Scientific Computing
Chapter I
Computer Arithmetic

Jonathan Goodman
Courant Institute of Mathemaical Sciences

Last revised January 23, 2002

1 Introduction

One of the many sources of error in scientific computing is inexact computer arithmetic, which is called "roundoff error". Roundoff error is inevitable but its consequences vary. Some computations yield nearly exact results, while others give answers that are completely wrong. This is true even for problems that involve no other approximations. For example, the gaussian elimination algorithm for solving systems of linear equations would give the exact solution if all the computations were performed exactly. When these computations are done in finite precision floating point arithmetic, we might or might not get something close to the right answer.

Suppose your computed answer is completely wrong when a computation without roundoff would be exactly right. There are three possible situations. Distinguishing these is one of the most important lessions of the course. The first guess would be that your program has a bug. You can test for this by finding an instance of your problem that is small enough and easy enough for roundoff to be a less likely excuse. Assuming the coding is correct, the problem may be so "ill conditioned" that no algorithm could reasonably be expected to work. The last possibility is that there is an accurate method for getting the answer in floating point, just not yours. In this case, your method is called "unstable". You can analyse an unstable algorithm and try to find a stable variant. However, if your problem is ill conditioned, you took a wrong path long before the solution algorithm design stage. A large class of problems ranging from sophisticated "inverse problems" to simple polynomial interpolation are ill conditioned.

2 Roundoff and cancellation

If `x` is a floating point variable (type `float` or `double` in C) and we say, for example, `x=1/3`, then `x` will not get the value exactly 1/3. It is impossible to represent $1/3$ as a finite "decimal" either in base 10 or in base 2. We can imagine that the computer first computes the mathematical $1/3$ then rounds the result to a nearby number with a finite computer representation. Most arithmetic operations involving floating point numbers require roundoff. As a result, many properties of arithmetic, such as associativity of addition ($(x+y)+z = x+(y+z)$), and distributivity ($x \cdot (y+z) = x \cdot y + x \cdot z$), do not hold exactly when done in floating point.

We must be careful not to write codes that depend on exact arithmetic to work correctly. For example, suppose we want to divide the interval $[0,T]$ into n equal parts. The following code will probably be an infinite loop:

```
float deltaT, t;
deltaT = T/n;
for ( t = deltaT; t != T; t+= deltaT ) {...}
```

If we replace the continuation criterion `t != T` by `t <= T`, we will get either n or $n+1$ trips through the loop, depending on the sign of the roundoff error. To get the correct result, use exact integer arithmetic such as this:

```
float deltaT, t;
int   i;
deltaT = T/n;
t      = (float) 0;
for ( i = 0; i < n; i++) { t+=deltaT; ...}
```

In discussing the accumulation of roundoff errors, we suppose x is a real number that would result from a sequence of computations, and $\tilde{x}$ is what the computer produces in floating point arithmetic. To start, we consider the case where $\tilde{x}$ is simply the result of "rounding" x, that is, finding the computer floating point number closest to x. This rounding error, $x - \tilde{x}$, is generally proportional to x. The small constant of proportionality characterizes the accuracy of the arithmetic. For example, if we use decimal arithmetic and keep 4 significant decimals, then $\pi = 3.1415926535\cdots$ would round to $\tilde{\pi} = 3.142$. The rounding error is about $4 \cdot 10^{-4}$. The mass of a proton, in grams[1], is $m \approx 1.67252 \cdot 10^{-24}$. This number would round to $\tilde{m} = 1.673 \cdot 10^{-24}$, for a roundoff error of $\tilde{m} - m \approx -2 \cdot 10^{-28}$. It might seem that the rounding error for m is much less than for π. However, in relative terms they are about the same:

$$\left| \frac{\pi - \tilde{\pi}}{\pi} \right| \sim \left| \frac{m - \tilde{m}}{m} \right| \sim 10^{-4} \, ,$$

where we write $a \sim b$ to mean that a and b have the same order of magnitude.

It is convenient to represent the above rounding error by writing

$$\tilde{x} = (1+\epsilon)x \, . \tag{1}$$

[1] It might be more correct to write $m = 1.67252 \cdot 10^{-24}$ *grams*. In the computer we need to give a pure number. It is up to the programmer to remember that `m` is the number of grams.

