX) is a general-purpose software package for data visualisation and analysis. It employs a data-flow driven client-server execution model and provides a comprehensive range of data manipulation, visualisation and display functions. Visualisations can be generated using a visual programming editor or a text-based scripting language. DX is the visualisation package recommended by Starlink, particularly for three-dimensional scalar and vector data. Starlink has produced a set of enhancements to DX. If you are using DX at a Starlink site then these enhancements should be available automatically. The use of DX at Starlink sites and the Starlink enhancements to DX are documented in SUN/203.

---

[38]IDL experts will know that the common IDL idiom for this is `d=shift(dist(64),32,32)`, but have you ever actually compared `dist` with its documentation? The documentation for `dist` suggests that this idiom wouldn't work, but the function's actual behaviour seems to (substantially) depart from the claimed behaviour in exactly the right way.

3.2.4 PGPLOT

The above recommendations describe standalone packages which work on data produced by your code in a separate step. An alternative route is to incorporate the plotting facilities within your program, and the recommended way of doing this is by using the PGPLOT library.

The library, which was written to support astronomical applications, consists of a collection of high-level routines for producing plots, maps and images either on-screen or as Postscript to a file. Refer to SUN/15, *PGPLOT --- Graphics Subroutine Library* for further details, or to the PGPLOT home page at <<>>.

Note that there are *two* versions of PGPLOT currently available on Starlink, 'native' PGPLOT, and a Starlink version which uses GKS. The latter is being deprecated, with a view to being ultimately phased out, and this will affect how you link your program against the library. At the time of writing (December 1998), the way in which the dual versions will be supported has not been finalised; ask your system manager for advice.

3.3 Producing images

If you wish to include images in your LaTeX output, do so using the standard graphics package. That is, include in your file the command `\usepackage{graphics}` (if you're obliged to use the old LaTeX2.09, you can use the `epsf` option to the document style). Include the graphics with the command `\starincludegraphics{file.eps}`. So how do you produce the postscript?

An important point is that the postscript should be *encapsulated* postscript. This is postscript intended to be incorporated within another document: it has a `BoundingBox` comment at the top of the file, and typically has the extension `.eps`.

See *Section 3.2* for details of how to produce EPS plots from gnuplot and IDL.

If it's diagrams you want to produce, then `xfig` has its adherents. There's a *large* manual page for `xfig`, but you can do pretty well just starting it up, and pointing and clicking.

If point and click isn't your style, try MetaPost[39]. This is a variant of Knuth's MetaFont (which is used for designing TeX fonts), which produces postscript instead. To produce a document using MetaPost, you produce a text file specifying the objects you want to draw and their spatial relationships. It can be hard work, but the results can look very good. If you wished to automate producing diagrams, perhaps because you want to produce a set of diagrams which are systematically different, then MetaPost could very well help you with this. See `.../texmf/doc/metapost` under the (Starlink) TeX tree for further details.

---

[39]`http://cm.bell-labs.com/who/hobby/MetaPost.html`

4 Astrophysical and other libraries

4.1 Astrophysical modelling codes

This section aims to be no more than a set of pointers to more complete information and resources. I'm no expert in this field, myself, so if I've missed some important resource, or miscontextualised something, please do let me know.

Stellar atmosphere codes:

- CCP7[40] was a Collaborative Computational Project[41], 'concerned with the calculation of theoretical models suitable for the interpretation of stellar and interstellar spectra'. As well as promoting the use of advanced computers in astronomy, it developed a library of stellar-atmosphere and spectroscopy codes. The project finished in April 2001, but the resources it produced are still useful.

- Simon Jeffery[42] makes available a number of tools for stellar atmosphere calculations.

- At Robert Kurucz's home page[43] you can obtain copies of his ATLAS and other codes, along with grids of model atmospheres.

- PHOENIX[44]: state-of-the-art M-star model atmospheres.

Atomic and molecular data:

- The Opacity Project[45] is a collaborative project to produce the atomic data required for stellar envelope calculations. The project makes the results generally available both through a database called 'TOPbase', and by anonymous FTP.

- Other opacity data is available from the OPAL code[46]. As well as making precalculated opacity tables available, you can make requests for new tables to be generated.

- Atomic data at Kentucky[47] and Harvard[48]

- JPL Molecular Spectroscopy[49]

- The Vienna Atomic Line Database (VALD)[50] is an excellent site with up-to-date atomic data needed for spectrum synthesis.

Astronomical chemistry (loosely) -- dust, clouds and the interstellar medium:

---

[40]http://ccp7.dur.ac.uk/
[41]http://www.dci.clrc.ac.uk/ListActivities.asp?Class=5;Classtype=21;
[42]http://star.arm.ac.uk/~csj/
[43]http://kurucz.harvard.edu/
[44]http://phoenix.physast.uga.edu/
[45]http://vizier.u-strasbg.fr/OP.html
[46]http://www-phys.llnl.gov/V_Div/OPAL/
[47]http://www.pa.uky.edu/~verner/atom.html
[48]http://cfa-www.harvard.edu/amdata/ampdata/amdata.html
[49]http://spec.jpl.nasa.gov/
[50]http://www.astro.univie.ac.at/~vald/

- Cloudy[51] is a arge-scale plasma simulation code that is widely used across the astronomical community as an aid in the interpretation of spectroscopic data.

- DUSTY[52] is a dust-shell modelling code, also originating at Kentucky.

A number of CFD codes[53] are available to buy or download. This is a collection of pointers and notes about a large number of CFD codes, from a variety of sources.

Thanks to [SJ], [BS] and [AS] for much of the content of this section.

4.2 General-purpose and numerical libraries

There are numerous software libraries available. Many of these are free, many are public domain. The difference between the two is that 'public-domain software' is in some sense 'ownerless': it's yours to play with, or modify, or exploit, as you wish, with only good manners requiring you to acknowledge the source (this is not an authoritative statement of copyright law, by the way...). 'Free' software, on the other hand, is still copyrighted, but you may have a free licence to use it for certain purposes. One of the best known of these licences is the GNU General Public Licence[54], which gives you very good access to the code, limited only by the requirement that programs produced using GPL'd code are themselves made as freely availale. Other licences might make the code available to academic users only, or only for non-commercial use. The GNU project have a useful collection of free and not-so-free licences[55], which is useful even though they are sometimes a little fervent about the issues. The type of licence makes a difference if you plan to redistribute your code. See also the observations about libraries, and thoughtless use of them, in *Section 2.3*.

Probably the most commonly used numerical library is the NAG library[56]. This is a long-established, and very highly thought-of, library, which contains codes to deal with a broad range of numerical problems. The routines tend to come in several versions, typically a highly general, and highly configurable, one, accompanied by easy-to-use drivers. The NAG library is expensive, but Starlink's size allows it to negotiate with NAG to provide the library at all Starlink sites. The routines are generally in Fortran, but C versions of some of them are becoming available.

The PDA library is a collection of public-domain and free routines, assembled by Starlink, and intended to replace the NAG library in Starlink application code. The collection was assembled using GAMS, with routines drawn from FFTPACK, SLATEC, NMS, OPT, plus other isolated sources. If you need to use one of the available algorithms, then the advantage of using the library version is that the (possibly non-trivial) work of building it for your architecture has already been done, leaving you able to simply link against the library. The collection is fully documented in SUN/194.

The remaining libraries are typically free.

In your search for codes, you would do well to start at GAMS[57]: Guide to Available Mathematical Software. This describes itself as 'A cross-index and virtual repository of mathematical and

---

[51]http://www.pa.uky.edu/~gary/cloudy/
[52]http://www.pa.uky.edu/~moshe/dusty/
[53]http://www.fges.demon.co.uk/cfd/CFD_codes.html
[54]http://www.gnu.org/copyleft/gpl.html
[55]http://www.gnu.org/licenses/license-list.html
[56]http://www.nag.co.uk
[57]http://gams.nist.gov

statistical software components of use in computational science and engineering'. You can either search for the code you need by keyword, or work through their classification of problems (a classification which is occasionally used more widely) to find references. They point to both free and commercial software.

