COMPUTER NUMBERS AND SOURCES OF ERRORS

MARIA CAMERON

Contents

1. Source of errors

References:

- Section 2.1 from D. Bindel's and J. Goodman's book "Principles of Scientific Computing".

Scientific computing has penetrated to various fields of science, humanities, and even pure mathematics. Computations play a crucial role in chemical physics, geophysics, planning of cancer treatment, developing of financial strategies, etc. Computations also became a tool for proving some theorems in pure mathematics. These are called computer-aided proofs. Unlike analytical formulas, results of computations almost always involve errors. In this course, we will learn some basics that will enable us to evaluate the results of numerical computations: are they sensible or nonsense, if they are sensible, how accurate they are, etc.

There are four major ways in which error is introduced into a computation:

- *Roundoff error* due to inexact computer arithmetic;
- *Truncation error* due to approximate formulas;

- *Termination error* in iterative methods;
- *Statistical error* in Monte Carlo methods.

Let us illustrate these kinds of errors.

1.1. **Roundoff error.** Here is a set of matlab commands for getting $\sqrt{2}$ and accessing the error of this calculation:

```
>> format long
>> a = sqrt(2)
a =  1.414213562373095
>> aa = a*a
aa = 2.000000000000000
>> d = aa - 2
d = 4.440892098500626e-16
```

We see that there is a discrepancy between 2 and a^2 which occurs due to the roundoff. Let us find the roundoff error of this calculation. The answer a can be written as $a = \sqrt{2} + \epsilon$ where ϵ is the roundoff error. Then

$$a^2 - 2 = \left(\sqrt{2} + \epsilon\right)^2 - 2 = 2\sqrt{2}\epsilon + \epsilon^2 = d.$$

Since ϵ is small, approximately 10^{-16}, we neglect ϵ^2 which is approximately 10^{-32} and find

$$\epsilon \approx \frac{d}{2\sqrt{2}} = 1.570092458683775 \cdot 10^{-16}.$$

For all practical purposed, this estimate of the roundoff error is good enough. However, if you want to see what happens if you solve the quadratic equation

$$\epsilon^2 + 2\sqrt{2}\epsilon - d = 0,$$

here is the result:

```
>> roots([1,2*sqrt(2),-d])
ans =
  -2.828427124746190
  -0.000000000000000
>> fprintf('%.15e, %.15e\n',ans(1),ans(2))
-2.828427124746189e+00, 1.570092458683775e-16
```

There appears a large extra root in this calculation that needs to be eliminated. The other root matches the previously found result.

1.2. **Truncation error.** Suppose we need to approximate the first derivative of a function f at a point x whose values are available only at the grid points x, $x \pm h$, $x \pm 2h$, etc. We will approximate it using so called *finite differences*. The commonly used *forward difference* scheme gives the following approximation:

$$f'(x) = \frac{f(x+h) - f(x)}{h}. \tag{1}$$

Let us estimate its error. We assume that the grid step h is small so that $h^2 \ll h$. Then we do Taylor expansion of $f(x+h)$ and get:

$$\begin{aligned}\frac{1}{h}\left(f(x+h)-f(x)\right) &= \frac{1}{h}\left(f(x)+f'(x)h+\frac{f''(x)}{2}h^2+\frac{f'''(x)}{6}h^3+O(h^4)-f(x)\right)\\ &= f'(x)+\frac{f''(x)}{2}h+\frac{f'''(x)}{6}h^2+O(h^4) = f'(x)+\frac{f''(x)}{2}h+O(h^2).\end{aligned}$$

This calculation shows that the truncation error, i.e., the difference between the exact f' and its approximation by the forward difference (2) is $0.5f''(x)h$ plus smaller terms. Since the largest term in expression for the error is proportional to h, i.e., h to the first power, we say that this is the finite difference approximation to f' is *first order accurate.*

