One, Two, Three, . . . , ∞?

D. E. Stevenson and D. D. Warner

January 6, 1999

The purpose of this module is to examine the effects of finiteness on mathematical computations. The effects are known as *errors*, which may be a bit prejudicial. These errors occur due to the following sources:

- Truncation errors. Using the computer requires that we use an approximate formulation for an exact but infinite formulation. For example, we often use a finite Taylor series instead of the infinite one. The requirement is that an unending *process* is replace by one that finitely terminates.
- Discretization errors. Descretization occurs when an infinite set is replaced by one that is finitely parametrizable. This is not the same thing as truncation. For example, an infinite function space is replaced by a finite one.
- Perturbation errors. Because of the approximation errors, we end up with a solution slightly perturbed version of the problem. Thus, the inputs and functions are not computed exactly, causing the program to be slightly altered.
- Inferential structure errors. An algorithm has its own unique behavior. Two formulas which are *algebraically* equivalent may not produce the same answer and therefore cause different decisions to be made.
- Rounding errors. Rounding errors are the response the machine to the exact instance at hand.
- Bugs. Bugs are a type of error caused by humans.

Our goal in this module is to concentrate on two types of errors: (1) *trucation errors* that occur by changing an infinite formulation to a finite one rounding errors and (2) *rounding errors* caused by using finite arithmetic.

1 A Problem to Think About

This module also illustrates a particular style of teaching calld *problem based learning* or PBL. In PBL, we use a problem to direct out investigation; this is in contrast to lecturing, where the instructor gives out

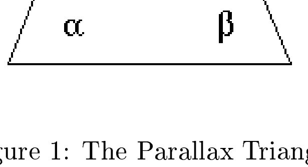

Figure 1: The Parallax Triangle

information. Lecturing is sometimes said to be the process of taking information from the mind of the professor and onto the notes of the students without ever having either think about the information.

1.1 The Model

In order to guide our investigation, we look to an age-old problem. One way to measure distance is to use *parallax*. Parallax uses a triangle in which one side and the two angles at the end of the known side can be measured (Figure 1). Let c be the measured side and A, B be the measured angles.
Exercise. Before you go on, take a few minutes to formulate the equation(s) that are needed to solve the problem.

$\Diamond$

We want to find the height h from c to the vertex. We can do that in two steps. First, we note that $h = b \sin A$. Next, note that b can be found by using the Law Sines:

$$\frac{c}{\sin C} = \frac{b}{\sin B}.$$

Therefore, we can get $b = c \sin C / \sin B$. Since we have A and B, we find $C = \pi - (A + B)$ in radians. So,

$$\begin{aligned} h &= \frac{c \sin B}{\sin C} \sin A, \\ &= \frac{c \sin B}{\sin \pi - (A + B)} \sin A, \\ &= \frac{c \sin B}{\sin (A + B)} \sin A. \end{aligned}$$

Aside. Note that we are making two assumptions based on *Euclidean* geometry. Firstly, that the Law of Sines holds and secondly, that the sum of the angles of a triangle is π radians. In hyperbolic space, the "Law of Sines" is $\sinh c / \sin C = \sinh b / \sin B$. The third angle of a triangle in hyperbolic geometry is given by

$$-\cos C = \cos A \cos B - \sin A \sin B \cosh \cosh c.$$

Elliptical geometries are different again.

$\Diamond$

Question. We to want to know, given the errors of our measuring devices *and* the finiteness of computer arithmetic, how much confidence can we put into the computation of h. There are several ways to look at this, but for this module we ask the question this way: Given a particular accuracy in the physical measurement, can the computer *correctly* compute h?

$\Diamond$

1.2 Error Measurement and Quantifying Error

We know from our own experience that whenever we take repeated measurements we do not get the exact value for whatever we are measuring. The instrument may be out of adjustment, our ability to make precise movements of the mechanism and inability to read the measurement are common problems. But how do we deal with this mathematically? We will assume that there is a "correct" value of a variable, call it x.
Exercise. We are now stuck with a thorny problem: what do we mean by *correct*. Take a few minutes and discuss this with your team members.

$\Diamond$

1.3 Types of Errors

Suppose then we call the measurement $\hat{x}$. One way to talk about error is to look at the difference:

$$\text{signed absolute error} = x - \hat{x}.$$

With this type error, we know not only the difference but which way the difference goes. However, we know from experience that dealing with signed variables can be difficult and lead to confusion. Let us call *absolute error* the absolute value of the signed absolute error:

$$\text{absolute error} = |\text{signed absolute error}| = |x - \hat{x}|.$$

Example. Suppose that $x = 1.5 \times 10^{15}$ and $\hat{x} = 7.55 \times 10^{15}$ Then the signed absolute error is $1.5 \times 10^{15} - 7.55 \times 10^{15} = -6.05 \times 10^{15}$ and the absolute error is 6.05×10^{15}. The problem with absolute error is that it does not capture what we think about when we discuss errors. In most engineering and science applications we ignore *small* errors and worry about *large* errors. Is 6.05×10^{15} large or small? The observation is that we can't tell directly. One way out of this is to look at the *relative error*: take the absolute error and divide by the *correct* value. In this case, this value is $6.05 \times 10^{15}/1.5 \times 10^{15} = 4.03\bar{3}$. Surely, we would say this is a large error.

The relative error δ_x, then, is defined as

$$\delta_x = \frac{|x - \hat{x}|}{x}.$$

This obviously will not work if $x = 0$. In that case, the absolute error can be used. There really cannot be a fixed definition of small and large—these are examples of *fuzzy values*. However, we would usually think of small errors as those relative errors down in the thousandths or ten-thousandths. Anything larger than a relative error of one is probably large.

Summary. So we have a sort-of criterion for correct. A computation is correct to a relative error, usually pre-stated in the problem statement.

◊

1.4 Relative Error in Parallax

Extending the concept of relative error from a single variable to a function comes naturally from the calculus. What is the natural cognate of difference? Why sure, differentiation. We can start by looking at the derivation of derivative a bit differently. Recall that

$$\frac{f(x + \Delta x) - f(x)}{\Delta x} = \frac{\Delta f}{\Delta x} = f'(x) + o(\Delta x).$$

Put another way:

$$\Delta f \approx f'(x)\Delta x.$$

That is, we can use Δf as our concept signed absolute error. If $f(x) \neq 0$ then the relative error defintion makes sense:

$$\delta_f = \frac{f'(x)}{f(x)}\delta_x.$$

Things are only slightly more complicated if we have multiple independent variables. In this case we have to take the sum of the partial derivatives for each variable $x_1, \ldots, x_n$:

$$\delta_f = \Sigma_{i=1}^{n} \frac{\partial f}{\partial x_i}\delta_{x_i}.$$

Using these definitions, we can derive δ_h for parallax. The relative error with respect to c is easy to interpret: δ_f is changes little when c is large.

Exercises. Find the relative error of the following formulas:

1. $f(x) = mx + b$
2. $f(x) = e^{2x+5}$
3. $f(t) = t^3 \cos t$
4. Our problem formulation.