The first collection to be aware of is Netlib. The archive is based at `<http://www.netlib.org>`, but there are UK mirrors[58] Although there are some facilities which are primarily available at Netlib, it also mirrors several other archives.

NHSE[59], the (US) National High-Performance Software Exchange, is 'a distributed collection of software, documents, data, and information of interest to the high performance and parallel computing community'. NHSE is part of Netlib, and incorporates software repositories HPC-netlib[60] for high-performance software, PTlib[61] for parallel tools, and CSIR[62] for chemistry software.

The CERN program library[63] includes CERNLIB[64], which consists of a number of component libraries[65]. This is a long-standing and well-known library of general-purpose Fortran (typically) numerical routines with, obviously, some bias towards particle physics. The service is free to all HEP users, and to physics departments in CERN member states, with separate non-commercial and commercial rates available. There is a FAQ[66].

JPL has a Computational Mathematics Library[67]. These appear to be free, but no longer formally supported by JPL.

The ACM Transactions on Mathematical Software[68] (TOMS) is a journal produced by the ACM. The software discussed in there is available at GAMS, and mirrored at HENSA[69].

The Scientific Computing Division[70] of the (US) National Center for Atmospheric Research[71], has an overview of mathematical and statistical packages[72]. Not all the packages reviewed there are freely available, but the discussions are useful.

PORT[73] is a collection of general maths subroutines. Its description of itself is on the front page: 'The PORT Mathematical Subroutine Library (third edition) is a collection of Fortran 77 routines that address many traditional areas of mathematical software, including approximation, ordinary and partial differential equations, linear algebra and eigensystems, optimization, quadrature, root finding, special functions, and Fourier transforms, but excluding statistical calculations. PORT stands for Portable, Outstanding, Reliable, and Tested.' Some routines are public-domain when, for example, they are developments of public routines, others have a non-commercial-use licence condition.

---

[58]`http://www.netlib.org/bib/mirrors.html`
[59]`http://www.nhse.org/`
[60]`http://www.nhse.org/hpc-netlib/`
[61]`http://www.nhse.org/ptlib/`
[62]`http://www.csir.org/`
[63]`http://wwwinfo.cern.ch/asd/index.html`
[64]`http://wwwinfo.cern.ch/asd/cernlib/overview.html`
[65]`http://wwwinfo.cern.ch/asd/cernlib/libraries.html`
[66]`http://wwwinfo.cern.ch/asdcgi/listcernlibfaqs.pl`
[67]`http://math.jpl.nasa.gov/`
[68]`http://math.nist.gov/toms/Overview.html`
[69]`http://www.hensa.ac.uk/netlib/toms/`
[70]`http://www.scd.ucar.edu/`
[71]`http://www.ncar.ucar.edu/`
[72]`http://www.scd.ucar.edu/softlib/mathlib.html`
[73]`http://www.bell-labs.com/project/PORT/`

Slatec[74] is 'a comprehensive software library containing over 1400 general purpose mathematical and statistical routines written in Fortran 77.'

Specifically concerned with minimisation, minpack[75] (or at netlib[76]), is a library for solving linear and nonlinear minimisation problems. It has documentation within the source code.

4.2.1 Reading and writing data

The codes you write do not exist in isolation, and at some point you will have to read data from, or write it to, files. You might, therefore, need to read or write one of a number of standard data formats. Clive Davenhall wrote an article on this in the September 1998 issue of the Starlink Bulletin[77]. This covered reading and writing using the IMG library (SUN/160), using NDF files and the HDS files they are a special case of (SUN/33 and SUN/92 respectively), and reading and writing FITS files (SUN/136).

You can convert between different data formats using the CONVERT package, documented in SUN/55. CONVERT is extremely easy to use, and converting a FITS file, say, to an NDF is as easy as

```
fits2ndf myfile.fits myfile
```

If you have to read or write FITS files, then visit the FITS home page[78] for the FITS users guide and the `cfitsio`[79] library. Although FITS files have a very simple format, there are enough ways of getting things wrong that you will, as usual, save yourself time in the long run by taking the trouble to use the `cfitsio` library. It's easier than you might think: the Quick Start Guide contains most of what you'll ever need to know. Note that, although the library is called *c*fitsio, it's designed to be used with Fortran as well, and the programmer's reference guide comes in two flavours.

Once the data is there, you will need to visualise it. See *Section 3.3* for some pointers to suitable software.

4.3 Link farms

<http://cdsweb.u-strasbg.fr/astroweb.html>: AstroWeb is one of the most useful link collection, because it has been developed by the astronomy community.

<http://math.nist.gov/toms/Resources.html>: The TOMS list of web resources for mathematical software is a collection of pointers maths software on the web.

Yahoo has managed to assemble a respectable collection of pointers to Scientific software at <http://uk.yahoo.com/Computers_and_Internet/Software/Scientific/>, as well as more specific pointers to maths (<http://uk.yahoo.com/Science/Mathematics/Software/>), physics (<http://uk.yahoo.com/Science/Physics/Software/>) and astronomy (<http://uk.yahoo.com/Computers_and_Internet/Software/Scientific/Astronomy/>) resources.

---

[74]http://www.hensa.ac.uk/netlib/slatec/index.html
[75]http://www.hensa.ac.uk/netlib/minpack/
[76]http://www.netlib.org/minpack/
[77]http://star-www.rl.ac.uk/bulletin/98sep/node12.html
[78]http://fits.gsfc.nasa.gov/
[79]http://heasarc.gsfc.nasa.gov/docs/software/fitsio/fitsio.html

5 Paper production

5.1 TeX and LaTeX

Starlink supports LaTeX and TeX users by maintaining and packaging for release a TeX distribution. This is typically based on one of the standard TeX distributions, plus an effort to ensure that the distribution includes packages of interest to astronomers, particularly some of the relevant journal style files. See the Starlink LaTeX page[80] for details.

LaTeX[81] is nominally (though not actually) re-released every six months, in June and December, with each release incorporating bugfixes, but no significant development. New features are being incorporated into LaTeX3, which is a major upgrade, still being developed. Versions more than a year old (that is, more than two releases ago) are formally 'obsolete', and bug reports won't be accepted for them.

The current version of LaTeX is also known as LaTeX2e, to distinguish it from the by now completely obsolete LaTeX2.09, which is the version of LaTeX described in the first edition of Lamport's book. LaTeX2e has significant internal differences from LaTeX2.09, but was intended to appear much the same to the user. The most prominent difference is that LaTeX2e files start with the declaration `\documentclass[options]{classname}`, and invoke further packages using `\usepackage{packagename}`, whilst LaTeX2.09 files start `\documentstyle[options]{stylename}`, with the options list being a mixture of style file options and package names. LaTeX2e will attempt to go into a 'compatibility mode' if it sees a file start with `\documentstyle`, but this isn't terribly reliable, and you *certainly* shouldn't create new files like this, unless some primitive publisher (which used to include MNRAS until rather recently) absolutely insists on it.

Other LaTeX resources you might want to examine are the UK TeX FAQ[82] (this is *very* good), CERN's TeXpages[83], and the catalogue[84] at CTAN[85] (the 'Comprehensive TeX Archive Network', hosted on three peer machines in the UK, Germany and the US, and mirrored worldwide; all TeX is there).

All this, of course, assumes that you're already a LaTeX user. If you're just starting with LaTeX, then the canonical place to start is with Leslie Lamport's *LaTeX: a Document Preparation System, 2nd edition* [lamport]. I think this is a good introduction, because it concentrates on the basics, and leaves the elaborate details to others. The bulk of the book is an accessible introduction to LaTeX (and note that you really ought to avoid asking LaTeX questions until you've read chapter 2); appendix C is a reference manual.