Another common finite difference approximation for $f'(x)$ is the *central difference*:

$$f'(x) = \frac{f(x+h)-f(x-h)}{2h}. \tag{2}$$

Using Taylor expansion we obtain an error formula for it:

$$\begin{aligned}&\frac{1}{2h}\left(f(x+h)-f(x-h)\right)\\ =&\frac{1}{2h}\Bigg(f(x)+f'(x)h+\frac{f''(x)}{2}h^2+\frac{f'''(x)}{6}h^3+\frac{f^{(4)}(x)}{24}h^4+\frac{f^{(5)}(x)}{120}h^5+\\ &\quad -f(x)+f'(x)h-\frac{f''(x)}{2}h^2+\frac{f'''(x)}{6}h^3-\frac{f^{(4)}(x)}{24}h^4+\frac{f^{(5)}(x)}{120}h^5+\dots\Bigg)\\ =&f'(x)+\frac{f'''(x)}{6}h^2+O(h^4).\end{aligned}$$

Here the truncation error is proportional to h^2. Therefore, this central difference approximation for $f'(x)$ is *second order accurate.* Imagine that $h = 0.01$. Then, unless $f'''(x)$ is much larger that $f''(x)$, the second order scheme will give two more correct digits in comparison with the first order scheme.

1.3. **Termination error.** Iterative schemes are often used in scientific computing. Some problems can be solved only by iterations. For example, it is a well-known fact that there is no formula expressing roots of a polynomial of degree 5 or higher via its coefficients. The corollary of this fact is that there is no formula expressing eigenvalues of a square matrix of size 5×5 or larger via its entries. A practical consequence of this is that, in general, roots of a polynomial of degree n and the eigenvalues of an $n \times n$ matrix can be found only by iterative methods. Besides, some problems, such as solving large linear systems of algebraic equations, $Ax = b$, where the matrix A is sparse, can be solved at a finite number of steps e.g. by Gaussian elimination, but it might take way too much time and require prohibitively large storage for matrix entries at the intermediate stages of Gaussian elimination. In such cases, often iterative schemes for solving $Ax = b$ that give approximate solution of desirable accuracy are preferable. Moreover, such a solution is often reached at a rather small number of iterations.

Example Let us consider an example of iterative scheme called the Babylonian method for finding square roots. Later we will learn that this is the Newton-Raphson method for solving the nonlinear equation $x^2 - m = 0$. Start with some initial guess for $\sqrt{m}$, for example $x_0 = m$. Then iterate:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{m}{x_n}\right). \tag{3}$$

Stop when $|x_{n+1}^2 - m| < \epsilon$ where ϵ is the user-prescribed tolerance.

```
>> m = 2;
>> x = 2;
>> tol = 1e-15;
>> while abs(x*x - m) > tol x = 0.5*(x + m/x); fprintf('x = %.15e\n',x); end
x = 1.500000000000000e+00
x = 1.416666666666666e+00
x = 1.414215686274509e+00
x = 1.414213562374689e+00
x = 1.414213562373095e+00
```

The outcome is consistent with the one found by Matlab's root finder. In this example, the termination error coincides with the roundoff error because we have picked a stringent tolerance of 10^{-15}.

If we pick tolerance 10^{-9} we get:

```
>> tol = 1e-9;
>> x = 2;
>> while abs(x*x - m) > tol x = 0.5*(x + m/x); fprintf('x = %.15e\n',x); end
x = 1.500000000000000e+00
x = 1.416666666666666e+00
x = 1.414215686274509e+00
x = 1.414213562374690e+00
```

Here the error is mainly due to the termination of iterations. The error can be notably reduced by performing just one extra iteration.

On the other hand, if we pick the tolerance 10^{-16} which is smaller than the *machine epsilon*

```
>> eps
ans =  2.220446049250313e-16
```

(note that Matlab displays the machine epsilon multiplied by 2 if you type the command `eps`) the iterations will never stop because the roundoff error will prevent further error decay.

Scheme (3) raises the following questions:

(1) Do the iterations converge for any positive real number m and for any initial guess x_0 in exact arithmetic? Apparently it is so for $m = x_0 = 2$.
(2) If the answer to question 1 is 'yes', do they converge to $\sqrt{m}$?

(3) If the answer to question 1 is 'for all m but not for all x_0', what is the range of initial guesses for which the convergence will take place?

(4) The example with $m = 2$, $x_0 = 2$ shows that the iterations converge pretty fast. In general, how many iterations should we expect to perform in order to get the termination error approximately equal to the machine epsilon? The answer to this question is often given using to the concept of the *rate of convergence.*

We will address all these questions in the chapter on solving nonlinear equations.