◊

2 Conditioning

There are other ways to explore how functions behave and how they relate to the problem at hand. The next concept, that of *conditioning* comes about as follows. We would like to know how changes in the parameters affect the changes in a function. But our only concept of change so far is relative error. Relative error says "The size of the change in the function value compared to the function value is 'relative error."' We want to also know how much the change in the value is "blown up."

2.1 Condition Numbers

To contrast the two ideas, think of the traditional concept of calculus, that of the small change Δx. We can use this idea to develop the traditional calculus concept of derivative provided we cheat a bit and assume that we know that a function can be expanded into a Taylor series. First, we start with the $y = f(x)$. Then,

$$\begin{array}{rcl} y + \Delta y & = & f(x + \Delta x) \\ & = & f(x) + f'(x)\Delta x + \ldots \\ -y & = & f(x) \\ \hline \Delta y & = & f'(x)\Delta x \end{array} \tag{1}$$

In other words, "y changes by the product of the value of the derivative times the change in the parameter."

Now, let's use the same basic approach but from the point of view of *relative change*. Remember that we have defined relative change by relating the "correct" value to the "perturbed" value. If x is the correct value and $\hat{x}$ is the perturbed value, then the relative error, call it δ_x, is

$$\delta_x = \frac{x - \hat{x}}{x}.$$

Solving for $\hat{x}$ we get

$$\hat{x} = x(1 + \delta_x).$$

This is true regardless of the arithmetic system as long as the term $(1 + \delta_x)$ is not 1. Therefore, we could take this as a *axiom* if we wanted to.

Now let's retrace our steps through the derivation of the derivative using $x(1 + \delta_x)$ for $x + \Delta x$ and similar changes for y.

$$\begin{array}{rcl} y(1 + \delta_y) & = & f(x(1 + \delta_x)) \\ & = & f(x) + f'(x)(x\delta_x) \ldots \\ -y & = & -f(x) \\ \hline y\delta_y & = & f'(x)(x\delta_x) \end{array}$$

Solving for δ_y,

$$\begin{array}{rcl} \delta_y & = & \dfrac{f'(x)(x\delta_x)}{y} \\ & = & x\dfrac{f'(x)}{f(x)}\delta_x. \end{array} \tag{2}$$

This last equation gives us a first order relation for relative changes in y given a change in x (and *vice versa*). The *relative conditioning of y with respect to x* is denoted $C_R(y; x)$ and is defined by

$$C_R(y; x) = \frac{x\frac{df(x)}{dx}}{f(x)}.$$

Examples. As a first example, let $y = a$, a is a constant. Then

$$\begin{array}{rcl} C_R(y; x) & = & \dfrac{x \cdot 0}{a}, \\ & = & 0. \end{array}$$

For our next example, take the linear function $y = mx + b$. Then

$$C_R = \frac{xm}{mx+b}.$$

If b and m are finite, then the series converges by the ratio test. Notice, though, that the behavior is not uniform. For example, let $b >> m$ and compare x close to 1 and x close to b. An easy example of this relationship is when $m = 1$ and $b = 0$. This is the *identity function.* In this case, we get that $C_R(y; x) = 1$.

On the other hand, let's look at e^x.

$$\begin{aligned} C_R &= \frac{xe^x}{e^x}, \\ &= x. \end{aligned}$$

◊ This tells us that things get "wilder" the further out we go—which is exactly what happens. If the absolute value of C_R is large, then we say that the problem is *ill-conditioned*; otherwise, it is *well-conditioned.*

2.2 Chain Rule for Condition Numbers

Since the condition number is related to the derivative, we should expect the condition number to share properties of derivatives. Of importance is the concept of the *chain rule.* Suppose $y = f(x)$ and $w = g(y)$. What is $C_R(w; x)$? Proceeding directly,

$$\begin{aligned} C_R(w;x) &= \frac{xg'(y)}{g(y)}, \\ &= \frac{xg'(y)}{g(y)} f'(x), \\ &= \frac{g'(y)}{g(y)} x f'(x), \end{aligned}$$

We can introduce y/y to re-introduce the form of the condition number. Since $y = f(x)$ we get

$$\begin{aligned} C_R(w;x) &= \frac{yg'(y)}{g(y)} x \frac{f'(x)}{f(x)}, \\ &= C_R(w;y) C_R(y;x) \end{aligned} \tag{3}$$

We can interpret the condition numbers in light of the total computation. In the first instance, low condition numbers means low cancellation of significant digits (see below). Here, of course, "low" is taken to mean relative to the inputs as in 2.

Computation is really a series of compositions: each step composes previous results by the current statement. Seen in this way, we can look at the condition numbers for parts of the computation and derive inferences on how various blocks of code interact.

Looking back at the identity function, we see that its condition number is 1. Let's consider a program which first computes a function and then its inverse — this is not as rare a problem as might seem at first blush.

$$C_R(f \circ f^{-1}; x) = C_R(ident; x) = 1.$$

This basically says that the overall computation is well-conditioned but this does not mean that each of the individual pieces is well-conditioned! A well-known problem has to do with multiple roots of a function. Let's take a look at the quadratic with multiple roots.

$$(x-a)(x-a) = x^2 - 2ax + a^2$$

which has a condition number of

$$C_R(quadratic; x) = x\frac{2x-2a}{x^2-2ax+a^2} = \frac{2x}{x-a}.$$

So how well conditioned is the system close to a? Not very good!
Exercise. Perform the conditioning analysis of our parallax problem.

◊

2.3 Summary

At this point we have developed quite a bit of machinery

1. Absolute error for both values and functions,
2. Relative error for both values and functions,
3. Relative conditioning for constants, functions, and composition of functions.

We have applied these ideas to our parallax problem. We have learned that relative error is a measure of accuracy (and therefore correctness). We have learned that the condition number tells us something about how well behaved the problem is (well- or ill-conditioned).

To this point, though, we have not calculated one thing! We now turn to the number system.

3 Introduction and The Wilkinson model

The purpose of this section is to explore machine arithmetic in terms that the user (not the machine designer or the compiler writer) is concerned about. This section is really all about *scientific notation.*

Exercise. As an individual, take five minutes and write down all the rules you can remember or generate about scientific notation. After these five minutes, work with your group to come up with one list of properties.

$\Diamond$

3.1 Background and notation definitions

The purpose of this module[1] is to explore the Wilkinson Floating Point Arithmetic model. This will take us through the mathematical as well as the engineering aspects of computer arithmetic. We (and most other workers) call this system the *Wilkinson model* after J. H. Wilkinson whose pioneering working was done in the 1950s [1].