If it's the elaborate details you want, then you'll need to supplement Lamport. Victor Eijkhout's TeX by Topic[86] is well spoken-of, though I haven't examined it myself (the book is now out of print, but the author has made it available for free, for a donation). Two other good books to examine are *A Guide to LaTeX* [kopka], and *LaTeX Line by Line* [diller]. Both are substantially more advanced than Lamport, and cover a lot of material densely but reasonably clearly. If forced to choose, I would go for Kopka and Daly over Diller, partly because they have produced

---

[80]`http://star-www.rl.ac.uk/%7eacc/tex/tex.html`
[81]`http://www.latex-project.org/`
[82]`http://www.tex.ac.uk/faq`
[83]`http://wwwinfo.cern.ch/asdoc/textproc.html`
[84]`http://www.tex.ac.uk/tex-archive/help/Catalogue/catalogue.html`
[85]`http://www.tex.ac.uk`
[86]`http://www.eijkhout.net/tbt/`

a second edition covering LaTeX2e, but also because Diller seems to try to pack in too much, including some material (such as the intricacies of maths typesetting) which, if you're going to learn, you should probably learn from Knuth's TeXbook. Having said that, the advantage is a narrow one, and you'd be well-advised to choose based on whose writing style you prefer, and which one happens to be in the bookshop on the right day[87]. Both of the last two should be kept away from anyone not already convinced of LaTeX's virtues, since they both make LaTeX seem much more forbidding than it actually is. Finally, Goossens, Mittelback and Samarin's *The LaTeX Companion* [goossens] is useful, but intended as a reference, not as an introduction. For other print books, and details of these ones, see the TeX FAQ's bibliography at `<http:/ /www.tex.ac.uk/cgi-bin/texfaq2html?label=books>`, and Adam Lewenberg's collection of TeX and LaTeX print resources[88].

There are a number of online tutorials[89]. Both the 'Gentle Introduction[90]' and the 'Not so Short Introduction[91]' are available at CTAN. Peter Flynn's 'Beginner's LaTeX[92]' is well written and informative, and spends a good deal more time on the preliminary technicalities than others; but -- since it grew out of a two-day introduction to LaTeX aimed primarily at humanities users -- it does not cover maths.

If you prefer learning by example, take a look at the standard sample files, `small2e.tex` and `sample2e.tex`. These will be somewhere in your LaTeX distribution, typically with a path ending in `.../tex/latex/base/small2e.tex` (try `echo $TEXINPUTS` or `kpsepath tex` to help locate the TeX distribution on your local machine).

The book on TeX is Knuth's original, *The TeXbook* [knuth]. As well as describing the underlying TeX engine, this also describes Knuth's very basic macro package, `plain`. You need the TeXbook if you're writing a LaTeX macro package, if you demand complete control over positioning (tricky in TeX, but an out-and-out hassle in LaTeX), or if you just don't like the way LaTeX lays things out and want to do it all yourself. If you don't fall into any of those categories, `plain` TeX is probably not where to start (and I speak as a lapsed TeXie who has spluttered furiously at Leslie Lamport's failure to impose a satisfactorily rigourous line on even such fundamental doctrinal matters as how the word 'LaTeX' is to be pronounced).

You should be aware of `pdflatex`, which is a version of TeX which produces PDF files directly rather than DVI files (though note that versions before `1.0-prerelease` produce bad PDF which breaks Windows Acroread 5 at least).

Starlink has produced several documents concerned with LaTeX. The LaTeX user's guide, SUN/9, concentrates on the practical details of LaTeXing your document and ultimately transforming it into PostScript. You should also refer to SC/9, the LaTeX cookbook. However, SGP/28, *How to write good documents for Starlink*, SUN/199, *Star2HTML* and SGP/50, *Starlink Document Styles* are primarily intended for those writing Starlink documentation.

---

[87]It's unfortunate that neither book is much of an advert for TeX's potentially beautiful typesetting -- both seem to be produced using a practically unmodified LaTeX `book` style.

[88]`http://www.macrotex.net/texbooks/`
[89]`http://www.tex.ac.uk/cgi-bin/texfaq2html?label=tutorials`
[90]`http://www.tex.ac.uk/tex-archive/info/gentle/`
[91]`http://www.tex.ac.uk/tex-archive/info/lshort/english/`
[92]`http://www.silmaril.ie/downloads/documents/beginlatex.pdf`

5.2 The LANL archive

The LANL preprint archive is based, in the UK, at `<http://xxx.soton.ac.uk>` which is a mirror of the master archive[93] at LANL. The archive is easy to use as a reader but not, unfortunately, transparently easy to use as an author (not helped by the rather snide error messages that come back if you slip up...).

The full instructions[94] are comprehensive, but boil down to the following:

- The first time you submit anything to the archive, you need to register as an author. That registers your email address with them and sets a few preferences (such as the default archive you'll submit to).

- Bundle up the LaTeX source for your paper in a `.tar.gz` archive. The automatic-processing software at LANL seems to LaTeX every `.tex` file in sight, so if your submission isn't in the canonical format of one TeX file plus a bunch of PostScript figures, you'd probably best read the instructions one more time. It follows from this that it doesn't matter what you call your TeX file: the processing software will still TeX it.

  LANL has copies of journal styles such as the A&A one -- when they process your paper it'll be formatted according to that style file. They suggest that the PostScript figures in the article be numbered something like `figure1.ps`, `figure2.ps`, so that they alphabetise correctly. See *Considerations for TeX submissions*[95] for the gory details.

- Read the uploads help again, particularly the information on the information fields you'll have to fill in, then go to the uploads page at `<http://xxx.lanl.gov/uploads>`.

- The uploads page gives you a form to type in the various details of author and title and the like, and to type in the abstract. You give the name of your tar file, press the button and wait. After a pause you can check if the automatic processing succeeded. At this stage, you'll be given a password for this paper. This allows you and your co-authors to retrieve the paper before it's put in the public archive, and will be needed to make alterations, or update the journal publication details, in future.

- Now, you need a drink.

---

[93]`http://xxx.lanl.gov`
[94]`http://xxx.soton.ac.uk/help/`
[95]`http://xxx.soton.ac.uk/help/submit_tex`

6 General astronomical and astrophysical resources

The Astrophysics Data System[96] (ADS) is a NASA project, providing an abstract service, along with a subsidiary data service. Further to this, SUN/174, *An Astronomer's Guide to On-line Bibliographic Databases and Information Services*, describes bibliographic and other resources of interest to astronomers.

6.1 Observations

SIMBAD[97] is a 'Set of Identifications, Measurements and Bibliography for Astronomical Data'. It's a large database (5,416,970 objects in November 1998) of astronomical objects, which you can search by identifier, coordinates, or other sampling criteria. As well as basic data for the objects, it includes some observational data from, and journal references to, the object. You have to register before you can use it; it's not free, but if you're in the US or in an ESO or ESA member state, the costs are covered for you. Also at CDS is the CDS catalogue service[98], which allows you to search a list of almost 3000 catalogues, plus the contents of tables published in A&A.

If you're using such catalogues, you might want to take a look at CAT, the Starlink *Catalogue and table manipulation library*, documented in SUN/181. This is a subroutine library for manipulating astronomical catalogues and similar tabular datasets. See also SUN/127, *The EXOSAT Database System*, and more generally SUN/162, *A Guide to Astronomical Catalogues, Databases and Archives available through Starlink*.