We say that ϵ_{mach} is the "machine precision" if the rounding error in (1) satisfies

$$|\epsilon| \leq \epsilon_{\text{mach}} \ , \tag{2}$$

as long as x is in the range of "normalized numbers" where floating point arithmetic usually takes place. Actually, the "machine precision", ϵ_{mach}, is no longer a function of the machine since most machines use IEEE arithmetic. It does depend on whether you use single or double precision. Since the relative distance between computer floating point numbers is more or less uniform across the normal range, we may as well take $x = 1$. Thus, is possible to characterize ϵ_{mach} as the largest ϵ so that $1 + \epsilon$ rounds to 1. This is often taken as the definition of ϵ_{mach}.

Our model of computer arithmetic is that all error comes from rounding error. The result of any arithmetic operation should be "the exact result, correctly rounded". Suppose, for example, some place in a computation we have $z = x + y$. At that point, we have $\tilde{x}$ and $\tilde{y}$. First we perform the mathematical addition to get the exact $z_1 = \tilde{x} + \tilde{y}$. Then we round, finding the closest floating point number to z_1: $\tilde{z} = (1 + \epsilon_r)z_1$, where ϵ_r satisfies (2) because it corresponds to a single rounding operation.

Rounding error can accumulate in a computation. If we have $\tilde{x} = (1 + \epsilon_x)x$ and $\tilde{y} = (1 + \epsilon_y)y$, then $\tilde{z} = (1 + \epsilon_z)z$, where

$$\epsilon_z \approx \epsilon_r + \frac{x\epsilon_x + y\epsilon_y}{z} \ . \tag{3}$$

In this way, roundoff error can accumulate during a long computation and lead to errors much larger than roundoff. If there are many steps in the computation, then we may have $|\epsilon_z| >> \epsilon_{\text{mach}}$, and it is all but impossible that $|\epsilon_z| << \epsilon_{\text{mach}}$.

Even the accumulation of roundoff error would be usually harmless[2] but for "cancellation", the increase in error that comes from subtracting two positive (or negative) numbers. A world with no negative numbers or subtraction would have more accurate computer arithmetic than ours. Subtraction magnifies error because we measure error in relative terms rather than absolute terms. If x and y are positive, then (3) shows that $|\epsilon_z| \leq \epsilon_{\text{mach}} + |\epsilon_x| + |\epsilon_y|$. Error accumulates but it not amplified. On the other hand, if x and y have opposite signs but nearly equal magnitudes, then we may have $|z| >> |x| + |y|$ and we can have

$$|\epsilon_z| >> \epsilon_{\text{mach}} + |\epsilon_x| + |\epsilon_y| \ ,$$

because the denominator in (3) is much smaller than x or y. The error has not grown much in absolute terms:

$$|z - \tilde{z}| \leq \epsilon_{\text{mach}} |z| + |x - \tilde{x}| + |x - \tilde{x}| \ .$$

The error has grown in relative terms because the answer has shrunk.

[2] This is because of the accuracy of computer arithmetic. It would take a very long computation build up enough error to overwhelm the answer.

Error amplification through cancellation does not have to happen all at once. It can build up through a long computation with many subtractions. For example, start with the geometric series

$$3 = 1 + \frac{2}{3} + \left(\frac{2}{3}\right)^2 + \cdots \ .$$

We denote the tails of this series

$$d_k = 3 \cdot \left(\frac{2}{3}\right)^k = \left(\frac{2}{3}\right)^k + \left(\frac{2}{3}\right)^{k+1} + \cdots \ .$$

Clearly $d_{k+1} = d_k - \left(\frac{2}{3}\right)^k$. Suppose we start with $d_0 = 3$ and subtract the right hand terms $\left(\frac{2}{3}\right)^k$ one by one, giving

$$2 = 3 - 1 \ , \quad \frac{4}{3} = 2 - \frac{2}{3} \ , \quad \frac{8}{9} = \frac{4}{3} - \frac{4}{9} \ , \ \text{etc.}$$