We use, as everyone knows, the *arabic notation*[2]. We write numbers in a positional way. Suppose

$$N = d_{n-1}d_{n-2}\ldots d_0 \cdot d_{-1}\ldots d_{-m}$$

represents a number. We have n *digits* to the left of the decimal point and m digits to the right of the decimal point. This notation stands for the following *series*:

$$N = d_{n-1} \times 10^{n-1} + \ldots + d_0 \times 10^0 \cdot d_{-1} \times 10^{-1} + \ldots + d_{-m} \times 10^{-m}.$$

In a more compact form, we have

$$N = \sum_{i=-m}^{n-1} d_i \times 10^i.$$

We can generalize this form of notation by the following: we call the d_i's *digits*[3]. The generalization of the 10 is called the *base b* or sometimes the *radix.* So we can represent a number as

$$N = \sum_{i=-m}^{n-1} d_i b^i$$

under the general condition that $0 \leq d_i < b$. Because we can have the same number in many different bases(radices), we often indicate the base as a subscript as in the following example:

$$\begin{aligned} 100_2 &= 1 \times 2^2 = 4 \text{ but} \\ 100_8 &= 1 \times 8^2 = 64. \end{aligned}$$

[1]These notes are due to D. D. Warner, Department of Mathematical Sciences, Clemson University, Clemson, SC. They were presented in a course in the fall of 1990. They have been extended by D. E. Stevenson.

[2]Actually, the notation seems to have been developed in India and exported to Western Asia by way of Babylon.

[3]Short for "decimal units".

The digit d_{n-1} is called the *most significant digit* while the digit d_{-m} is called the *least significant digit.* The *precision* of the number is the number of digits: $n + m + 1$.

For most scientific work, we use the *scientific notation.* In scientific notation, the number is usually written in the form $d_1.d_2 \cdots d_n \times 10^e$. For example,

$$1234.567 \text{ is written } 1.234567 \times 10^3.$$

In most computers (especially true for IEEE 754 computers), we use *normalized floating point notation.* Floating point representation is the machine equivalent to the *scientific notation* used in scientific work. In computers, we think of the number as being written in the form $0.d_1 d_2 \cdots d_n \times 10^e$

$$1234.567 \text{ is written } 0.1234567 \times 10^4.$$

This representation goes back to Zuse (1941), Stibitz (1947) and Aiken (1947). The value of this form of storage is that the programmer does not get involved with scaling. (See McCracken, Chapter 4 and especially exercises 14-17).

To define the machine representation, we choose the following parameters:

1. A *radix* or *base* of representation, b, like 10 or 2.
2. A fixed number of units (in the radix) to represent the *fractional part* or the *mantissa*, t.
3. A fixed number of radix units to represent the *exponent.* The are two: a low e_min and a high e_max.

In addition to these parameters, there are two storage issues

1. How the exponent is actually stored.
2. How the mantissa is stored.

The storage and the actual physical layout are the providence of computer engineers. For our purposes, IEEE Standard 754 determines how the representation must appear to the user.

Aside. We will not attempt to go into the engineering issues. However, the terminology occurs often enough in the literature that we take a quick aside to explain some of the more common terms.

There are three standard ways to interpret the mantissa:

1. sign-magnitude. In signed magnitude, the number is stored with one unit as the sign and the remaining units as digits. This is similar to the way we normally write a negative number: say, -123.45.
2. signed radix complement. In signed radix complement, the numbers are stored following algorithm. Suppose N is the number to be stored and $\overline{N}$ represents the number actually stored. Then

   $$\overline{N} = (radix)^n - N,$$

   where n is the number of units in the mantissa. For example,

   $$\overline{123.45_{10}} = 876.55_{10'scomplement}.$$

   In signed radix complement, -123.45 would be -876.55.
3. signed reduced-radix complement. In reduced-radix complement the number is stored in complement of $radix - 1$ For our general model of n integer digits and m fractional digits, the reduced radix value, denoted $\overline{\overline{N}}$ is defined by

   $$\overline{\overline{N}} = (radix - 1)^n - N - (radix)^{-m}.$$

Since the exponent is an integer, it can be represented in one of the above ways or with what is known as *excess* notation. In excess notation, the actual value is stored after a constant has been added. This makes the stored value always positive. there is no known optimal way to choose these parameters and representations.

Because there are many different representations, the concept of "adding two numbers" is an algorithm specific to the particular method of representation. This also brings up a very interesting theoretical issue:

Question. *Given all the current methods for representing floating point numbers, are these methods equivalent? In this case,* equivalent *systems would be able to represent exactly the same numbers. Can they all do the same arithmetic operations?*

$\diamond$

3.2 Wilkinson Set

We want to define a set of objects called *numbers* and define some operations that will work *similar* to the way we would like to believe the *real* numbers work. We want these numbers to act like they were defined and manipulated in scientific notation. We begin with the numbers.

3.2.1 The Theoretical Wilkinson Set

We have two major elements. There is the *mantissa* which is the "decimal" part and there is the *exponent*. The values which can be stored in these two pieces represents the universe of numbers available to us.

The values which can be represented in a floating point number are zero and all the rational numbers describable by the below parameters. We define $\mathcal{W} = \mathcal{W}(b, t, e_{min}, e_{max})$ as follows:

b. b is the base of the arithmetic. For example, most non-IBM clones are base 2, while IBM and its clones are base 16.

t. t is the number of b-units in the mantissa.

e_{min}. e_{min} is the *smallest* number which can be stored in the exponent. *I. e.,* the smallest exponent value is $b^{e_{min}}$

e_{max}. e_{max} is the *largest* number which can be stored in the exponent. *I. e.,* the largest exponent value is $b^{e_{max}}$

Based on this, we can visualize the numbers as

$$\pm[\frac{\beta_1}{b} + \frac{\beta_2}{b^2} + \ldots + \frac{\beta_t}{b^t}] \times b^e,$$

where

$0 < \beta_1 < b$,

$0 \le \beta_k < b$ for $k = 2, 3, \ldots, t$, and

$e_{min} \le e \le e_{max}$.

The first condition is the *normalization* condition. In normalized arithmetic, the first b-unit must be non-zero; the other $t-1$ digits could be zero. This representation is called *quasi-logarithmic.* If this were *logarithmic*, then

$$N = b^{(exponent+mantissa)} = b^{exponent} b^{mantissa}.$$

In quasi-logarithmic systems,

$$N = b^{(exponent+mantissa)} = (b^{exponent}) \times coeffient.$$

An7 set which is defined with the above scheme is called a *Wilkinson set.*
Exercise. Write down all the numbers with the definition $\mathcal{W}(2, 3, -2, 2)$.

$\diamond$

Let's consider the model $\mathcal{W}(2, 3, -2, 2)$. This is the case when we use base two arithmetic, there are three bits in the mantissa and the exponents run from $[-2, 2]$. The first thing to get used to is that the "decimal point" (more properly called a "radix point") is really a "binary point." The first place to the right of the binary point must always be one [Why?] Then there are two values each for the second and third bits. So, the possible values are $\{0, .100, .101, .110, .111\}$. These values, base 10, are $0, 1/2, 5/8, 3/4, 7/8$. There certainly are not many mantissas to work with.

Well, what about the exponents? They run $2^{-2}, 2^{-1}, 2^0, 2^1, 2^2 = 1/4, 1/2, 1, 2, 4$. The smallest number, then, that we can deal with in this model is $1/4 \times 1/2 = 1/8$ while the largest is $4 \times 7/8 = 7/2$.
Exercise. Using standard graph paper and semilog graph paper, graph the elements of $\mathcal{W}(2, 3, -2, 2)$. What conclusions can you draw?