1.4. **Statistical error.**

In some cases, it is infeasible to solve a problem using deterministic methods. Then Monte Carlo methods may come in handy. Monte Carlo methods are those in which random numbers are used to find something nonrandom. For example, calculation of integrals over high-dimensional regions or manifolds by discretizing the domain of integration may be impossible, however, Monte Carlo methods can give a satisfactory estimate using little effort. The results of Monte Carlo methods always involve statistical error.

1.5. **The absolute and relative errors.**

Definition 1. *Let A be the quantity that we want to estimate, and $\hat{A}$ be an estimate for A. Then the absolute error of this estimate is the difference*

$$e = \hat{A} - A,$$

while the relative error is

$$\epsilon = \frac{\hat{A} - A}{A}.$$

Note that then

$$\hat{A} = A + e = A(1 + \epsilon).$$

The relative error is often a more insightful quantification of error than the absolute error. For example, Avogadro's number, the number of constituent particles (usually molecules, atoms or ions) that are contained in one mole, is

$$N_0 = 6.02214076 \cdot 10^{23}.$$

It is often approximated by $6 \cdot 10^{23}$. The absolute error of this approximation is approximately $-2 \cdot 10^{21}$ which seems like a huge number while the relative error is about one-third of one percent which is acceptable for many practical purposes. This estimate will be good enough, for example, if the mass of the substance is known with a relative error of one percent.

2. Floating point arithmetic

References:

- Section 2.2 from D. Bindel's and J. Goodman's book "Principles of Scientific Computing";
- Lectures 7 and 8 in G. W. Stewart "Afternotes on Numerical Analysis".

2.1. **Integers.** The basic unit of computer storage is a bit, a binary unit, that can be 0 or 1. Bits are organized in 32-bit or 64-bit words. For example, in the Mac OS Catalina or later versions, only 64-bit word apps are acceptable. In C language, there are several integer types: `char`, `short`, `int`, `long`, etc. Most commonly, `char`'s range is from -128 to 127, `short` runs from -32768 to 32767. The typical range of `int` is from -2^{31} to $2^{31}-1$, i.e., from -2147483648 to 2147483647. The typical range of `long` is from -2^{63} to $2^{63}-1$ which is approximately from $-9.22\cdot 10^{18}$ to $9.22\cdot 10^{18}$. Note that if you keep increasing an integer variable in C and reach the maximal positive integer of this type, the next value will be the largest negative integer. For this reason, the loop in the C program below never stops:

```
#include <stdio.h>
int main(void);
int main() {
long j = 0;
int i;
for( i = 0; i < 3e9; i++ ) {
if( i == 0 ) printf("i = %i, j = %li\n",i,j);
j++;
};
return 1;
}
```

This program prints:

```
mariacameron@Marias-iMac Codes % gcc int_test.c -lm -O3
mariacameron@Marias-iMac Codes % ./a.out
i = 0, j = 0
i = 0, j = 4294967296
i = 0, j = 8589934592
i = 0, j = 12884901888
i = 0, j = 17179869184
i = 0, j = 21474836480
i = 0, j = 25769803776
i = 0, j = 30064771072
^C
mariacameron@Marias-iMac Codes %
```

I killed this run to terminate it. In Matlab, integers are automatically switched to floating-point numbers if you run out of integer range.

2.2. **Floating point numbers and the IEEE standard.** Floating point numbers are data types that approximate real numbers. The typical structure of a floating point number x is

$$x = (-1)^s \cdot 2^e \cdot m, \tag{4}$$

where s is the sign that can be 0 or 1, e is its exponent, and m is its mantissa. Double precision floating point numbers, i.e., the default type in Matlab and type `double` in C, use 64 bits to store floating point numbers. Out of them, one bit is used for sign, 11 bits for the exponent ranging from -1022 to 1023, and 52 bits for mantissa. Normalized floating-point numbers always have the first bit of mantissa equal to 1 which does not need to be stored for this reason. Hence, a normalized floating-point number is of the form

$$x = (-1)^s \cdot 2^e \cdot (1.b_1 b_2 \dots b_{52})_2$$

where b_i is 0 or 1, and the subscript 2 indicates that the mantissa is a binary number, i.e., its value is

$$(1.b_1 b_2 \dots b_{52})_2 = 1 + b_1 2^{-1} + b_2 2^{-2} + \dots + b_{52} 2^{-52}.$$