If we do this on the computer, we quickly come to a number that is 100% wrong even though the two terms on the right side of the subtractions differ by a factor of 3 always. The C program below illustrates these points.

```
#include <stdio.h>

#define MAXN 100
#define PRINTFREQUENCY 10

int main() {

   int n;
   double twoby3, x, dapprox, dbetter, diff, rdiff;

   twoby3  = ( (double) 2)/( (double) 3);

   dapprox = (double) 3;
   dbetter = (double) 3;
   x       = (double) 1;

   for( n=0; n < MAXN; n++) {
      dapprox -= x;
      dbetter *= twoby3;
      x       *= twoby3;
      diff     = dbetter - dapprox;
      rdiff    = diff/dbetter;
      if( n % PRINTFREQUENCY == 0 ) {
        printf("n = %3d,  d = %10.3e,", n, dbetter);
        printf("  abs diff = %21.14e,", diff);
```

```
        printf("  rel diff = %10.3e\n", rdiff);  }
      }
    return 0;
 }
```

The output is

```
n =   0,  d =  2.000e+00,  abs diff =  0.00000000000000e+00,  rel diff =  0.000e+00
n =  10,  d =  3.468e-02,  abs diff = -3.26128013483640e-16,  rel diff = -9.403e-15
n =  20,  d =  6.015e-04,  abs diff = -3.31006923259825e-16,  rel diff = -5.503e-13
n =  30,  d =  1.043e-05,  abs diff = -3.31113649411179e-16,  rel diff = -3.175e-11
n =  40,  d =  1.809e-07,  abs diff = -3.31115317007293e-16,  rel diff = -1.831e-09
n =  50,  d =  3.137e-09,  abs diff = -3.31115347612976e-16,  rel diff = -1.056e-07
n =  60,  d =  5.439e-11,  abs diff = -3.31115347968405e-16,  rel diff = -6.087e-06
n =  70,  d =  9.433e-13,  abs diff = -3.31115347969819e-16,  rel diff = -3.510e-04
n =  80,  d =  1.636e-14,  abs diff = -3.31115347969917e-16,  rel diff = -2.024e-02
n =  90,  d =  2.837e-16,  abs diff = -3.31115347969920e-16,  rel diff = -1.167e+00
```

The output illustrates this point. The absolute error is nearly constant after the first ten steps but the relative error increases exponentially as the answer shrinks.

3 Stability and conditioning

The conditioning of a problem refers to how sensitive the answer is to small changes in the data. For this general discussion, we write $A(x)$ for some computed quantity, the "answer", that depends on data, x. The sensitivity of A with respect to x is $\frac{\partial A}{\partial x}$. This means that if Δx is a small change in x, and ΔA is the resulting change in A, then

$$\Delta A \approx \frac{\partial A}{\partial x} \Delta x \ .$$

As explained in the previous section, it is often more useful to calculate relative changes. The ratio between the relative change in A and the relative change in x is κ, the condition number:

$$\frac{\Delta A}{A} \approx \kappa \frac{\Delta x}{x} \ . \tag{4}$$

From this, a formula for the condition number is

$$\kappa = \frac{\partial A}{\partial x} \frac{x}{A} \ . \tag{5}$$

The condition number is a dimensionless measure of sensitivity. It is the same, for example, when A is the number of seconds or the number of hours.

The condition number limits the accuracy whith shich A can be calculated in floating point arithmetic. The first thing we do in any calculation is to round the data, x, to the nearest floating point number, causing an error on the order of the machine precision

$$\left| \frac{\Delta x}{x} \right| \lesssim \epsilon_{\text{mach}} \ .$$

Therefore, the error in A will be at least

$$\left| \frac{\Delta A}{A} \right| \gtrsim \kappa \epsilon_{\text{mach}} \ .$$

We will encounter problems in later chapters whose condition numbers are so large that A is essentially uncomputable even in double precision arithmetic.

Here is a simple example of an ill conditioned problem. Suppose $x = (x_1, x_2)^t$ is a two dimensional vector (the t superscript means transpose so that x is a column vector). Let

$$P = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{3}{4} & \frac{1}{4} \end{pmatrix} \ .$$

Consider the iteration process $x^{k+1} = Px^k$, starting with some x^0. The problem is to use the two numbers x^k to reconstruct the two numbers x^0. We can understand what happens here using the eigenvalues and eigenvectors of P. It is easy to check that

$$P \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \text{and} \quad P \begin{pmatrix} 1 \\ \frac{-3}{2} \end{pmatrix} = -\frac{1}{4} \begin{pmatrix} 1 \\ \frac{-3}{2} \end{pmatrix} \ .$$