◊

In general, there will be a smallest and a largest number in the number system. Let σ be the smallest and λ be the largest.
Exercise. Show that $\sigma \equiv b^{e_{min}-1}$ and that $\lambda = b^{e_{max}}(1 - b^{-t})$. *Hints:* σ telescopes.

◊

3.2.2 The Tests of the Wilkinson Set

We want to test all this theory against a real machine. The below discussion revolves around the *VAX 8800*-series machines. The first test is the check the mantissa. The *VAX* has a thirty-two bit word which is ostensively broken into an eight bit exponent and a twenty-four bit mantissa. The exponent uses *excess-64* arithmetic which means that numbers below sixty-four are considered negative exponents, those above are positive and sixty-four itself is considered zero[4]. The mantissa has one bit for a sign and twenty-three bits usable. But, since the *Vax* uses normalized arithmetic only, it can simulate $t = 24$ bit arithmetic. The Wilkinson model for the *Vax* is

$$\mathcal{W}(2, 24, -128, 127).$$

These can be encoded as parameters in the test program. We will use `matlab` as the vehicle. Unfortunately for documentation purposes, `matlab` does not have declarations. However, we can at least make them comments.

⟨Parameters ?⟩ ≡

```
% integer base. This is the base from above, and so should be 2.
% integer tdigits, emin, emax. These are the remaining parameters.
% Matlab is C based and has taken the C idea of always doing
% computations in double precision. The values for double precision
% are those given below

base =     2;
tdigits = 53;
emax =  1023;
emin = -1024;

◊
```

Macro referenced in scrap ?.

Our first test is to try the expression $1 - b^{-t}$ to see if it is correct. The implementation of `matlab`'s exponentiation may not be correct, so we want to test that it is.
H.ow do you think exponentiation is done? Here's part of the way it might be done

[4]This is a common and generally useful trick

⟨Naive Power Function ?⟩ ≡

```
function f = pow(a,b)
% this does powering function independent of `^'. For integers only.
f = 1;
if b ~= 0
   ndx = abs(b);
   for i = 1:ndx
        f = f*a;
   end;
   if b < 0
      f = 1.0/f;
   end
end
◊
```

Macro never referenced.

Even with this, there is still a question. For the $b < 0$ case, should we invert a and then power or do it the way it is done here?

◊

We do that by defining `lowman` as the value of $1 - b^{-t}$ and then seeing if $1 - b^{-(t+1)}$ is still 1. We make use of this fact even though it is not discussed until Section 4.2

⟨Declarations ?⟩ ≡

```
% double lowman. lowman will have the value of 1-epi.
% double epsi. Compute the term for the subtraction.
◊
```

Macro defined by scraps ?, ?, ?.
Macro referenced in scrap ?.

Now go ahead and compute both ways and check the error. It better be zero.

⟨Compute Machine Epsilon Two Ways ?⟩ ≡

```
% Check the machine epsilon.
epsi = pow(base,-tdigits+1)

lowman = 1.0 - epsi;

t  = 1.0 - 1.0/(base^(tdigits-1))

error = (t-lowman)/lowman

if (t-lowman)==0
   disp('Naive calculation is done correctly.')
else
   disp('Naive calculation is not done correctly.')
end;
◊
```

Macro never referenced.

Notice that `matlab` also has the machine epsilon as a parameter. Make sure that this is also correct.

⟨Check eps and epsi ?⟩ ≡

```
if eps ~= lowman
   disp('"eps" is  correct.')
else
   disp('"eps" is not correct.')
end;

◊
```

Macro never referenced.

Now we have a handle on the machine epsilon. What happens when we try to compute the next smaller power of two? We should get 1.

The next test computes λ. This is somewhat tricky on real machines. The reason, of course, is that the computation will overflow if done as written. *I. e.,* $(1 - b^{-t})b^{e_{max}}$ involves computing $b^{e_{max}}$ which is too large. To circumvent this problem, we "sneak" up on the value by first computing $\lambda/2$ and then multiplying that value by 2.

⟨Declarations ?⟩ ≡

```
% double lambda. This will be the value of the largest representable value
◊
```

Macro defined by scraps ?, ?, ?.
Macro referenced in scrap ?.

There are two different subprojects here. One is to get the largest possible mantissa. This would be the mantissa that has all 1s in the maintissa but with a positive sign.

⟨Lambda Test ?⟩ ≡

```


maxman = 0;
man = 1;
for i=1;tdigits
    man = man/2.0;
    maxman = maxman + man;
end;
◊
```

Macro defined by scraps ?, ?, ?.
Macro referenced in scrap ?.

Exercise. How do we know that we got it right?

⟨Lambda Test ?⟩ ≡

```
test = maxman + epsi
◊
```

Macro defined by scraps ?, ?, ?.
Macro referenced in scrap ?.

Explain the value of `test`

◊

Given the value of the mantissa, we should be able to multiply by $base^{emax}$ and not overflow. *But* it should also be the case that $\mathtt{maxman} * \mathtt{base^emax} + \mathtt{eps} * \mathtt{base^emax}$ should also overflow.

⟨Lambda Test ?⟩ ≡

```
lambda = maxman*base^emax
lambda2 = lambda
overflow1 = lambda*base
overflow2 = lambda2+eps*base^emax
◊
```

Macro defined by scraps ?, ?, ?.
Macro referenced in scrap ?.

Now it's time to test σ. The test proceeds in exactly the same manner as with λ. In this case, the mantissa is simple: it's 1. The question before us is whether or not the machine has *subnormalizing* or not. The computed version of $sigma = b^{emin}$. Anything smaller than that that is not zero is caused by subnormalization.

⟨Declarations ?⟩ ≡

```
%double sigma
%integer iexp      integer iexp
◊
```

Macro defined by scraps ?, ?, ?.
Macro referenced in scrap ?.

Now we compute the σ directly and compare

⟨Sigma Test ?⟩ ≡

```
sigmat = base^emin;
small = sigmat;
cnt = 0;
tester = 1;
while small ~= 0
    cnt = cnt+1;
    small = small/2.0;
    error = (sigmat-small)/sigmat;
    tester = tester /2.0;
    error2 = small/sigmat;
    if error2 ~= tester & error2 ~= 0
       disp('error2 and tester out of sync')
       disp(tester), disp(error2)
       break;
       end;
    end;

sprintf('the number of places that are subnormalized is %d\n' , cnt)

◊
```

Macro referenced in scrap ?.

Finally, the whole program.

`"wilkinson.m"` ? ≡

```
        ⟨Parameters ?⟩
        ⟨Declarations ?, ... ⟩
        ⟨Lambda Test ?, ... ⟩
        ⟨Sigma Test ?⟩
◊
```

4 The rules for arithmetic in a perfect world

4.1 Axioms for Real Numbers

Having defined the numbers, we want to define how the operations work. In order to do that, we need to look at the target—the set of real numbers. We indicate the set of reals by **R**. In the case of the real numbers, we have a set of axioms to guide us. We have two standard operations to consider: addition and multiplication. We also have two *inverse operations*: the *additive inverse* or subtraction and the *multiplicative inverse* or division. Rather than develop all the underlying theory, we simply state the axioms of the reals[5]:

For all real numbers x, y, and z we have

Ax1 Additive Closure. $x + y$ is a real number.