The *machine epsilon* is usually defined as *the half-distance between 1 and the next normalized floating-point number.* In double precision it is

$$\epsilon_m = 2^{-53} \approx 1.11 \cdot 10^{-16}.$$

The machine epsilon gives the upper bound for the relative roundoff error to the nearest floating-point number.

The IEEE floating-point standard is a set of conventions for computer representation and processing of floating-point numbers. Modern computers follow these standards for the most part. The standard has three 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.

2.3. **Modeling floating point error.**

There are five arithmetic operations in the floating point arithmetic:

(1) addition,
(2) subtraction,
(3) multiplication,
(4) division,
(5) taking a square root.

The IEEE standard for these operations is that the floating-point result is the *exact result, correctly rounded.* This means that if x and y are floating-point numbers, then the outcomes of arithmetic operations with them in the floating-point arithmetic are

$$\mathsf{fl}(x+y) = \mathsf{round}(x+y) = (x+y)(1+\epsilon_1); \quad \mathsf{fl}(x*y) = \mathsf{round}(x*y) = (x*y)(1+\epsilon_2);$$

$$\mathsf{fl}(x-y) = \mathsf{round}(x-y) = (x-y)(1+\epsilon_3); \quad \mathsf{fl}(x/y) = \mathsf{round}(x/y) = (x/y)(1+\epsilon_4);$$

$$\mathsf{fl}(\mathtt{sqrt}(x)) = \mathsf{round}(\sqrt{x}) = \sqrt{x}(1+\epsilon_5),$$

where round means rounding to the nearest floating point number, and $|\epsilon_i| \le \epsilon_m$, the machine epsilon, $i = 1, 2, 3, 4, 5$. Note that floating-point addition and multiplication are commutative. A sum of more floating point numbers is performed pairwise and typically left to right:

$$\begin{aligned}\mathsf{fl}(x_1+x_2+x_3) &= \mathsf{round}(\mathsf{round}(x_1+x_2)+x_3)\\ &= \left[(x_1+x_2)(1+\epsilon_1)+x_3\right](1+\epsilon_2)\\ &= (x_1+x_2+x_3)+(x_1+x_2)\epsilon_1+(x_1+x_2+x_3)\epsilon_2+(x_1+x_2)\epsilon_1\epsilon_2.\end{aligned}$$

The last term with the product $\epsilon_1\epsilon_2$ can be neglected because it is small relative to the other two error terms.

2.3.1. *Backward error analysis.* We can recast the result of $\mathsf{fl}(x_1+x_2+x_3)$ in a different manner that will illustrate for us the concept of *backward stability.* We will neglect terms with products of more that one ϵ:

$$\begin{aligned}\mathsf{fl}(x_1+x_2+x_3) &= \left((x_1+x_2)(1+\epsilon_1)+x_3\right)(1+\epsilon_2)\\ &= (x_1+x_2+x_3)+(x_1+x_2)\epsilon_1+(x_1+x_2+x_3)\epsilon_2\\ &= x_1(1+\epsilon_1+\epsilon_2)+x_2(1+\epsilon_1+\epsilon_2)+x_3(1+\epsilon_2)\\ &= x_1(1+\eta_1)+x_2(1+\eta_2)+x_3(1+\eta_3), \qquad (5)\end{aligned}$$

where $|\eta_1|=|\eta_2|\le 2\epsilon_m$, $|\eta_3|\le\epsilon_m$. Ansatz (5) shows that the result of the floating-point addition equals to *the exact sum of slightly perturbed numbers.* The relative backward errors η_i, $i=1,2,3$, are bounded by small multiples of ϵ_m. *An algorithm is called backward stable if backward error bounds are small.* Generalizing the above calculation to the sum of n floating point numbers we get:

$$\mathsf{fl}(x_1+x_2+\ldots+x_n)=x_1(1+\eta_1)+x_2(1+\eta_2)+\ldots+x_n(1+\eta_n), \qquad (6)$$

where

$$\begin{aligned}|\eta_1| &\le (n-1)\epsilon_m,\\ |\eta_2| &\le (n-1)\epsilon_m,\\ |\eta_3| &\le (n-2)\epsilon_m,\\ &\vdots\\ |\eta_{n-1}| &\le 2\epsilon_m,\\ |\eta_n| &\le \epsilon_m.\end{aligned}$$