---

[96]`http://cdsads.u-strasbg.fr/`
[97]`http://simbad.u-strasbg.fr/Simbad`
[98]`http://cdsweb.u-strasbg.fr/Cats.html`

7 Acknowledgements

A number of people have contributed to this document, either with helpful general remarks or corrections or amplifications on points of detail. Where their contribution was precisely localisable -- in particular where I have merely sub-edited their comments into place -- they are also acknowledged at the appropriate specific points in the text.

- SG: Sergio Gelato (`Sergio.Gelato@Durham.ac.uk`) added remarks about optimizing C, and compiler switches.

- SJ: Simon Jeffery (`csj@star.arm.ac.uk`) added pointers to model atmosphere codes.

- JL: Jon Lockley (`jjl@astro.soton.ac.uk`)

- AS: Anne Sansom (`aesansom@uclan.ac.uk`).

- BS: Barry Smalley (`bs@astro.keele.ac.uk`). Pointers to model atmospheres.

- MBT: Mark Taylor (`mbt@ast.cam.ac.uk`) made a number of points about my coverage of Fortran, and several insightful remarks about numerical programming.

- RFWS: Rodney Warren-Smith made many comments on an early draft of the text, which made it much better than it otherwise might have been.

- RW: Robin Williams (`Robin.Williams@astro.cf.ac.uk`) general comments, and information about Alpha switches.

A Example programs

This section contains several example programs. These are also distributed with the source of this document. Look in `</star/examples/sc13*>`. The examples are available for download from `<http://www.astro.gla.ac.uk/users/norman/star/sc13/sc13-examples.tar.gz>`.

A.1 islittlendian.c

If you want to find out, possibly as part of a script, whether a particular platform is big- or little-endian, the information will probably be squirrelled away somewhere in the headers for the compiler you're using, unfortunately not in any standard way. However, a portable way of doing this (which incidentally illustrates the contrast between the two systems) is to use this little program. It isn't bullet-proof, but if it fails, this is probably the least of your cross-platform problems. You can use the program as follows:

```
% echo '#define BIGENDIAN' \
    'if ./islittleendian; then echo 0; else echo 1; fi' >bytesex.h
```

to create a file `bytesex.h` with either `#define BIGENDIAN 1` or `#define BIGENDIAN 0` in it.

```
main () {
    /* Are we little or big endian?  Originally from Harbison and Steele. */
    union
    {
long l;
char c[sizeof(long)];
    } u;
    u.l = 1;
    exit (u.c[sizeof(long)-1] == 1);
    /* Return 0 (success) if we're little-endian, or 1 (fail) if big-endian */
}
```