This gives the eigenvalues ($\lambda_1 = 1$, $\lambda_2 = -\frac{1}{4}$ and the eigenvectors. If we write

$$x = y_1 \begin{pmatrix} 1 \\ 1 \end{pmatrix} + y_2 \begin{pmatrix} 1 \\ \frac{-3}{2} \end{pmatrix} \ ,$$

then, (letting y^k correspond to the iterate x^k)

$$y_1^{k+1} = y_1^k \quad \text{and} \quad y_2^{k+1} = -\frac{1}{4} y_2^k \ .$$

so

$$y_1^k = y_1^0 \quad \text{and} \quad y_2^k = -\left(\frac{1}{4}\right)^k y_2^0 \ .$$

This means that the iteration process rapidly removes the information about y_2 while preserving y_1. If we perturb x^k by roundoff error in each component, that will perturb two components of y^k by about the same amount. However, the y_2 perturbation will be amplified by a factor of $(-4)^k$. For $k = 25$ in single precision or $k = 50$ in double precision, this will lead to complete loss of accuracy. The underlying reason it is impossible to reconstruct y_2^0 from x^{50} is that the information is not there. The iteration process has essentially removed all trace of y_2.

Even if A is "well condtioned", i.e. κ is not too large, we may still get the wrong answer in a computation, if we use an "unstable" algorithm. The stability of a computational algorithm is a measure of the relative error amplification beyond the minimum (4) implied by the condition number. For example, in the program above, the number d is computed in two ways, by repeated subtraction (the variable `dapprox`) and through repeated multiplication by 2/3 (the variable `dbetter`). It is clear that `dbetter` is correct to within a factor of 100 of ϵ_{mach}. On the other hand, the relative error in `dapprox` grows exponentially because d decreases exponentially. Note that the absolute difference between d and `dapprox` essentially stops changing after the first few steps. In this case, repeated subtraction is an unstable way to compute d.

4 The IEEE floating point standard

The IEEE floating point standard is a set of rules issued by the IEEE (Institute of Electrical and Electronics Engineers) on computer representation and processing of floating point numbers. Today, most computers claim to be IEEE compliant but many cut corners in what (they consider to be) minor details. The standard is currently being enlarged to specify some details left open in the original standard, mostly on how programmers interact with flags and traps. The standard has four main goals:

(1) To make floating point arithmetic as accurate as possible.

(2) To produce sensible outcomes in exceptional situations.

(3) To standardize floating point operations across computers.

(4) To give the programmer control over exception handling.

The IEEE standard specifies exactly what floating point numbers are and how they are to be represented in hardware. The most basic unit of information that a computer stores is a *bit*, a variable whose value may be either 0 or 1. Bits are organized into groups of 8 called bytes. The most common unit of computer number information is the 4 byte (32 bit) word. For higher accuracy this is doubled to 8 bytes or 64 bits. There are 2 possible values for a bit, there are $2^8 = 256$ possible values for a byte, and there are 2^{32} different 4 byte words, about 4.3 billion. A typical computer should take less than an hour to list all 4.3 billion 32 bit words.

There are two computer data types that represent numbers. A fixed point number has type `int` in C and type `integer` in FORTRAN. A floating point number has type `float` in C and `real` in FORTRAN. In most C compilers, a `float` by by default has 8 bytes instead of 4.

Integer arithmetic is very simple. There are $2^{23} \approx 4 \times 10^9$ 32 bit integers filling the range from about $-2 \cdot 10^9$ to $2 \cdot 10^9$. Addition , subtraction, and multiplication are done exactly whenever the answer is within this range. Most computers will do something unpredictable when the answer is out of range (overflow). For scientific computing, fixed point (integer) arithmetic has two drawbacks. One is that there is no representation for numbers that are not integers. Equally important is the small range of values. The number of dollars in the US national debt, several trillion (10^{12}), cannot be represented as a 32 bit integer but is easy to approximate in 32 bit floating point.

A floating point (or "real") number is a computer version of the exponential (or "scientific") notation used on calculator displays. Consider the example expression:

$$-.2491\mathrm{E}-5$$

which is one way a calculator could display the number -2.491×10^{-6}. This expression consists of a sign bit, $s = -$, a mantissa, $m = 2491$ and an exponent, $e = -5$. The expression $s.m\mathrm{E}e$ corresponds to the number $s \cdot .m \cdot 10^e$.