Ax2 Additive Commutativity. $x + y = y + x$.

Ax3 Additive Associativity. $x + (y + z) = (x + y) + z$.

Ax4 Additive Identity. There exists a unique number 0, called the *additive identity*, with the property that $x + 0 = x$ for any real number x.

Ax5 Additive Inverse. For any real number x, there is a real number $-x$ with the property that $x+(-x) = 0$. $-x$ is called the *additive inverse of* x.

Ax6 Multiplicative Closure. xy is a real number.

Ax7 Multiplicative Commutativity. $xy = yx$.

Ax8 Multiplicative Associativity. $x(yz) = (xy)z$.

Ax9 Multiplicative Identity. There exists a unique number 1, called the *multiplicative identity*, with the property that $x1 = x$ for any real number x.

Ax10 *Multiplicative Inverse.* For any real number $x \neq 0$, there is a real number w^{-1} with the property that $xw^{-1} = 1$. x^{-1} is called the *multiplicative inverse of* x.

Ax11 Distributivity of Multiplication over Addition. $x(y + z) = xy + xz$.

An analysis of the axioms with respect to the Wilkinson set. Our next task is to see how our model of the floating point numbers fits in. Unfortunately, it does not fit very well.

Let's start with **Ax1**. Remember, that for our definition to be a model of the real numbers, **Ax1** must hold for any two model numbers. To see that this is not the case, look at the sum $\lambda + \lambda$. Using our example $\mathcal{W}(2, 3, -2, 2)$ we see that the sum of the two largest numbers cannot be represented. Not much we can do about this situation and maintain a finite set. Of course, other values will fail as well. The same fate should befall **Ax6**: if you multiply λ by itself, then the result is unrepresentable. But there are other problems with **Ax6**. The number 3/4 and $3/8 = (3/4)(1/2)$ are each representable. The product is 9/32 which is not representable. But $1/4 \leq 9/32 \leq 3/8$ and each limit is representable. There is some hope, then, if we define addition and multiplication correctly that we could have a usable system.

Clearly, **Ax3**, **Ax8**, and **Ax11** all have trouble with largest element, λ. Notice too that $\lambda^{-1} = 1/\lambda$ is not in the set. We have

$$\begin{aligned}
\lambda^{-1} &= [b^{e_{max}}(1 - b^{-t})]^{-1}, \\
&= \frac{b^{-e_{max}}}{1 - b^{-t}}, \\
&= b^{-e_{max}} \sum_{k=0}^{\infty} b^{-kt}
\end{aligned}$$

[5]Technically, the reals for a *field*. We are not going to develop the algebraic underpinings. See H. L. Royden, *Real Analysis*. New York:Macmillan. 1968.

There seems to be no good alternative definition since there is a true loss of information. On the other hand, $\sigma\sigma = b^{-2e_{min}-2}$ is not in the set either. But in this case, it makes sense to force something like this to 0.

Even **Ax4** is not without its problems. Depending on how we choose to deal with the representation problems pointed out in **Ax1** and **Ax6** we might have problems. For example if we use a modulo system we introduce some strange number combinations as zeros.

4.2 Machine Epsilon Discussion with simple program

The machine epsilon ϵ plays an important role in understanding the behavior of the numbers on the computer. We emphasize that the word *machine* in important: the value of the machine epsilon is a function of the machine being used and sometimes even the compiler. We define the machine epsilon, ϵ, to be the smallest number such that $fl(1+\epsilon) > fl(1)$ A simple program for determining the machine epsilon is below:

4.3 Machine Epsilon Program

"epsilon.m" ? $\equiv$

```
%double precision dbase
dbase = 2;

% integer ri, dpi
ri= 0;
dpi = 0;
% double double dpepsi,dpres
realepsi = 1.0;
dpepsi = 1.0;

while (1d0 + dpepsi) > 1.0
      dpres = dpepsi;
      dpepsi = dpepsi / dbase;
      dpi = dpi + 1;
      end;

sprintf("The double residue is %.17e\n", dpres);
sprintf("The double epsilon is %.17e\n", dpepsi);
sprintf("The double precision power is %.17e\n", dpi);
sprintf("Now compute final values\n");

tdepsi = 1d0/(2d0**(dpi-1));

error = (tdepsi-depsi)/depsi;

if error == 0.0
   sprintf("The computed and counted values are the same\n")
else
   sprintf("The computed and counted values are the NOT the same, error = %.17e\n",error)
end
◊
```

Exercise.. For all machines at your disposal, determine the machine epsilon.

◊

4.4 Gill machine epsilon program with explantion

There are some subtle problems with the machine epsilon program. These have to do with the very last division. We want the machine epsilon to represent the *the largest relative spacing between numbers.* Using the paramters of the model, we see that

$$\epsilon = b^{(1-t)}.$$

In the simple program, it is possible to compute a number other than this. This is especially true for machines that round up preferentially. An alternative program is presented below. The first module computes the *estimate* of ϵ. The second module employs an important "trick" known as *Gill's device.*

"gill.m" ? $\equiv$

```
% double teps, eps1
⟨Compute Estimate ?⟩
⟨Make Gill Move ?⟩
sprintf("Machine epsilon = %.17e\n",teps)
◊
```

The Compute Estimate module computes the epsilon approximation as we did in the naive computation earlier.

⟨Compute Estimate ?⟩ $\equiv$

```
teps = 1
eps1 = 2.0
while eps1 > 1.0
      teps = teps/2e0;
      eps1 = teps +1.0;
      end;
◊
```

Macro referenced in scrap ?.

The following code now computes the "correction" to the app

⟨Make Gill Move ?⟩ $\equiv$

```
teps = 2*teps;
eps1 = teps+1e0;
teps = eps1-1e0;
◊
```

Macro referenced in scrap ?.

5 The function fl

In order to solve some of these problems, we define the function fl for "float". fl maps the rational numbers, denoted $\mathbb{Q}$, to the Wilkinson set. We will call such a system as

$$(\mathcal{W}(b,t,e_{min},e_{max}),fl) \equiv \mathcal{F}(b,t,e_{min},e_{max})$$

a *floating point system.* We define fl by the following: if $x \in \mathbb{Q}$ then

1. if $x \in \mathcal{W}$ then $fl(x) = x$;
2. if $\sigma \leq x \leq \lambda$ then $fl(x)$ is the model number $x' \in \mathcal{W}$ closest to x. This strategy is called the *correctly rounded strategy*[6]
3. if $\sigma \leq x \leq \lambda$ and x is halfway between two model numbers, then any tie breaking strategy can be used.

Notice that we don't say what happens to numbers outside the range $[\sigma, \lambda]$. In order to make this a Scott domain (this is a requirement for computational purposes in computer science), we add two new conditions on fl:

1. if $x < -\lambda$ then $fl(x) = \perp = -\infty$ and

[6]There is a strategy called *chopped* which "truncates" by choosing the model number closer to zero. IEEE 754 also has two other rounding modes.

2. if $x > \lambda$ then $fl(x) = \top = \infty$.