A.2 fpp.c

The first `fpp.c`, allows you to explore the representation of floating-point numbers on your machine. Note that machines may have either 'big-endian' or 'little-endian' addressing schemes (referring to the order in which integers and floating-point numbers are stored in memory). Alpha and Intel chips are little-endian, Sparcs, the Motorola 68k family used in Macs, and Motorola PowerPC chips, are big-endian. On the latter machines, you'll need to compile these programs with the `-DBIGENDIAN` option to the C compiler.

```
/*
 * Explore IEEE single- and double-precision floats.
 *
 * Invoked with an argument, it reprints that number as a double, hex
 * and binary.  Without argument, it reads numbers from stdin.  The
```

```
 * argument is either a double or an integer (hex, with leading '0x',
 * eg 0x3f800000) as argument.
 *
 * $Id: fpp.c 12350 2004-03-10 19:16:05Z norman $
 */

#include <config.h>

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#if HAVE_ERRNO_H
#include <errno.h>
#else
extern int errno;
#endif

/* bytesex.h defines BIGENDIAN=1 (true) or 0 */
#include "bytesex.h"

#ifndef DOUBLEPRECISION
#define DOUBLEPRECISION 1 /* double-precision by default */
#endif

#if DOUBLEPRECISION
#define EXPONENTWIDTH 11
#define MANTISSAWIDTH 20 /* actually just the part of the
   mantissa in the MSB */
#define ZEROMANTISSA(F) ((F).ieee.m == 0 && (F).ieee.m2 == 0)
#define NINTS 2 /* number of ints in a double */
typedef double Float;
#else
#define EXPONENTWIDTH 8
#define MANTISSAWIDTH 23
#define ZEROMANTISSA(F) ((F).ieee.m == 0)
#define NINTS 1 /* number of ints in a float */
typedef float Float;
#endif
/* Create an integer with EXPONENTWIDTH set bits.  Relies on
   zero-filling when left-shifting. */
#define FULLEXPONENT ~(~0 << EXPONENTWIDTH)

/* Define the structure we'll use to do the manipulations */
typedef union {
    Float f;
    unsigned int i[NINTS];
    struct {
#if DOUBLEPRECISION
#if BIGENDIAN
unsigned int s  : 1;
unsigned int e  : 11;
unsigned int m  : 20;
unsigned int m2 : 32;
```

```
#else
unsigned int m2 : 32;
unsigned int m  : 20;
unsigned int e  : 11;
unsigned int s  : 1;
#endif
#else  /* DOUBLEPRECISION */
#if BIGENDIAN
unsigned int s : 1;
unsigned int e : 8;
unsigned int m : 23;
#else
unsigned int m : 23;
unsigned int e : 8;
unsigned int s : 1;
#endif
#endif
    } ieee;
} ieeefloat;

/* Function prototypes */
/* Utility functions */
int pisnan (const Float darg); /* true if argument is a NaN */
int pisinf (const Float darg); /* true if argument is infinite */
int pisfinite (const Float darg); /* true if argument is finite */
int pisnormal (const Float darg); /* true if argument is normal */
Float pnan (const char *tagp); /* return a NaN */
Float pinfinity (Float darg); /* return +ve or -ve infinity */

/* Functions which do the work of this demo program */
char *pbits (unsigned int i, unsigned int n);
int parse_int (const char *s, unsigned int i[NINTS]);
int reprint_number (char *s, Float farg);



/* Following are utility functions, which are portable */

/* Returns non-zero if argument is a NaN -- all bits in exponent set,
   and mantissa non-zero. */
int pisnan (const Float darg)
{
    ieeefloat d;
    d.f = darg;
#if DOUBLEPRECISION
    return (d.ieee.e == FULLEXPONENT && !(d.ieee.m == 0 && d.ieee.m2 == 0));
#else
    return (d.ieee.e == FULLEXPONENT && d.ieee.m != 0);
#endif
}

/* returns non-zero if argument is infinity -- all bits in exponent
   set, and mantissa zero. */
int pisinf (const Float darg)
```

```
    {
        ieeefloat d;
        d.f = darg;
#if DOUBLEPRECISION
        return (d.ieee.e == FULLEXPONENT && d.ieee.m == 0 && d.ieee.m2 == 0);
#else
        return (d.ieee.e == FULLEXPONENT && d.ieee.m == 0);
#endif
    }

    /* returns non-zero if argument is finite (normal, subnormal or zero)
       -- not all bits in exponent set. */
    int pisfinite (const Float darg)
    {
        ieeefloat d;
        d.f = darg;
        return (d.ieee.e != FULLEXPONENT);
    }

    /* Returns non-zero if argument is normal -- exponent is not all-ones
     * nor all-zeros.
     *
     * Note that this refers to the argument as a _double_, and might not
     * give the expected results if it's applied to a single which is
     * promoted to a double when given as a parameter.
     */
    int pisnormal (const Float darg)
    {
        ieeefloat d;
        d.f = darg;
        return (d.ieee.e != FULLEXPONENT && d.ieee.e != 0);
    }

    /* returns a NaN.  Argument is ignored */
    Float pnan (const char *tagp)
    {
        ieeefloat d;
        d.ieee.s  = 0;
        d.ieee.e  = FULLEXPONENT;
        /* Set leading bit in mantissa -- quiet NaN. */
        d.ieee.m = (1<<(MANTISSAWIDTH-1));
#if DOUBLEPRECISION
        d.ieee.m2 = 0;
#endif
        return d.f;
    }

    /* returns positive or negative infinity, depending on the sign of darg */
    Float pinfinity (Float darg)
    {
        ieeefloat d;
        d.f = darg; /* copies sign */
        d.ieee.e = FULLEXPONENT;
        d.ieee.m = 0;
```

```
#if DOUBLEPRECISION
    d.ieee.m2 = 0;
#endif
    return d.f;
}


/*
 * pbits: given an integer i (assumed 32 bits),
 * return the rightmost n (<=32) bits of the integer expressed in binary.
 */
char *pbits (unsigned int i, unsigned int n)
{
    static char s[33];
    char *p;
    /* printf ("pbits(%x/%d)", i, n); */
    s[n] = '\0';                           /* terminate the string */
    for (p=s+n-1; p>=s; p--, i>>=1)
        *p = ((i&1) ? '1' : '0');
    return s;
}


#if DOUBLEPRECISION
/* Take a string of (up to) 16 hex digits, remove leading '0x', and
 * any spaces, and convert the first 8 into i[0], and the second 8 into
 * i[1].  If the string is shorter than 16 digits, pad the remainder
 * with '0'.
 *
 * Only need this for converting 16-digit strings to d-p floats.
 *
 * Return non-zero on error.
 */
int parse_int (const char *s, unsigned int i[2])
{
    char digits0[9], digits1[9], *p;
    int ndigits = 0;

    if (!(s[0] == '0' && s[1] == 'x'))
return 1;
    p = digits0;
    s += 2;
    while (ndigits <= 16)
    {
while (isspace(*s)) /* skip blanks */
    s++;
if (*s == '\0') /* pad remainder */
    *p++ = '0';
else
    *p++ = *s++;
ndigits++;
if (ndigits == 8) /* done 8 digits - switch to second 8*/
    p = digits1;
    }
    digits0[8] = digits1[8] = '\0'; /* terminate the strings */
```

```
        i[0] = strtoul (digits0, (char**)NULL, 16); /* ...and convert them */
        i[1] = strtoul (digits1, (char**)NULL, 16);

        /* printf ("parse_int: digits=%s %s --> %08x %08x\n",
            digits0, digits1, i[0], i[1]); */
        return 0;
}
#else
/* same, but simpler for single precision */
int parse_int (const char *s, unsigned int i[1])
{
        char digits0[9], *p;
        int ndigits;

        if (!(s[0] == '0' && s[1] == 'x'))
return 1;
        p = digits0;
        s += 2;
        for (ndigits=0; ndigits<=8; ndigits++)
        {
while (isspace(*s)) /* skip blanks */
        s++;
if (*s == '\0') /* pad remainder */
        *p++ = '0';
else
        *p++ = *s++;
        }
        digits0[8] = '\0'; /* terminate the strings */

        *i = strtoul (digits0, (char**)NULL, 16); /* ...and convert them */

        /* printf ("parse_int: digits=%s --> %08x\n", digits0, *i); */
        return 0;
}
#endif


/*
 * Convert string to number and display in various formats.
 * If s is NULL, then use the number farg instead.  If the input
 * string cannot be interpreted, return non-zero, or zero on success.
 */
int reprint_number (char *s, Float farg)
{
        ieeefloat F;
        if (s == NULL)
F.f = farg;
        else
        {
if (s[0]=='0' && s[1]=='x') { /* it's an integer */
        if (parse_int (s, F.i)) {
printf ("can't parse integer <%s>\n", s);
return 1;
        }
```

```
} else {
            char *endptr;
            errno = 0;
    F.f = strtod (s, &endptr);
            if (errno != 0)
                return 1;
            for (; isspace(*endptr); endptr++)
                ;
            if (*endptr != '\0')
                return 1;
        }
    }

#if DOUBLEPRECISION
    /* 49 bits of precision is 49*log_10(2)=14.75 dec.digits of precision */
printf ("double  %24.16e\n   hex  %08x %08x\n",
    F.f, F.i[0], F.i[1]);
#else
printf (" float  %12.7e\n   hex  %08x\n", F.f, F.i[0]);
#endif
    printf ("binary  %s ", pbits (F.ieee.s, 1));
    printf ("%s ", pbits (F.ieee.e, EXPONENTWIDTH));
    printf ("%s ", pbits (F.ieee.m, MANTISSAWIDTH));
#if DOUBLEPRECISION
    printf ("%s", pbits (F.ieee.m2, 32));
#endif
    printf ("\n  type  ");

    /* now print out what type of number it is */
    if (F.ieee.e == 0)
        if (ZEROMANTISSA(F))
            printf ("%s zero",
                    (F.ieee.s ? "Negative" : "Positive"));
        else
            printf ("Subnormal");
    else if (F.ieee.e == FULLEXPONENT)
        if (ZEROMANTISSA(F))
            printf ("%s infinity",
                    (F.ieee.s ? "Negative" : "Positive"));
        else
            printf ("%s NaN",
                    (F.ieee.m & (1<<(MANTISSAWIDTH-1))
     ? "Quiet"
     : "Signalling"));
    else
        printf ("Normal");

    /* test the pis??? routines */
    printf (" (%c%c%c%c)\n",
    (pisnan    (F.f) ? 'N' : 'n'),
    (pisinf    (F.f) ? 'I' : 'i'),
    (pisfinite (F.f) ? 'F' : 'f'),
    (pisnormal (F.f) ? 'L' : 'l'));
```

```
        return 0;
    }

    int main (int argc, char **argv)
    {
        Float f = 0;

        printf ("%s-precision %s endian...\n",
        (DOUBLEPRECISION ? "Double" : "Single"),
        (BIGENDIAN ? "big" : "little"));

        if (argc > 1)
            reprint_number (argv[1], 0);
        else
        {
            char line[80];
            while (fgets (line, sizeof(line), stdin) != NULL)
        if (line[0] == '=') {
    switch (line[1]) {
      case 'n':
        reprint_number (NULL, pnan(""));
        break;
      case 'i':
        reprint_number (NULL, pinfinity(+1.0));
        break;
      case 'z':
        reprint_number (NULL, 0);
        break;
      default:
        printf ("eh?\n");
        break;
    }
                } else {
    if (reprint_number (line, 0))
                        printf("Enter float, 0x... integer, =n, =i, =z, "
                               "or ^D to exit\n");
                }
        }
        exit (0);
    }
```