The IEEE format replaces the base 10 with base 2, and makes a few other changes. When a 32 bit word is interpreted as a floating point number, the first bit is the sign bit, $s = \pm$. The next 8 bits form the "exponent", e, and the remaining 23 bits determine the "fraction", f. There are two possible signs, 256 possible exponents (ranging from 0 to 255), and $2^{23} \approx 8.4$ million possible fractions. Normally a floating point number has the value

$$x = \pm 2^{e-127} \cdot (1.f)_2 \quad ,$$

where f is base 2 and the notation $(1.f)_2$ means that the expression $1.f$ is interpreted in base 2. Note that the mantissa is $1.f$ rather than just the fractional part, f. In base 2 any number (except 0) can be normalized so that the mantissa has the form $1.f$. There is no need to store the "1." explicitly. For example, the number $2.752 \cdot 10^3 = 2572$ can be written

$$\begin{aligned} 2752 &= 2^{11} + 2^9 + 2^7 + 2^6 \\ &= 2^{11} \cdot \left(1 + 2^{-2} + 2^{-4} + 2^{-5}\right) \\ &= 2^{11} \cdot \left(1 + (.01)_2 + (.0001)_2 + (.00001)_2\right) \\ &= 2^{11} \cdot (1.01011)_2 \quad . \end{aligned}$$

Thus, the representation of this number would have sign $s = +$, exponent $e - 127 = 11$ so that $e = 138 = (10001010)_2$, and fraction $f = (01011000000000000000000)_2$. The entire 32 bit string corresponding to $2.752 \cdot 10^3$ then is:

$$\underbrace{1}_{s}\underbrace{10001010}_{e}\underbrace{01011000000000000000000}_{f} \quad .$$

The exceptional cases $e = 0$ (which would correspond to 2^{-127}) and $e = 255$ (which would correspond to 2^{128}) have complex and carefully engineered interpretations that make the IEEE standard distinctive. If $e = 0$, the value is

$$x = \pm 2^{-126} \cdot 0.f \quad .$$

This feature is called "gradual underflow". ("Underflow" is the situation in which the result of an operation is not zero but is closer to zero than any floating point number.) The corresponding numbers are called "denormalized". Gradual underflow has the consequence that two floating point numbers are equal, $x = y$, if and only if subtracting one from the other gives exactly zero.

The use of denormalized (or "subnormal") numbers makes sense when you consider the spacing between floating point numbers. If we exclude denormalized numbers then the smallest positive floating pont number (in single precision) is $a = 2^{-126}$ (corresponding to $e = 1$ and $f = 00 \cdots 00$ (23 zeros)) but the next positive floating point number larger than a is b, which also has $e = 1$ but now has $f = 00 \cdots 01$ (22 zeros and a 1). Because of the implicit leading 1 in $1.f$, the distance between b and a is 2^{22} times smaller than the distance between a and zero. That is, without gradual underflow, there is a large and unnecessary gap between 0 and the nearest nonzero floating point number.

The other extreme case, $e = 255$, has two subcases, `inf` (for infinity) if $f = 0$ and `NaN` (for Not a Number) if $f \neq 0$. Both C and FORTRAN print[3] "`inf`" and "`NaN`" when you print out a variable in floating point format that has one of these values. The computer produces `inf` if the result of an operation is larger than the largest floating point number, in cases such as `x*x*x*x` when $x = 5.2 \cdot 10^{15}$, or `exp(x)` when $x = 204$, or `1/x` if $x = \pm 0$. (Actually $1/+0 =$ `+inf` and $1/-0 =$ `-inf`). Other invalid operations such as `sqrt(-1.)`, `log(-4.)`, and `inf/inf`, produce `NaN`. It is planned that f will contain information about how or where in the program the `NaN` was created but this is not standardized yet. It might be worthwhile to look in the hardware or arithmetic manual of the computer you are using.

Exactly what happens when a floating point exception, the generation of an `inf` or a `NaN`, occurs is supposed to be under software control. There is supposed to be a flag (an internal computer status bit with values 0 or 1) that the program can set. If the flag is on, the exception will cause a program interrupt (the program will halt there and print an error message), otherwise, the computer will just give the `inf` or `NaN` as the result of the operation and continue. The default in most situations is the worst of both actions. The computer generates an interrupt, which has enormous computational overhead (see chapter ??), then it puts in the `inf` or `NaN` without stopping to print an error message. In practice, it may be hard for the programmer to figure out how to set the arithmetic flags and other arithmetic defaults.