It is reasonable to assume that the number of summands n is very small compared to ϵ_m^{-1} because the addition of 10^{16} numbers would take a really large time. Eq. (6) together with the expressions for η_i's show that *the addition of floating-point numbers is backward stable.*

2.3.2. *Condition number for floating-point sum.* Next, we will find the *condition number* of the addition of n numbers and use it to assess the relative error of the addition of floating-point numbers.

Definition 2. *Suppose we want to evaluate a function f depending on input data x. Suppose the relative error in the input data is ϵ. The condition number of the function f is defined as the supremum of the ratio of the relative error in f to the relative error in the input x:*

$$\kappa := \sup_x \frac{|f(x(1+\epsilon)) - f(x)|}{|\epsilon||f(x)|}. \tag{7}$$

Therefore, the relative error in f is bounded by:

$$\left|\frac{f(x(1+\epsilon)) - f(x)}{f(x)}\right| \le \kappa\epsilon.$$

Now we return to the floating-point sum of n numbers. The exact sum is

$$s_n = x_1 + \ldots + x_n.$$

The floating-point sum $\mathsf{fl}(s_n)$ is given by (6): this is the exact sum of slightly perturbed input numbers. Let us forget for a moment about the origin of these perturbations and merely think that we are finding a sum of perturbed numbers. The question that we want to address is how large is the relative error of the sum of perturbed numbers if the relative error of the perturbations is bounded by ϵ? We denote the sum of perturbed numbers by $\tilde{s}_n$, and the relative perturbations by μ_i's, and do the following calculation:

$$\begin{aligned}
\frac{|\tilde{s}_n - s_n|}{|s_n|} &= \left|\frac{x_1(1+\mu_1) + x_2(1+\mu_2) + \ldots + x_n(1+\mu_n) - (x_1 + \ldots + x_n)}{x_1 + \ldots + x_n}\right| \\
&= \left|\frac{x_1\mu_1 + x_2\mu_2 + \ldots + x_n\mu_n}{x_1 + \ldots + x_n}\right| \\
&\le \frac{|x_1||\mu_1| + |x_2||\mu_1| + \ldots + |x_n||\mu_n|}{|x_1 + \ldots + x_n|} \\
&\le \frac{|x_1| + |x_2| + \ldots + |x_n|}{|x_1 + \ldots + x_n|}\epsilon, \quad \text{where} \quad \epsilon = \max_{1 \le j \le n} |\mu_j|.
\end{aligned}$$

Therefore, the condition number of the sum of n numbers is

$$\kappa = \frac{|x_1| + |x_2| + \ldots + |x_n|}{|x_1 + \ldots + x_n|}. \tag{8}$$

Eq. (8) tells us, in particular, that if two numbers are close in magnitude but have opposite signs, their addition has a large condition number and hence is *ill-conditioned*. The relative error in the input number of such addition will be amplified by a large factor. On the other hand, if all numbers have the same sign, the condition number is 1, i.e., there will be no error amplification.

As we have established, the floating-point addition of n numbers gives the exact answer for slightly perturbed input. The relative error in this input is bounded by $(n-1)\epsilon_m$. Hence, the relative error of the floating-point sum of n numbers is bounded by

$$\frac{|\mathsf{fl}(s_n) - s_n|}{|s_n|} \le \frac{|x_1| + |x_2| + \ldots + |x_n|}{|x_1 + \ldots + x_n|}(n-1)\epsilon_m.$$