A.3 fpdemo.c

The program `fpdemo.c` is discussed in section *Section 2.3.2.2*, and shows the effects of catastrophic cancellation.

```
    #include <stdio.h>

    /* bytesex.h defines BIGENDIAN=1 (true) or 0 */
    #include "bytesex.h"

    /*
```

```
 * pbits: given an integer i, put the rightmost n bits of the integer
 * expressed in binary, into the string s.
 */
void pbits (unsigned int i, unsigned int n, char *s)
{
    char *p;
    for (p=s+n-1; p>=s; p--, i>>=1)
        *p = ((i&1) ? '1' : '0');
    return;
}


/* Given a float, return a (static) string holding its bit pattern */
char *print_float (float f)
{
    static char b[35];
    union {
        float f;
        struct {
#if BIGENDIAN
            unsigned int s : 1;
            unsigned int e : 8;
            unsigned int m : 23;
#else
            unsigned int m : 23;
            unsigned int e : 8;
            unsigned int s : 1;
#endif
        } ieee;
    } ieeefloat;

    ieeefloat.f = f;

    pbits (ieeefloat.ieee.s, 1,  b);
    b[1] = ' ';
    pbits (ieeefloat.ieee.e, 8,  &b[2]);
    b[10] = ' ';
    pbits (ieeefloat.ieee.m, 23, &b[11]);
    b[34] = '\0';

    return b;
}



main ()
{
    float f, a, b, c;
    double fd, ad, bd, cd;

    printf ("\n  You lose accuracy adding small numbers to large ones...\n");
    /* 2^23+1 works OK... */
    f = 8388608.0;
    printf ("2^23 = %e = %s\n", f, print_float (f));
    f = f + 1.0;
    printf ("  +1 = %e = %s\n", f, print_float (f));
```

```
      /* ...but 2^24+1=2^24 - catastrophic loss of precision */
      f = 16777216.0;
      printf ("2^24 = %e = %s\n", f, print_float (f));
      f = f + 1.0;
      printf ("  +1 = %e = %s\n", f, print_float (f));

   printf ("\n\n  Display the dread effects of catastrophic cancellation\n\
by calculating a*b-a*c, in several different ways.  These are equivalent\n\
arithmetically, but not computationally.\n");
    /* Now show the effect of catastrophic cancellation by calculating
     * ab-ac, when a, b and c are such that the terms nearly cancel.
     * To generate numbers which best demonstrate the effect,
     * set a = (1+da/2^12), b=(1+db/2^12), etc.  Thus
     * a * b = (1 + db/2^12 + da/2^12 + da.db/2^24).
     * Put da=1.0.  Now pick db=1-, so that final term is just below 1/2^24,
     * then dc=1+, so that final term is just above 1/2^24.
     * Thus a*b rounds down, and a*c rounds up.
     * Both are quite accurate, but large errors are revealed when
     * they're subtracted.  We can ameliorate this by rewriting the expression.
     */

    /* First, do the calculation in double, to get relative errors. */
    /* The cancellation is ignorable in double */
    ad = 1.0 + ((double)1/4096);
    bd = 1.0 + ((double)0.9/4096);
    cd = 1.0 + ((double)1.1/4096);
    fd = ad*bd-ad*cd;

    a = 1.0 + ((float)1/4096);   /* a=1.000244; */
    b = 1.0 + ((float)0.9/4096); /* b=1.000220; */
    c = 1.0 + ((float)1.1/4096); /* c=1.000269; */

    /* first method - naive */
    f = a*b-a*c;
    printf ("a=%e  b=%e  c=%e\n", a, b, c);
    printf ("a*b-a*c      = %e  (error=%e)\n", f, ((double)f/fd-1.0));

    /* pre-subtract the nearly-equal b and c */
    f = a*(b-c);
    printf ("a*(b-c)      = %e  (%e)\n", f, ((double)f/fd-1.0));

    /* rewrite the expression, to calculate a * ((b-1) - (c-1)).
       Thus b-1 and c-1 have full accuracy */
    b = ((float)0.9/4096);       /* = (above b) -1 */
    c = ((float)1.1/4096);
    f = a*(b-c);

    printf ("a((b-1)-(c-1))= %e  (%e)\n", f, ((double)f/fd-1.0));
    /* Can't do the same trick with a.  If we calculate (a-1)*() + 1*(),
       we don't get any improvement.  We're not carelessly discarding
       accuracy, now - we can't keep any more than this. */

    printf ("\n  ...and further illustrate what's happening by showing\n\
b and b-1.  Note the extra accuracy in the latter.\n");
```

```
/* Display b and (b-1).  Note extra accuracy in latter. */
b = 1.0 + ((float)0.9/4096);/* = (above b) */
printf ("1+1/10000 = %14.7e = %s\n", b, print_float (b));
b = ((float)0.9/4096);        /* = (above b) -1 */
printf ("  1/10000 = %14.7e = %s\n", b, print_float (b));;

printf ("\n\n  Display Infinity and NaN\n");
/* Display NaNs and Infinity */
/* Don't just write a=1.0/0.0, since compiler can warn about this */
a = 0.0;                      /* and log(0) */
a = 1/a;
b = 1/a;
c = a*b;
printf ("a = 1/0.0 = %14e = %s\n", a, print_float (a));
printf ("b = 1/a   = %14e = %s\n", b, print_float (b));
printf ("c = a*b   = %14e = %s\n", c, print_float (c));
}
```

A.4 Profiling

The fortran files p?.f are example files for *Section 2.5.1*, on profiling. These are drawn from
[sunf77].

A.4.1 p1.f

```
program silly
     implicit none
     integer n, i
     real determinant
     parameter (n=2)
     real twobytwo(2,2) /4*-1/
     do i = 1, 100000
        call mkidentity (twobytwo, n)
     enddo
     print *, determinant (twobytwo)
     end
```

A.4.2 p2.f

```
subroutine mkidentity (matrix, dim)
     implicit none
     integer dim
     real matrix (dim,dim)
     integer m,n
     do m = 1, dim
        do n = 1, dim
           if (m.eq.n) then
              matrix(m,n) = 1.
           else
```

```
            matrix(m,n) = 0.
        endif
      enddo
    enddo
    return
    end
```

A.4.3 p3.f

```
real function determinant(m)
    implicit none
    real m(2,2)
    determinant = m(1,1) * m(2,2) - m(1,2) * m(2,1)
    return
    end
```

A.5 crash.c

The program crash.c crashes and dumps core. See *Section 2.5.3* for discussion.

```
#include <stdio.h>

int bananas (int i)
{
    char buf[80];
    sprintf (buf, "banana number %d\n", i);
    if (i != 0)
        fputs (buf, stdout);
    else
    {
        int *zp = 0;
        printf ("banana split: %d\n", *zp);
    }
    return 0;
}

int main (void)
{
    printf ("Entering the bananas function: 1\n");
    bananas (1);
    printf ("No bananas left!\n");
    bananas (0);
    printf ("We made it out!\n");
}
```

A.6 Mixed language programming

The files mixed-c.c and mixed-f.f illustrate some of the techniques involved in calling C from Fortran and *vice versa*, as described in *Section 2.5.4*.

On a Sun, compile and link them as follows:

```
f77 -c mixed-f.f -ext_names=plain
cc -c mixed-c.c
f77 -o mixed mixed-c.o mixed-f.o
```

The extra option to the first `f77` command tells the compiler not to add an underscore to function names; different compilers will have different switches for accomplishing the same thing. See *Section 2.5.4.3* for discussion.

A.6.1 mixed-c.c

```c
#include <stdio.h>

int main (void)
{
    float m[3][3];
    int i, j, n;

    /*
     * Note the C declaration of this Fortran function.  Although
     * we're passing a matrix, we declare the first argument just
     * float* .  This is because a dynamically-dimensioned array can't
     * be expressed in C.
     */
    extern void addarr (float *m, int *n);

    n = 3;                          /* dimension of matrix */
    for (i=0; i<3; i++)
        for (j=0; j<3; j++)
            m[i][j] = (float)(10*i+j);

    printf ("Before:\n");
    for (i=0; i<3; i++)
    {
        printf ("m[%d][j]: ", i);
        for (j=0; j<3; j++)
            printf ("%6.0f", m[i][j]);
        printf ("\n");
    }

    /* Pass the array, coerced to (float*). */
    /* &m[0][0] (address of the first element of the array) would also work. */
    addarr ((float*)m, &n);

    printf ("\nAfter:\n");
    for (i=0; i<3; i++)
    {
        printf ("m[%d][j]: ", i);
        for (j=0; j<3; j++)
            printf ("%6.0f", m[i][j]);
```

```
        printf ("\n");
    }
}

/* cadd2 adds twice its second argument to its first. */
void cadd2 (float *a1, float a2)
{
    a2 *= 2;                /* double a2 - this doesn't change a2 in caller */
    *a1 += a2;              /* add a2 to what a1 points to */
}
```

A.6.2 mixed-f.f

```
c   Given a square array m(n,n), call cadd2 on each element
    subroutine addarr (m, n)
    real m(n,n)
    integer n
    integer i,j
    real addval

    addval = m(n,n)
    do j=1,n
       do i=1,n
          call cadd2 (m(i,j), %val(addval))
       enddo
    enddo

    return
    end
```