The floating point standard specifies that all arithmetic operations should be a accurate as possible within the constraints of finite precision arithmetic. The result of arithmetic operations is to be "the exact result correctly rounded". This means that you can get the computed result in two steps. First interpret the operand(s) as mathematical real number(s) and perform the mathematical operation exactly. This usually gives a result that cannot be represented exactly in IEEE format. The second step is to "round" this mathematical answer, that is, replace it with the IEEE number closest to it. Ties (two IEEE numbers the same distance from the mathematical answer) are broken in some way (e.g.

[3] In keeping with its mission to maximize incompatibility, Microsoft has changed the names `inf` and `NaN` to something similar but different.

round to nearest even). Any operation involving a `NaN` produces another `NaN`. Operations with `inf` are common sense: `inf` + *finite* = `inf`, `inf`/`inf` = `NaN`, *finite*/`inf` = 0., `inf` − `inf` = `NaN`.

From the above, it is clear that the accuracy of floating point operations is determined by the size of rounding error. This rounding error is determined by the distance between neighboring floating point numbers. Except for denormalized numbers, neighboring floating numbers differ by one bit in the last bit of the fraction, f. This is one part in $2^{23} \approx 10^{-7}$ in single precision. Note that this is relative error, rather than absolute error. If the result is on the order of 10^{12} then the roundoff error will be on the order of 10^5. This is sometimes expressed by saying that `z = x + y` produces $(x + y)(1 + \epsilon)$ where $|\epsilon| \leq \epsilon_{\text{mach}}$, and ϵ_{mach}, the "machine epsilon" is on the order of 10^{-7} in single precision.

Double precision IEEE arithmetic uses 8 bytes (64 bits) rather than 4 bytes. There is one sign bit, 11 exponent bits, and 52 fraction bits. Therefore the double precision floating point precision is determined by $\epsilon_{\text{mach}} = 2^{-52} \approx 5 \cdot 10^{-16}$. That is, double precision arithmetic gives roughly 15 decimal digits of accuracy instead of 7 for single precision. There are 2^{11} possible exponents in double precision, ranging from 1023 to −1022. The largest double precision number is of the order of $2^{1023} \approx 10^{307}$. Not only is double precision arithmetic more accurate than single precision, but the range of numbers is far greater.