2.4. **Solving quadratic equations.** Let us consider the problem of solving quadratic equations in floating-point arithmetic. Suppose we have an equation of the form

$$x^2 - bx + c = 0, \quad \text{where} \quad b > 0.$$

Assume that $b^2 - 4c > 0$, so that the equation has two real roots. The familiar formula reads:

$$r_{1,2} = \frac{b \pm \sqrt{b^2 - 4c}}{2}. \tag{9}$$

In order to calculate root r_1 we first add b and $\sqrt{b^2 - 4c}$ and then divide the result by 2. The condition number for this addition is

$$\kappa_1 = \frac{|b| + \left|\sqrt{b^2 - 4c}\right|}{\left|b + \sqrt{b^2 - 4c}\right|} = 1,$$

hence this addition is well-conditioned. To calculate r_2, we subtract $\sqrt{b^2 - 4c}$ from b and then divide the result by 2. The condition number for this subtraction is

$$\kappa_2 = \frac{|b| + \left|-\sqrt{b^2 - 4c}\right|}{\left|b - \sqrt{b^2 - 4c}\right|} = \frac{b + \sqrt{b^2 - 4c}}{\left|b - \sqrt{b^2 - 4c}\right|}.$$

If $|c| \ll b^2$ then the denominator is approximately equal to

$$\left|b - \sqrt{b^2 - 4c}\right| \approx \left|b - b + \frac{2c}{b}\right| = \left|\frac{2c}{b}\right|,$$

while the enumerator is approximately $2b$. This gives an estimate for κ_2:

$$\kappa_2 \approx \left|\frac{b^2}{c}\right|,$$

which is large. Therefore, the calculation of r_2 by formula (9) might lead to significant amplification of roundoff errors. A remedy for this issue is the use of a different formula for r_2:

$$r_1 = \frac{b + \sqrt{b^2 - 4c}}{2}, \quad r_2 = \frac{c}{r_1} = \frac{2c}{b + \sqrt{b^2 - 4c}}. \tag{10}$$

The example above with $|c|$ being small compared to b^2 shows how a calculation can be salvaged by formula rewriting. Unfortunately, nothing much can be done to boost up accuracy when the problem is intrinsically ill-conditioned. This situation, for example occurs when the discriminant is close to zero, i.e, $r_1 \approx r_2$. In this case, a small variation in the coefficient c of size ϵ will lead to variation in roots approximately proportional to $\sqrt{\epsilon}$ which is much larger. Some variations may lead to the disappearance of real roots. We will return to this issue in the chapter on solving nonlinear equations.

2.5. **Roundoff and recurrence relationships.** Let us discuss an interesting example from G. W. Stewart "Afternotes on Numerical Analysis". Consider the recurrence relationship

$$x_{k+1} = 2.25x_k - 0.5x_{k-1} \tag{11}$$

with initial condition $x_0 = 1/3$, $x_1 = 1/12$. The general solution to (11) is of the form

$$x_k = Ar_1^k + Br_2^k$$

where A and B are arbitrary constants and r_1 and r_2 are the roots of the characteristic equation

$$r^2 - 2.25r + 0.5 = 0.$$

Solving it, we find $r_1 = 1/4$ and $r_2 = 2$. Hence, the solution to the recurrence relationship is

$$x_k = \frac{A}{4^k} + B2^k.$$

Plugging in the initial conditions, we find $A = 1/3$, $B = 0$. Hence, the solution

$$x_k = \frac{1}{3 \cdot 4^k}$$

decays to zero. However, the numbers $x_0 = 1/3$, $x_1 = 1/12$ are not represented exactly using binary numbers. Hence, instead of them we have $1/3(1+\epsilon_1)$ and $1/12(1+\epsilon_2)$ where $0 < |\epsilon| < \epsilon_m$, $i = 1, 2$. Hence, the constant B is likely to turn out to be nonzero when (11) is iterated in the floating-point arithmetic. As a result, the solution will eventually blow up. Here is a C program demonstrating this phenomenon:

```
#include <stdio.h>
int main(void);
int main() {
double x0 = 1.0/3.0,x1 = 1.0/12.0,x = 0;
int k = 1;
while( x < 1.0 ) {
x = 2.25*x1 - 0.5*x0;
x0 = x1;
x1 = x;
k++;
printf("x[%i] = %.15e\n",k,x);
}
return 1;
}
```