A.7 maple.ms

This example Maple file gives a slightly fuller example of Maple use than given in *Section 3.1*.

```
# read into maple with   > read 'maple.ms':

# Define a 2-d gaussian
gi:=amp*exp(-(aparam^2/sa^2+bparam^2/sb^2)/2):
aparam:=cos(theta)*(xc-x0)+sin(theta)*(yc-y0):
bparam:=-sin(theta)*(xc-x0)+cos(theta)*(yc-y0):

# Declare arrays.  We expressed the gaussian in terms of intelligible
# identifiers, but we want these to be output as array references when
# we produce the Fortran at the end
xa := array(1..5):
ca := array(1..1):

x0 := xa[1]:
y0 := xa[2]:
sa := xa[3]:
```

```
sb := xa[4]:
theta := xa[5]:
amp := ca[1]:


# aadf are the variables we'll differentiate with respect to
aadf:=[x0,y0,sa,sb,theta]:
# ...and array of differentials.
dyda:=array(1..6):
for i to 5 do
        dyda[i]:=diff(gi,aadf[i])
od:
dyda[6] := gi:

# Output the result as Fortran
fortran (dyda, filename='gaussianab.f', optimized);
```

A.8 idl-functions.pro

Here is an example of IDL programming, dealing with the common situation of reading in columns of data. See also *Section 3.2.2*.

```
; Common IDL functions

; Read the contents of an array from a file.
;
; The file is in 'Fortran format': it holds the contents of the array
; as a list of numbers, with the left-most array index changing
; fastest.  For simplicity, the function does no error checking, and
; assumes that the array is the right length.
pro read_file, a, fn
        openr, iu, fn, /get_lun    ; open the file to read, allocate unit
        readf, iu, a
        free_lun, iu               ; close the file and release the unit.
end


; Read columns of numbers into a 2-d array.
;
; Given a file with  n  columns and  m  rows, this takes an  n x m
; array and fills it with the contents of the file.
;
; It ignores lines beginning #
pro read_columns, a, fn
        dims = size (a)            ; get array dimensions
        line = ''
        print, "cols=",dims[1]," rows=",dims[2]
        openr, iu, fn, /get_lun
        linevals = fltarr(dims[1])
        i = 0
        while not eof(iu) do begin
                readf, iu, line
```

```
                    if (strmid(line,0,1) ne '#') then begin
                            reads,line,linevals
                            if (n_elements(linevals) ne dims[1]) then begin
                                    print,'line ',i,": expected',dims[1], $
                                            ' got',n_elements(linevals)
                                    return
                            endif
                            if (i ge dims[2]) then message,'too many lines in file'
                            a(*,i) = linevals
                            i = i+1
                    endif
            end
            if (i ne dims[2]) then message,'too few lines in file'
            close, iu
    end
```

B References: Starlink documents

This section contains a (non-exhaustive!) selection of Starlink documents which should intersect with the interests of this cookbook's audience. Where possible, I have included a reference to a relevant section of this document.

An overall reference for all Starlink users is the sug/.

	Starlink User Notes	
Code	Title	Sect.
SUN/1	Starlink software collection	
SUN/9	LaTeX -- Document preparation system v2e User's Guide	*Section 5.1*
SUN/11	ARY -- Subroutines for accessing ARRAY data structures	
SUN/12	LaTeX -- Cook-book	*Section 5.1*
SUN/15	PGPLOT -- Graphics subroutine library	*Section 3.2.4*
SUN/28	NAG -- Numerical and Graphical Libraries Mk 16/4 UG	*Section 4.2*
SUN/33	NDF -- Routines for accessing extensible n-D data 1.3	*Section 4.2.1*
SUN/55	CONVERT -- A format-conversion package 1.1, UM	*Section 3.2.2*
SUN/73	FORCHECK -- Fortran verifier and programming aid	*Section 2.4.1*
SUN/92	HDS -- Hierarchical data system 4.2, PG	*Section 4.2.1*
SUN/93	TeX -- Superior document preparation system	*Section 5.1*
SUN/107	MAPLE -- Mathematical manipulation language	*Section 3.1*
SUN/127	EXOSAT database system	*Section 6.1*
SUN/136	FITSIO -- Disk FITS I/O routines 5.03	*Section 4.2.1*
SUN/145	UNIX -- An introduction	*Section 2.1*
SUN/160	IMG -- Simple image data access 1.2	*Section 4.2.1*
SUN/162	A guide to astronomical catalogues/databases/archives	*Section 6.1*
SUN/167	PERIOD -- A time-series analysis package	
SUN/170	Editors on Unix	*Section 2.1*
SUN/172	FTNCHEK -- A Fortran 77 source-code checker	*Section 2.4.1*
SUN/174	Guide to on-line bibliographies and information	
SUN/181	CAT -- Catalogue and table manipulation library 5.1	*Section 6.1*
SUN/193	PERL -- Practical Extraction and Report Language	*Section 2.4.6*
SUN/194	PDA -- Public domain algorithms library 0.4, PM	*Section 4.2*
SUN/203	SX and DX -- IBM data explorer for data visualisation	*Section 3.2.3*
SUN/209	CNF and F77 Mixed Language Programming v3.1: PM.	*Section 2.5.4*

	Starlink General Papers	
SGP/4	Starlink C programming standard	*Section 2.4.3*
SGP/16	Starlink application programming standard	*Section 2.4.1*
SGP/47	Computer Algebra Software	*Section 3.1*

	Starlink Cookbooks	
SC/2	The DX cookbook	*Section 3.2.3*
SC/9	LaTeX Cookbook	*Section 5.1*
SC/12	Writing your own data reduction software	

	Starlink Guides	
SG/6	ADAM -- Programmer's facilities and documentation guide	
SG/8	Introduction to Visualisation Software for Astronomy	*Section 3.2*

Miscellaneous User Documents are documents which are not originated by Starlink, which are held at RAL for the Starlink project. Copies may be available at other Starlink nodes. Ask your Site Manager if you want one.

Miscellaneous User Documents		
MUD/30	IDL -- User's guide	*Section 3.2.2*
MUD/48	LATEX -- User's guide and reference manual	*Section 5.1*
MUD/52	MAPLE -- Introduction	*Section 3.1*
MUD/55	NAG -- Fortran library manual: Mk 15 (10 vols)	*Section 4.2*
MUD/56	NAG -- A beginners guide (Book)	*Section 4.2*
MUD/57	NAG -- Newsletters, Error bulletins	*Section 4.2*
MUD/58	NAG -- Graphical library supplement: Mk 3 (2 vols	*Section 4.2*
MUD/72	TEX -- The TeXbook (Book)	*Section 5.1*
MUD/121	Unix for beginners	*Section 2.1*
MUD/122	An introduction to display editing with Vi	*Section 2.2*
MUD/123	Vi -- Quick reference card	*Section 2.2*
MUD/128	NAG -- Fortran Library, Introductory Guide (Mk 15)	*Section 4.2*
MUD/137	MAPLE V -- Language reference manual	*Section 3.1*
MUD/138	MAPLE V -- Library reference manual	*Section 3.1*
MUD/139	MAPLE V -- A tutorial introduction	*Section 3.1*
MUD/159	SM -- Interactive plotting program 2.3.1	*Section 3.2*
MUD/160	SM -- The SM tutorial	*Section 3.2*

C Release notes

C.1 Release 2.2

- Enhanced the coverage of Java (though it's still rather brief).

- Added pointers to stellar atmosphere codes (thanks to Barry Smalley, Ann Sansom, Simon Jeffery).

- A few additions to the TeX/LaTeX coverage.