In theoretical work, it is important to prove that fl is a computable function.
Exercise. Write a program which takes in two integers which represents the numerator and denominator of a rational number and returns as its value the fl of the input.

$\Diamond$

To make the system complete, we need to define the arithmetic operations. For this, we have the basic definition: the operations are done over $\mathbb{Q}$ and the fl is applied. In order to analyze what happens, we need to develop some understanding of numerical (round-off) errors.

The fundamental relationship of floating point arithmetic. Let $\circ$ be a binary operator define on the real numbers and $\odot$ be its floating point implementation. Then

$$\begin{aligned} fl(a \circ b) &\approx (fl(a) \odot fl(b))(1+\varepsilon) \\ &= (a \circ b)(1+\varepsilon) \end{aligned}$$

6 Errors and Conditioning

We will focus on the idea of estimating the error. There is an alternative: Interval arithmetic. This has been fashionable in Europe and has even been put into a `Pascal` dialect called `Pascal-XSC`. We will not pursue this alternative here.

6.1 Errors

As we have already seen, there are numerical values we cannot represent and there are operations on values which lead to small losses in value. In this section, we try to put some definiteness to these ideas.

Turning again to our function fl, we see that there is a slight difference, between a real number x and its value in $\mathcal{W}$. We call the difference $x - fl(x)$ the *absolute error* of $fl(x)$. The absolute error is misleading in many cases since the absolute magnitude may be very large. A more meaningful measure is how much error there is relative to the *correct* value. Thus, we call

$$\frac{x - fl(x)}{x}$$

the *relative error of* $fl(x)$ *in* x. The numerical value of the relative error expressed in scientific notation is an indicator of the digit in error. If the relative error is expressed as $ddd \times 10^n$, then the "10^nth digit" is the digit in error. Thus, if the relative error is 1×10^{-3} then the third digit to the left of the decimal point is the digit in error. As an example, let's look at π and its approximation 22/7:

$$\begin{aligned} 22/7 &= 3.142857142857\ldots, \\ \pi &= 3.1415926535\ldots, \\ \pi - 22/7 &= .001264\ldots, \text{ and} \\ \frac{\pi - 22/7}{\pi} &= .000402\ldots = .402 \times 10^{-3} \end{aligned}$$

This says that the third digit to the right of the decimal point, 10^{-3}, is the first digit in error and indeed it is: 3.142 *versus* 3.141. We can use this test on individual points or on functions. In order to simplify notation, we want to notationally differentiate between the *correct* or *true* value of a variable and its *approximated* or *computed* value. The convention is that the computed value is indicated by placing a "hat" over the computed term. For example, if x is the true value, $\hat{x}$ is the computed value.

7 The Floating Point Operations

Now that we have a set of values we can define operations on these sets. The general approach is to carry out the arithmetic in the rational numbers and the use fl to choose the floating point number. Since in most cases there will not be a floating point number equal to the computed rational number, we will have some round-off error. We can use the error analysis and conditioning concepts to study how floating point arithmetic behaves.

First, a note on representation. In general, the difference between a value computed on the rational numbers and the related floating point number differs by less than one machine epsilon. The absolute difference is, of course, dependent on the numbers involved. On the other hand, the computed (floating) value is related to the true (rational) value by the expression

$$\hat{x} = fl(x) \approx x(1+\epsilon).$$

7.1 Addition and Subtraction

We naturally begin with addition and subtraction. To define addition, we assume that a and b are true values. Let $c = a + b$ with $c \in \mathbb{Q}$ and $\hat{c}$ therefore is the result. How good an approximation is $\hat{c}$? From the relative error view, we have that $\hat{c} = (a+b)(1+\epsilon)$.

Problems arise, however, when we use values which themselves have been computed. Let's compute the relative error for addition with $\hat{a}$ and $\hat{b}$.

$$\begin{aligned}
\hat{c} &= \hat{a} + \hat{b}, \\
&= a(1+\epsilon_a) + b(1+\epsilon_b), \\
&= a + b + (\epsilon_a + \epsilon_b). \\
\text{Then} & \\
c - \hat{c} &= a + b - (a + b + (\epsilon_a + \epsilon_b)), \\
&= -(\epsilon_a + \epsilon_b), \\
\text{and} & \\
\frac{c - \hat{c}}{c} &= \frac{-(\epsilon_a + \epsilon_b)}{a+b}.
\end{aligned}$$

We return to this expression later.

7.2 Multiplication and Division

Multiplication and division follow the same thought line as we used in the definition of addition and subtraction. That is, that for exact a and b, we have $\hat{c} = a \cdot b(1+\epsilon)$. What about the relative error?

$$\begin{aligned}
\hat{c} &= \hat{a} \cdot \hat{b}, \\
&= a(1+\epsilon_a) \cdot b(1+\epsilon_b), \\
&= a \cdot b + a \cdot b \cdot \epsilon_a + a \cdot b \cdot \epsilon_b + a \cdot b \cdot \epsilon_a \cdot \epsilon_b.\text{Then} \\
c - \hat{c} &= a \cdot b - (a \cdot b + a \cdot b \cdot \epsilon_a + a \cdot b \cdot \epsilon_b) + a \cdot b \cdot \epsilon_a \cdot \epsilon_b, \\
&= -(a \cdot b \cdot \epsilon_a + a \cdot b \cdot \epsilon_b) + a \cdot b \cdot \epsilon_a \cdot \epsilon_b. \\
\text{And} & \\
\frac{c - \hat{c}}{c} &= \frac{-(a \cdot b \cdot \epsilon_a + a \cdot b \cdot \epsilon_b + a \cdot b \cdot \epsilon_a \cdot \epsilon_b)}{a \cdot b}, \\
&= -(\epsilon_a + \epsilon_b) + \epsilon_a \cdot \epsilon_b.
\end{aligned}$$

Comparing the relative errors of addition with multiplication, we see one glaring difference. The relative error in multiplication is independent of the values of a and b whereas that of addition is not. The consequences of this for addition are dire.

7.3 Cancellation of Significance in Addition

7.3.1 Total Error

Programs usually have more than one variable. When we have many variables, there is always a question of which one(s) have what effect on the final error. To deal with this problem, we turn to multivariate calculus.

In what follows, we use the following convention: if $f(x_1, \ldots, x_n)$ is a function of n variables then

$$\phi_{x_i} = \frac{\partial \phi}{\partial x_i}$$

is the "partial derivative of ϕ with respect to the variable x_i."