Its printout is:

```
mariacameron@Marias-iMac Codes % gcc recurrence.c -lm -O3
mariacameron@Marias-iMac Codes % ./a.out
x[2] = 2.083333333333334e-02
x[3] = 5.208333333333356e-03
```

```
x[4] = 1.302083333333381e-03
x[5] = 3.255208333334285e-04
x[6] = 8.138020833352365e-05
x[7] = 2.034505208371398e-05
x[8] = 5.086263021594628e-06
x[9] = 1.271565756730925e-06
x[10] = 3.178914418472664e-07
x[11] = 7.947286579088713e-08
x[12] = 1.986822710586282e-08
x[13] = 4.967078092747778e-09
x[14] = 1.241812155751090e-09
x[15] = 3.105383040660637e-10
x[16] = 7.780510627309835e-11
x[17] = 1.979233708143944e-11
x[18] = 5.630205296689555e-12
x[19] = 2.771793376831781e-12
x[20] = 3.421432449526730e-12
x[21] = 6.312326323019251e-12
x[22] = 1.249201800202995e-11
x[23] = 2.495087734305776e-11
x[24] = 4.989346502086499e-11
x[25] = 9.978485762541735e-11
x[26] = 1.995691971467565e-10
x[27] = 3.991382647674935e-10
x[28] = 7.982764971534822e-10
x[29] = 1.596552986211588e-09
x[30] = 3.193105970399332e-09
x[31] = 6.386211940292704e-09
x[32] = 1.277242388045892e-08
x[33] = 2.554484776088621e-08
x[34] = 5.108969552176451e-08
x[35] = 1.021793910435271e-07
x[36] = 2.043587820870536e-07
x[37] = 4.087175641741071e-07
x[38] = 8.174351283482141e-07
x[39] = 1.634870256696428e-06
x[40] = 3.269740513392857e-06
x[41] = 6.539481026785713e-06
x[42] = 1.307896205357143e-05
x[43] = 2.615792410714285e-05
x[44] = 5.231584821428570e-05
x[45] = 1.046316964285714e-04
x[46] = 2.092633928571428e-04
```

```
x[47] = 4.185267857142856e-04
x[48] = 8.370535714285713e-04
x[49] = 1.674107142857143e-03
x[50] = 3.348214285714285e-03
x[51] = 6.696428571428570e-03
x[52] = 1.339285714285714e-02
x[53] = 2.678571428571428e-02
x[54] = 5.357142857142856e-02
x[55] = 1.071428571428571e-01
x[56] = 2.142857142857142e-01
x[57] = 4.285714285714285e-01
x[58] = 8.571428571428570e-01
x[59] = 1.714285714285714e+00
mariacameron@Marias-iMac Codes %
```

Let us use the printout data to estimate the roundoff for $1/3$. We have:

$$x_0 = 1/3(4^{-0} + 2^{-p}),$$
$$x_k = 1/3(4^{-k} + 2^{-p+k}), \quad k = 1, \dots$$

The turning point is reached at $k = 19$, therefore,

$$4^{-19} = 2^{19-p}. \quad \text{Hence} \quad p = 57.$$

This gives the roundoff error of 2^{-57} for $1/3$.

This example might seem artificial, however, it is very relevant to analysis of linear multistep ODE solvers. These solvers boil down to iterating recurrence relationships. A necessary and sufficient condition for their stability is that all characteristic roots satisfy $|r| \le 1$, and those with $|r| = 1$ have algebraic multiplicity 1.

2.6. **Summary.** We have seen three ways in which the roundoff error can manifest itself:

- Slow accumulation line in the sum of n floating point numbers. A notable loss of accuracy may occur only if the problem is ill-conditioned.
- Roundoff error can reveal itself as a result of cancellation like in the formula for roots of a quadratic equation. In some cases, this problem can be fixed by rearranging the formula.
- Roundoff error can be magnified like in the example with the recurrence relationship. An algorithm involving iterations of relationships with characteristic roots $|r| > 1$ or roots with $|r| = 1$ with multiplicity more than 1 is unstable and should be avoided.

2.7. **Exceptions: underflow and overflow.** Please read Section 2.2.4 from D. Bindel's and J. Goodman's book "Principles of Scientific Computing".