Many features of IEEE arithmetic are illustrated in the program below. The reader would do well do do some of this kind of experimentation for herself or himself. The first section illustrates various how `inf` and `NaN` work. Note that e^{204} = `inf` in single precision but not in double precision because the range of values is larger in double precision. The next section shows that IEEE addition is commutative. This is a consequence of the "exact answer correctly rounded" procedure. The exact answer is commutative so either way the computer have the same number to round. This commutativity does not apply to triples of numbers because the computer only adds two numbers at a time. The compiler turns the expression `x + y + z` into `(x + y) + z`. It adds `x` to `y`, rounds the answer and adds the result to `z`. The expression `z + x + y` causes `z + x` to be rounded and added to `y`, which gives a (slightly) different result. Then comes an illustration of type conversion. Doing the integer division `i/j` gives 1/2 which is rounded to the integer value, 0, and then converted to floating point format. When `y` is computed, the conversion to floating point format is done before the division. Finally there is another example in which two variables might be expected to be equal but aren't because of inexact arithmetic. It is almost always wrong to ask whether two floating point numbers are equal. Last is a serendipitous example of getting exactly the right answer by accident. Don't count on this!

```
c   A program that explores IEEE arithmetic

c       Some double precision (8 byte or 64 bit) variables

      double precision xd, yd

c               C programmers note that other variables are declared
c               automatically and have default types.


c           Take an exponential that is out of range

      x = 204.
      y = exp(x)
      write(6,20) x,y
   20 format(' The exponential of ',e12.4,'   is ', e12.4)

c           Divide a normal number by infinity

      z = x/y
      write(6,40) x, y, z
   40 format(' ',e12.4,' divided by ',e12.4,' gives ',e12.4)

c           Divide infinity by infinity

      w = y
      z = w/y
      write(6,60) w, y, z
   60 format(' ',e12.4,' divided by ',e12.4,' gives ',e12.4)

c           Take the square root of a negative number

      z = sqrt( -1. )
      write(6,80) z
   80 format(' The square root of -1. is ',e12.4)

c            Add NaN to a normal number

      w = x + z
      write(6,100) x, z, w
  100 format(' ',e12.4,' plus ', e12.4,' gives ',e12.4)

c            Take the same exponential in double precision

      xd = 204.d0
```

```fortran
      yd = exp( xd )
      write(6,120) xd, yd
  120 format(' The double precision exponential of ',e12.4,' is ',e12.4)

c             Truncate a very large double precision number

      y = yd
      write(6,140) yd, y
  140 format(' The single precision version of ',e12.4,' is ',e12.4)

c            Explore the arithmetic, does addition commute?

      x =      sin(1.)
      y =  10.*sin(2.)
      z = 100.*sin(3.)
      s1 = x + y
      s2 = y + x
      s3 = x + z
      s4 = z + x
      if ( ( s1 .eq. s2 ) .and. ( s3 .eq. s4 ) ) then
         write (6,160) x, y, z
  160    format(' Adding pairs of   ',e12.4,', ',e12.4,', and ',e12.4,
     .                                    ' commutes')
      else
         write (6,180) x, y, z
  180    format(' Adding pairs of ',e12.4,', ',e12.4,', and ',e12.4,
     .                                    ' does not commute')
      endif

      s1 = x + y + z
      s2 = y + z + x
      s3 = z + x + y
      if ( ( s1 .eq. s2 ) .and. ( s2 .eq. s3 ) ) then
         write (6,200) x, y, z
  200    format(' Adding triples of ',e12.4,', ',e12.4,', and ',e12.4,
     .                                    ' commutes')
      else
         write (6,220) x, y, z
  220    format(' Adding triples of ',e12.4,', ',e12.4,', and ',e12.4,
     .                                    ' does not commute')
      endif

c             The result of an operation depends on the type of the operand

      i = 1
      j = 2
```

```
      x = i/j
      y = real(i)/real(j)
      write(6,240) i,j,x,y
  240 format(' Dividing ',i4,' by ',i4,' gives ',e12.4,
     .                 ' without conversion',/,
     .  '                        and ',e12.4,
     .                 ' with conversion')

c         Don't expect other arithmetic identities to be exactly true.

      x = z / 11
      y = x + x + x + x + x + x + x + x + x + x + x
      x = z / 5.
      w = 5*x
      if ( y .eq. z ) then
         write(6,260) z
  260    format(' Divided ',e12.5,' by 11 and got it back ',
     .                                     'by addition!!!')
      else
         write(6,280) z
  280    format(' Divided ',e12.5,' by 11 and did not get it back ',
     .                                     'by addition')
      endif
      if ( w .eq. z ) then
         write(6,300) z
  300    format(' Divided ',e12.5,' by  5 and got it back ',
     .                                     'by multiplication!!!')
      else
         write(6,320) z
  320    format(' Divided ',e12.5,' by  5 and did not get it back ',
     .                                     'by multiplication')
      endif

      stop
      end
```

This program produces the output

```
 The exponential of   0.2040E+03   is Inf
   0.2040E+03 divided by Inf          gives   0.0000E+00
 Inf          divided by Inf          gives NaN
 The square root of -1. is NaN
   0.2040E+03 plus NaN          gives NaN
 The double precision exponential of   0.2040E+03 is   0.3945E+89
 The single precision version of   0.3945E+89 is Inf
 Adding pairs of     0.8415E+00,   0.9093E+01, and   0.1411E+02 commutes
```

```
Adding triples of   0.8415E+00,   0.9093E+01, and   0.1411E+02 does not commute
Dividing    1 by    2 gives   0.0000E+00 without conversion
                        and   0.5000E+00 with conversion
Divided  0.14112E+02 by 11 and did not get it back by addition
Divided  0.14112E+02 by  5 and got it back by multiplication!!!
Note: the following IEEE floating-point arithmetic exceptions
occurred and were never cleared; see ieee_flags(3M):
Inexact;  Overflow;  Invalid Operand;
Note: IEEE Infinities were written to ASCII strings or output files; see econvert(3).
Note: IEEE NaNs were written to ASCII strings or output files; see econvert(3).
Sun's implementation of IEEE arithmetic is discussed in
the Numerical Computation Guide.
```