Errors and Accuracy

Leigh J. Little SUNY Brockport October 5, 2018

Contents

1 Computing Errors

Computers have limited precision. This means that every arithmetic calculation has the possibility to introduce errors. There are two main sources for these errors.

- Storage errors. These occur whenever a number that can't be represented exactly in binary is stored in the computer's memory. As it turns out, most decmial numbers fit into this category. For example, the value $a = (1.1)_{10}$ has the infinite repeating binary expansion $a = (1.00011001100....)_2$. At some point, this sequence of digits will need to be truncated, which can lead to a storage error of up to 1 bit in the last binary place.
- Round-off errors. Round-off errors occur whenever the results of a calculation has more digits than can be stored on a computer. For example, consider a fictional computer that uses base 10 (similar to a standard TI calculator) and can store 5-digit numbers. If you were to multiply two 5-digit numbers on this computer, the exact result would have 10 digits. This product would need to be rounded to 5 digits in order to fit into the computers storage scheme. Modern computers are designed so that temporary calculations are performed in a slightly higher precision. This guarantees that the round-off error in a single add, subtract, multiply or divide is at most 1 bit in the last place.

The magnitude of storage and a single round-off error depends on the specific computer architecture that is being used to store the numbers. Most computers obey what is called the IEEE 754 specification for floating point numbers. In this system, each double precision floating point value is stored using 54 binary digits, which corresponds to roughly 15-16 decimal digits. Single precision numbers are stored using 23 binary digits, which is roughly 7-8 decimal digits.

In IEEE double precision, a storage error or a rounding error of 1 binary digit in the last place has a relative magnitude of

$$2^{-52} \approx 2.2204 \cdot 10^{-16}.$$

This number is also a built in varible named `eps` in MATLAB.

```
>> eps
ans =
   2.2204e-16
```

It is the smallest positive number that can be added to 1 and obtain a value different from 1.

```
>> a = 1
a =
     1
>> a+eps == 1
ans =
   0
>> format hex
>> a = 1
```

```
a =
   3ff0000000000000
>> a+eps
ans =
   3ff0000000000001
```

As the above sequence shows, 1 + `eps` is 1 binary digit greater than 1.

1.1 Absolute and Relative Errors

We often need to compare a computed value with some known value or determine how close two values are to each other. This is commonly referred to as calculation error. There are two types of calculation error that can be computed

- Absolute error - The absolute error is defined as

  $$\text{Absolute error} = \text{Approximate value} - \text{Exact value}.$$

  Note that there is no use of absolute value in the formula. This is because the sign of this error tells you something important. If the absolute error is negative (positive), then the approxmiate value is an underestimate (overestimate).

- Relative error - Most of the time when we assess computing errors, we want the relative error. This is defined as

  $$\text{Relative error} = \frac{\text{Approximate value} - \text{Exact value}}{\text{Exact value}}.$$

  As with absolute error, the sign of the relative error indicates whether the approximate value is an underestimate or an overestimate.

1.2 Digits of Accuracy

A common way to express the desired accuracy in a computation is to say 'compute the answer corret to 3 digits.' Relative error gives us a way to turn this expression into a concrete statement. By convention, two numbers a and b have q digits in common if their relative error (or more precisely, their relative difference) satisfies

$$|\text{Relative error}| \leq 0.5 \cdot 10^{-q}.$$

For example, suppose $a = 1.4531$ and $b = 1.4529$. Then the relative error between a and b is (take a to be the exact value) is

$$|\text{Relative error}| = \left|\frac{1.4529 - 1.4531}{1.4531}\right| = 0.137 \cdot 10^{-3}$$

so a and b agree to 3 digits.

2 Generating Equally Spaced Points

This is a common calculation that needs to be performed. To this point we have used the `linspace` command to do this, but it is instructive to see what this calculation means and how MATLAB performs it. It also presents an opportunity to illustrate the ideas in the previous section.

Suppose you want to divide the interval [0,1] into $n = 4$ equal segments. A quick sketch would reveal that each segment would be $h = \dfrac{1-0}{4}$ long. In addition, you can see that there would be a total of 5 points with coordinates (0.00, 0.25, 0.50, 0.75, 1.00).

The problem can be stated in a general way as: Given some interval $[a, b]$ of the x-axis, subdivide the interval into n segments of equal length. This is illustrated in the figure below.

The formulas for generating this set of points are

$$\begin{aligned} h &= \frac{b-a}{n} \\ x_i &= a + (i-1)h, \quad i = 1, 2, \ldots, n+1 \end{aligned}$$

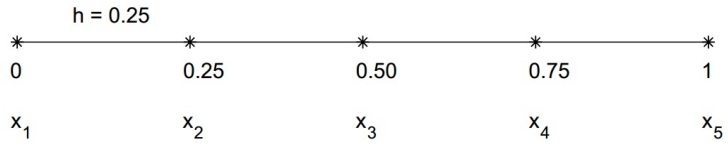

Figure 1: Subdividing the interval $[0, 1]$ into 4 segments.

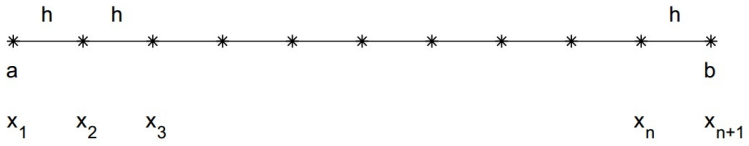

Figure 2: Subdividing the interval $a, b]$ of the x-axis into n equally spaced segments of length h (or $n + 1$ points spaced h units apart).

2.1 Method 1

The first method is given by

```
h = (b-a)/n
x(1) = a
for i = 2:n+1
  x(i) = x(i-1) + h
end
```

Expanding the first few steps of this process gives

```
h = (b-a)/n
x(1) = a
x(2) = x(1) + h      (= a +  h)
x(3) = x(2) + h      (= a + 2h)
x(4) = x(3) + h      (= a + 3h)
...
...
```

In this method, you can think of positioning yourself at point a on the x-axis. To get to the next coordinate, you take one step that is h units long. You keep stepping by h each time until you reach point b.

2.2 Method 2

The second method is given by

```
h = (b-a)/n
for i = 1:n+1
  x(i) = a + (i-1)*h
end
```

This method is somewhat different. Everytime you need to generate a new coordinate, you go back to the point a and compute how far you need to jump to get to the point. This distance is $(i - 1)h$. Expanding the first few steps of this process gives

```
h = (b-a)/n
x(1) = a
x(2) = a +  h
```

```
x(3) = a + 2h
x(4) = a + 3h
...
...
```

2.3 Difference in Methods

As you saw in the assignment, there is a huge difference in the relative error in the value of `x(n+1)` between these two methods when `n` is large. Why does this happen? The answer is round-off error.

In Method 1, the value of h contains several (small) errors. These come from storage errors in the values of a and b, and round-off errors in the calcluation of $b - a$ and the division by n. Since `x(2) = a + h`, `x(2)` can contain one additional round off-error. Because this pattern continues, `x(3)` contains one more round off error, `x(4)` contains one more round off error, *etc.*. By the time the final `x(n+1)` is computed, it can contain as many as `n+4` accumulated errors. This can be significant if `n` is large.

Method 2 begins in a similar way. h can possibly contain 4 errors. However, in this case, each new coordinate is computed by adding `(i-1)*h`, which adds 2 more errors. As a result, each `x(i)` contains at most 6 errors, regardless of the size of n. Thus, Method 2 is going to be more accurate. This is the approach that the `linspace` command uses.