C.2 Release 2.1

More detail about NaNs, compilers, largely incorporating the suggestions I received after the previous edition. Packaged set of examples.

C.3 Version 2

Assorted enhancements and additions. Expanded the section on model atmospheres. Links checked, and a couple of broken ones restored.

I haven't been able to incorporate all the suggestions I received after the previous edition.

C.4 Release 1.2

No significant changes as yet. Merely a conversion from the original LaTeX source.

Bibliography

References

[arnold98] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language.* The Java Series. Addison-Wesley, 3rd edition, 2000.

[bloch01] Joshua Bloch. *Effective Java.* The Java Series. Addison-Wesley, 2001.

[demmel] J~Demmel. Basic issues in floating point arithmetic and error analysis. Lecture notes, University of California, Berkeley, 1996. URL: <http://www.cs.berkeley.edu/%7Edemmel/cs267/lecture21/lecture21.html>. URL: <http://www.cs.berkeley.edu/%7Edemmel/cs267/lecture21/lecture21.html>.

[diller] Antoni Diller. *LaTeX Line by Line.* John Wiley, 1993.

[dowd] Kevin Dowd and Charles~R Severance. *High Performance Computing.* O'Reilly, 2nd edition, 1998.

[nutshell] Daniel Gilly. *Unix in a Nutshell: System V Edition.* O'Reilly, 1992. URL: <http://www.oreilly.com/catalog/unutv/>. URL: <http://www.oreilly.com/catalog/unutv/>.

[goldberg91] David Goldberg. What every computer scientist should know about floating point arithmetic. *Computing Surveys*, 23(1):5--48, March 1991. Reprinted in [sunncg].

[goossens] Michel Goossens, Frank Mittelback, and Alexander Samarin. *The LaTeX Companion*. Addison Wesley, 1993.

[hauser96] John~R Hauser. Handling floating-point exceptions in numeric programs. *ACM Transactions on Programming Languages and Systems*, 18(2):139--174, 1996. URL: <http: / / http.cs.berkeley.edu / %7Ejhauser / exceptions / HandlingFloatingPointExceptions.html>. URL: <http: / / http.cs.berkeley.edu / %7Ejhauser/exceptions/HandlingFloatingPointExceptions.html>.

[kahan96] William Kahan. IEEE standard 754 for binary floating-point arithmetic. Lecture notes, University of California, Berkeley, 1996. URL: <http://http.cs.berkeley.edu/ %7Ewkahan/ieee754status/ieee754.ps>. URL: <http://http.cs.berkeley.edu/%7Ewkahan/ ieee754status/ieee754.ps>.

[kr] Brian~W Kernighan and Dennis~M Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[knuth] Donald Knuth. *The TeXbook*, volume~A of *Computers and Typesetting*. Addison-Wesley, 1984.

[kopka] Helmut Kopka and Patrick~W. Daly. *A guide to LaTeX2e*. Addison-Wesley, 2nd edition, 1995.

[lamport] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison Wesley, 2nd edition, 1994.

[metcalf96] Metcalf and Reid. *Fortran 90/95 Explained*. Oxford University Press, 1996.

[metcalf85] Mike Metcalf. *Effective Fortran 77*. Oxford University Press, 1985.

[nr] William~H Press, Saul~A Teukolsky, William~T Vetterling, and Brian~P Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992. URL: <http://www.nr.com>. URL: <http://www.nr.com>.

[quigley] Ellie Quigley. *Unix Shells by Example*. Prentice Hall, 1997.

[schwartz97] Randal~L Schwartz and Tom Christiansen. *Learning Perl*. O'Reilly, 2nd edition, 1997.

[stroustrup] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1998.

[sundebug] Sun Microsystems. *Debugging a Program*. SunSoft WorkShop SPARCworks 3.1. SunSoft, 1995. Sun part number 802-3517-10.

[sunf77] Sun Microsystems. *Fortran 77 4.0 User's Guide*. SunSoft WorkShop SPARCompiler. SunSoft, 1995. Sun part number 802-2997-10.

[sunncg] Sun Microsystems. *Numerical Computation Guide*. SunSoft WorkShop SPARCompiler. SunSoft, 1995. Sun part number 802-3254-10, available for download from docs.sun.com.

[wall] Larry Wall, Tom Christiansen, and Randal~L. Schwartz. *Programming Perl*. O'Reilly, 2nd edition, 1996. URL: <http://perl.oreilly.com/>. URL: <http://perl.oreilly.com/>.

Change history

Distribution 2-5

Norman Gray, 10 March 2003

Minor edits, plus repackaging

5.1 TeX and LaTeX Added a pointer to Peter Flynn's Beginner's LaTeX

Distribution 2-4

Norman Gray, 22 July 2002

Minor edits for clarity, and to restore missing references.

2.3.2.1 Endianness of floating-point numbers Paragraph transformed into section

Distribution 2-3

Norman Gray, 16 July 2002

Minor edits -- typos and enhancements spotted now and again.

2.4.5 Java Hoisted the Java section up a level, and added some remarks about efficiency and profiling.

5.1 TeX and LaTeX Added a reference to the Adam Lewenberg book pointers

Distribution 2-2

Norman Gray, 1 December 2001

Slight reworking after comments

2.3.2.3 Other floating-point topics Mention that x.ne.x is true when x is a NaN

4.1 Astrophysical modelling codes Reworking of this section after comments from BS; added VALD and Dusty links.

Distribution 2-1

Norman Gray, 4 December 2001

Update to IEEE trap details

2.3.2.3 Other floating-point topics Added table of compilers and enabled IEEE traps

A Example programs Added download URL

Version 2

Norman Gray, 2 December 2001

Changes in version 2

Norman Gray, 2 December 2001

Updating, link-checking, general refreshing.

2.2 Editors Added more vi URLs, caught typos

2.3.2.2 Accuracy Added a brief description of the format of doubles

2.4 Programming languages Mention of SUN/209 in the context of dynamic memory allocation in Fortran.

2.4.5 Java Pointed out that JITs make Java a candidate for high-end codes, pointing to the Java Grande Forum for details.

2.5.2.3 I/O Checked fwrite/fread syntax, and discussed unformatted IO in Fortran.

2.5.5 Compilers, and other stray remarks on code Advice to use prototypes, to avoid another class of silly errors; added all-warnings switches for Sun and Compaq.

4.1 Astrophysical modelling codes Expanded this section considerably, with several more links.

4.2 General-purpose and numerical libraries Longer discussion of the GPL and other free licences.

4.2.1 Reading and writing data Added CONVERT example, and added pointers to FITS and cfitsio.

5.1 TeX and LaTeX Pointers to the LaTeX project, other tutorials, and the TeX FAQ. Some reorganisation.

A Example programs Added islittleendian example program

A.2 fpp.c Replaced with expanded/corrected program, which also deals with doubles.

Change 20 June 2000

Norman Gray, 20 June 2000

Incorporating comments from others.

2.2 Editors Expanded the description of vi

2.4.1 Fortran 77 Added pointers to Fortran standard document.

2.4.5 Java Added a description of Java's model of compiling to interpreted bytecodes, to the extent that this relates to efficiency and speed.

2.4.6.2 Perl Added pointer to the Schwartz reference, and a description of Perl's compile-then-interpret model.

2.5.2.1 Avoid using the `register` keyword in C Rewritten following comments

2.5.2.2 Walk through arrays in the correct order Updated/corrected comments on page faults by adding MBT's comments on cache faults.

2.5.2.3 I/O Added remarks and examples on raw IO

2.5.2.4 Use NaN and Infinity Mention speed tradeoffs, following comments

Change 14 June 2000

Norman Gray, 14 June 2000

Conversion to SGML

Version 1

Norman Gray, 22 December 1998

Changes in version 1

Norman Gray, 22 December 1998

Last LaTeX version