For multivariate functions, the concept of "derivative" is a bit different from the single variable case. For the multivariate case, it makes little sense to speak in terms of the change of ϕ with respect to a single variable since the function may change in many "directions" at once. Therefore, we want to expand the concept of derivative to take in these multiple directions. Therefore, we talk of the "(total) differential" of ϕ at a point. Notationally, we write

$$d\phi = \phi_{x_1} dx_1 + \phi_{x_2} dx_2 + \cdots + \phi_{x_n} dx_n,$$

where dx_i is the size of the "step" along the x_i direction. Using this, we can ask, "What is the total relative error in $y = \phi(x_1, \ldots, x_n)$ Let's derive the formula by following the definitions using the derivative forms. Initially, we have

$$dy = d\phi = \phi x_1 dx_1 + \phi x_2 dx_2 + \cdots + \phi x_n dx_n.$$

To get the relative error in y we need to divide the left and right sides by y. Since y and ϕ are the same, we can write the above as

$$\varepsilon_y = \frac{dy}{y} = \frac{\phi_{x_1}}{\phi} dx_1 + \frac{\phi_{x_2}}{\phi} dx_2 + \cdots + \frac{\phi_{x_n}}{\phi} dx_n.$$

Now, in order to get ε_y in terms of the relative errors of the x_is, we multiply and divide the ith term on the right by its associated variable x_i. In so doing, we can write $\varepsilon_{x_i} = dx_i/x_i$. This final expression is just our earlier statement using *condition numbers.* Let ϕ be a function of n variables. Then

$$\begin{aligned}
\varepsilon_y &= x_1 \frac{\phi_{x_1}}{\phi} \varepsilon_{x_1} + x_2 \frac{\phi_{x_2}}{\phi} \varepsilon_{x_2} + \cdots + x_n \frac{\phi_{x_n}}{\phi} \varepsilon_{x_n}, \\
&= C_R(\phi, x_1)\varepsilon_{x_1} + C_R(\phi, x_2)\varepsilon_{x_2} + \cdots + C_R(\phi, x_n)\varepsilon_{x_n}, \\
&= \sum_{i=1}^{n} C_R(\phi, x_i)\varepsilon_{x_i}.
\end{aligned}$$

Example. As an example, what is the total error for

$$y = a + b + c?$$

The condition number for the first term is $a/(a + b + c)$ and the error of the first term is ε_a. The other two terms are similar. Thus,

$$\varepsilon_y = \frac{1}{a + b + c}(a\varepsilon_a + b\varepsilon_b + c\varepsilon_c)$$

Then the sum $a + b + c$ is well-conditioned if a, b, and c are all small with respect to the sum.

Now, what if there are many output variables? That is, each algorithm computes many values; how are we to judge the final state? In this case, one should think "vector valued functions." That is, each state variable has a function applied to all other state variables. For this discussion we use boldface fonts to indicate vectors and normal mathematics font to set scalars. For example,

$$\mathbf{y} = \begin{bmatrix} \phi_1(x_1, \ldots, x_n) \\ \vdots \\ \phi_m(x_1, \ldots, x_n) \end{bmatrix} = \phi(\mathbf{x})$$

Thus, $\mathbf{y}$ is a vector written $[y_1, \ldots, y_m]^T$. The value of $\mathbf{y}$ is produced by $\phi(\mathbf{x}) = [\phi_1, \ldots, \phi_m]^T$. Each ϕ_i is applied to the vector $\mathbf{x} = [x_1, \ldots, x_n]^T$.

Now that we have the ability to talk about multiple outputs, we can turn our attention to the issue of *multiple steps.* Algorithms usually have multiple steps and we indicate step number as a parenthesized superscript. For example, the function at the ith step produces the $(i + 1)$th state:

$$\mathbf{y}^{(i+1)} = \phi^{(i)}(\mathbf{x}).$$

Lastly, we must ask how are we to think about the execution of these multiple steps? Suppose there are $(r+1)$ steps in a computation. The number of variables would be constant across the $(r+1)$ steps. Let $\mathbf{x}^{(0)}$ be the initial values. Then,

$$\begin{aligned} \mathbf{x}^{(1)} &= \phi^{(0)}\mathbf{x}^{(0)}, \\ \mathbf{x}^{(2)} &= \phi^{(1)}\mathbf{x}^{(1)}, \\ &\vdots \\ \mathbf{x}^{(r+1)} &= \phi^{(r)}\mathbf{x}^{(r)}. \end{aligned}$$

In other words, the entire computation is an r-fold composition of the intermediate functions $\phi^{(i)}$:

$$\phi = \phi^{(r)} \circ \phi^{(r-1)} \cdots \phi^{(0)}.$$

Example. Suppose we want to compute $y = (a^2 - b^2)$ using a computer restricted to add, subtract, multiply, and divide. One way would be to square the two individually and then subtract. To do so, we need two intermediate values (variables) and one final value to hold the result. So our state vector is five values $= [a, b, u, v, y]^T$. Using $\perp$ to indicate a value not yet computed, $\mathbf{x} = [a, b, \perp, \perp, \perp]^T$. The step functions are defined by the following:

$$\begin{aligned} \phi^{(0)}(a, b, u, v, y) &= [\hat{a}, \hat{b}, \hat{a^2}, \perp, \perp]^T, \\ \phi^{(1)}(a, b, u, v, y) &= [\hat{a}, \hat{b}, \hat{a^2}, \hat{b^2}, \perp]^T, \\ \phi^{(2)}(a, b, u, v, y) &= [\hat{a}, \hat{b}, \hat{a^2}, \hat{b^2}, u \hat{-} v]^T. \end{aligned}$$

$\Diamond$

Now that we have a way of describing a program, we want to re-derive the error expressions.

8 Numerical Stability

An algorithm is *numerically stable* if the accumulation of round-off errors is “harmless;” *i. e.,* these errors do not swamp the information in the answer with noise. For each problem, one can make these ideas precise by asking about the conditioning between the inputs and the answers.

9 Exercises

9.0.2 Exercises: Folk Theorems

1-1 Find the largest value of a such that

$$\frac{1}{1+a} \approx 1 - a.$$

Before you proceed, you must decide how you are going to define ‘$\approx$’. Does the approximation above hold uniformly from that point on? That is, let a_0 be the value determined in Problem 1-1. Is it true that for all $a < a_0$ that the approximation still holds?

1-2 The sun subtends an angle at the earth of $32'4''$. The distance to the sun is approximately 92 million miles. Estimate the diameter of the sun. Find the currently accepted distance to the sun and recalculate the diameter of the sun. What is the error in using the 92 million mile estimate for the distance to the sun?

1-3 Using the data from 1–2, how far must the eye be from a penny so as to just hide the sun? A penny is 3/4 of an inch in diameter.

1-4 A railroad is inclined $53'$ above the horizontal. How many feet does the track rise in one mile?

1-5 The tangent of the sum of two angles x and y is

$$\tan(x \pm y) = \frac{\tan x + \tan y}{1 - \tan x \tan y}.$$

Investigate the behavior for this sum when $x \approx \pi/2$ and $y \approx 0$.

1-6 A common problem is typified by trying to compute the average of two numbers, say A and B. The naive way is to set Avg by

$$Avg = (A + B)/2.0.$$

Consider the different values that the two variables can take on and comment and the alternative statement:

$$Avg = A + (B - A)/2.0.$$

1-7 Given your favorite machine, what is the range of values which can be used as arguments for the combination formula

$${}_NC_n = \frac{N!}{(N-k)!k!}?$$

9.0.3 Exercises in Counterexamples for Algebraic Properties

In general, the following properties of the real numbers do not hold for the floating point numbers. Explore these laws by finding cases which are counter examples for these laws and run them on your favorite computer.

1. *Cancellation Law 1.* If $ac = ab$ and $a \neq 0$ then $b = c$.
2. *Cancellation Law 2.* $(a + b) - a = b$.
3. *Cancellation Law 3.* For $a \neq 0$, $a(b/a) = b$.
4. *Distribution Law.* $a(b + c) = ab + ac$ and $(b + c)a = ba + ca$.
5. *Tricotomy Law.* For every pair of real numbers a and b, then one of the following is true: $a < b$ or $a = b$ or $a > b$.
6. *Order Law 1.* If $a < b$ then for all c, $a + c < b + c$.
7. *Order Law 2.* If $a < b$ and $c < d$ then $a + c < b + d$.
8. *Order Law 3.* If $b < c$ and $a > 0$ then $ab < ac$.

The three order law examples are not obvious. First you need to prove that strict inequalities cannot be guaranteed. That is, that for floating point, *Order Law 1* must be written as

Order Law 1. If $a < b$ then for all $c, a + c < b + c$.

Show that the equation $ax = 1$ does not have a solution whenever the mantissa of a is $1 - b^{-p}$, where b is the base and p is the precision.

Show that the equation $ax = 1$ always has a solution if $a > 0$ and the mantissa of a, m, is given by

$$b^{-1} \leq m \leq \frac{1 + b^{-1}}{2}.$$

Horner's rule has been known for centuries. In Horner's rule, we transform a computation over the "series" written as

$$y(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0$$

into a computation of the form

$$y(x) = (\cdots (a_n x + a_{n_1})x + \cdots + a_1)x + a_0.$$

Use the Taylor series expansion for arctan to write a program testing the value produced by the two different approaches.

9.0.4 Mesh Points

Many programs must compute a *mesh*, a sequence of N values taken at $1/N$ subintervals on a given interval. Two methods come to mind for computing a mesh. The first would be something like

"mesh1.f" ? ≡

```
              program mesh1
              integer I,N
              real    A,B,H,Mesh(0:N)
              N = value
              H = (B-A)/N
              Mesh(0) = A
              do I = 1,N
                 Mesh(I) = Mesh(I-1)+H
              end do
              stop
              end
```

◊

Another possible method might be the following:

"mesh2.f" ? ≡

```
              program mesh2
              integer I,N
              real    A,B,H,Mesh(0:N)
              N = value
              H = (B-A)/N
              Mesh(0) = A
              do I = 1,N
                 Mesh(I) = A + I*H
              end do
              stop
              end
```

◊

Explore these two possible approaches. In theory, `Mesh(N)` should hold `B`; is this the case?

9.0.5 Counter Examples

Compilers and machines often are counter intuitive. The below program may or may not work, depending on the exact configuration.

"Counter.f" ? ≡

```
            program counter examples
            real a,b,c,d,e,f,g
            a = 1.0/3.0
            b = 1.0/5.0
            c = a+a+a
            d = b+b+b+b+b
            e = 1.0 - c
            f = 1.0 - d
            g = e / f
            print*, g
            end
```

◊

9.0.6 Computing Series

9.0.7 Computing the Geometric Series

In the next example, we compute the geometric series. In this program, we read in a value for r and ϵ and we compute the sum of the series using a formula for the number of iterations developed below. We also compute the series based on the summation formula. Finally, we print out the theoretical number n and the required number k.

To determine the number of iterations needed to obtain a given level of accuracy by solving the inequality

$$|\frac{a}{1-r} - \frac{a(1-r^k)}{1-r}| \leq \epsilon.$$

"Convergence.f" ? $\equiv$

```
      program converge
      double precision dpn,r,a,epsilon
      double precision psum, rsum
      integer k,n
 10   continue
      read*,r,epsilon
      if(r.eq.0) stop
      print*, r,epsilon
      a = 1
      rsum = a/(1-r)
      dpn = dlog( epsilon*(1-r)/a) / dlog(r)
      n = dpn+1
      print*,'estimate ',n
      k = 0
 1    continue
      psum = a*(1 - r**k) / (1-r)
      k = k+1
      if( dabs(rsum-psum) .ge. epsilon) goto 1
      k = k-1
      print*,n,k
      goto 10
      end
```
◊

9.0.8 Dealing with Elementary Functions

Just learned about a new rounding error trick from Kahan. The problem is: ln(1+x) and exp(x)-1 are well conditioned for x near zero, but calculating them as written is unstable. The trick in the past has been to switch over to the series in the vicinity of zero, but knowing how many terms to take depends on knowing the precision of the machine. If the ln and exp provide values accurate to machine precision then there is a simple formula which will provide the answer for these related functions which is accurate to within 3 rounding units and doesn't use series and doesn't require a knowledge of the machine's precision. I'll make the answer available next time we meet.

9.1 Dealing with the Elementary Functions

The elementary functions are not immune to the vagaries of floating point computations. The most obvious problems are well covered in elementary calculus: the problems of remainders and truncation. However, these functions are also susceptible to round-off problems as well. In particular, consider the following question: what is the largest value of x such that $sin(x)$ is zero? At the other end of the scale, we can ask: what is the largest value of x which can be used in $sin(x)$ such that the computation does not overflow? We call these values σ_f and λ_f.

Construct a table, for your favorite machine, of the values of σ_f and λ_f for all the intrinsic functions provided by your Fortran compiler.

9.2 Powers of X

"Powers.f" ? ≡

```
      function rep(x,n)
      real rep,x
      integer n
      integer i
c
      rep = 1
      do i = 1,n
         rep = rep*x
      enddo
      return
      end
      function rep2(x,n)
      real rep2,x,p
      integer n,l,ln,k
c
      p=x
      ln = n
      rep2=1.0
 100    continue
        l=ln/2
        k=ln-2*l
        if(k.eq.1) rep2=rep2*p
        ln=l
        if(ln.eq.0) goto 200
        p=p**2
        goto 100
 200  continue
      return
      end

      function nest(x,n)
      real nest,x
      integer n,ln,ll,l
c...  compute ll
      ll = 1
      ln = n
      l = 0
 50   continue
      if( ll .gt. n ) goto 75
        ll = ll*2
        l = l+1
        goto 50
 75     continue
      print*,'l,ll=',l,ll
      nest=1
      do 100 i=1,l
         nest=nest*nest
         ln = ln*2
         if( ln.lt. ll) goto 100
         nest=nest*x
         ln=ln-ll
 100     continue
      return
      end

      program test
      real nest,a1,a2,a3
c
      a1=rep(10.0,7)
      print*,'a1=',a1
      a2=rep2(10.0,7)
```

This program tests the algebraic identity $x * (1/x) = 1$ by stepping through the first $1,000,000$ numbers. It prints out the number of times that the identity is not true.

"anothertest.f" ? ≡

```
      program test
      real x,y,z,err,merr
      integer d,c
      merr=0
      c=0
      do d=1,1000000
         x = d
         y = 1/x
         z = x*y
         if( z .ne. 1 ) then
            err = abs(z-1)
            c=c+1
            merr=amax1(merr,err)
         endif
      end do
      print*,c,merr
      stop
      end
```

◊

References

[1] J. H. Wilkinson. *Rounding Errors in Algebraic Processes.* Wiley, New York, 1963.