Errors and Accuracy

Leigh J. Little SUNY Brockport October 18, 2020

Contents

1 Absolute Error, Relative Error, Relative Difference

1.1 Absolute and Relative Errors for Scalars

We often need to compare a computed scalar value (a) with a known, exact value (a_{exact}). The difference between these two values is called the error. There are two types of error that can be computed

- Absolute error - The absolute error is defined as

$$\text{Absolute error} = a - a_{\text{exact}}.$$

  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 a is an underestimate (overestimate) of the exact value.

- Relative error - The relative error is defined as

$$\text{Relative error} = \frac{a - a_{\text{exact}}}{a_{\text{exact}}}.$$

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

In most cases, we want the relative error. This is because absolute errors can be misleading. Suppose someone told you they did an experiment and the absolute error was about 10^{10}. Your first reaction would be to say that this error is terrible. However, if the values were $a = 1.3425 \cdot 10^{12}$ and $a_{\text{exact}} = 1.3466 \cdot 10^{12}$, the relative error between these values would be

$$(1.3425 \cdot 10^{12} - 1.3466 \cdot 10^{12})/1.3466 \cdot 10^{12} = -0.003.$$

The relative error is quite good.

Similarly, if you were told that the absolute error in an experiment was 10^{-6}, you would be inclined to say that this is a good error, however if the values were $a = 6.388 \cdot 10^{-9}$ and $a_{\text{exact}} = 9.332 \cdot 10^{-6}$, the relative error would be

$$(6.388 \cdot 10^{-9} - 9.332 \cdot 10^{-6})/1.3466 \cdot 10^{-6} = -0.9993.$$

This is nearly a 100% error and is quite poor.

Relative error removes the magnitude effect of the numbers from the calculation. In the event that $a_{\text{exact}} = 0$, relative error reduces to absolute error.

1.2 Absolute and Relative Errors for Vectors/Matrices

In the event we need to determine how close a vector (x) is to some known, exact vector (x_{exact}) the formulas above can be generalized by using the norm.

- Absolute error = $\|x - x_{\text{exact}}\|$.
- Relative error = $\dfrac{\|x - x_{\text{exact}}\|}{\|x_{\text{exact}}\|}$.

A slightly different form of relative error can be obtained by first computing the relative error vector

```
relerr = (x-xexact)./(xexact);
```

then computing

- Relative error = $\|\text{relerr}\|$.

1.3 Relative Difference

Relative difference is similar to, but not quite the same as relative error. Relative difference determines how close two numbers are to each other. It is most frequently used in situations where you have two values that are approximating the same exact value, but the exact value itself is unknown. The relative difference between two scalars a and b is defined as

$$\text{Relative Difference} = \frac{a-b}{\frac{1}{2}(a+b)}.$$

For two vectors x and y, the relative difference is

$$\text{Relative Difference} = \frac{||x-y||}{\frac{1}{2}||x+y||}.$$

1.4 Digits of Accuracy

A common way to express the desired accuracy in a computation is to say 'compute the answer correct 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 their relative difference) satisfies

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

For example, suppose $a = 1.4531$ and $b = 1.4529$. Then the relative difference between a and b is is

$$|\text{Relative difference}| = \left| \frac{1.4529 - 1.4531}{\frac{1}{2}(1.4529 + 1.4531)} \right| = 0.138 \cdot 10^{-3}$$

so a and b agree to 3 digits.

2 Testing for Equality

We have seen in a previous homework assignment that testing for equality between two numbers that have digits after the decimal point is unreliable and almost always leads to a false conclusion in situations where the numbers would be equal in exact arithmetic. Relative error and relative difference provides a reliable mechanism for performing this test.

Rather than test for exact equality, the quantities should be tested to determine if their relative error or difference is less than some prescribed tolerance. Instead of doing

```
if(a == b)
   ....
end
```

The test should be restated as

```
if(abs(a-b)/((a+b)/2)) < TOL)
   ...
end
```

where `TOL` is some tolerance. The tolerance for single precision accuracy is typically 10^{-6} while double precision accuracy is typically 10^{-12}.

3 Entering Values Using Scientific Notation

Some of the values that have been output from the codes we have written have been displayed in scientific notation, but to this point we have not had to enter numbers in scientific notation. This is easy to do. Suppose you needed to use Avagadro's Number $N = 6.022 \cdot 10^{23}$. In MATLAB, you would enter this as

```
>> 6.022e23
ans =
   6.0220e+23
```

Similarly, if you needed the charge of an electron ($1.602 \cdot 10^{-19}$ C) you would do

```
>> 1.602e-19
ans =
   1.6020e-19
```

A common mistake is to accidentally add in an additional factor of 10. For example, if you needed to set TOL equal to 10^{-12} you would do

```
>> TOL =  1e-12
TOL =
   1.0000e-12
```

If instead you enter

```
>> TOL =  10e-12
TOL =
   1.0000e-11
```

your value will be off be a factor of 10.

4 The `break` Statement

Sometimes it is necessary to stop processing a loop due to some condition that arises in a calculation. The `break` statement is used to accomplish this. For example, consider the code section

```
i = 0;
while (i < 10)
    i = i  + 1
    if(i > 5)
       break
    end
end
disp('End of break test')
```

If you execute this code, the output would be

```
i = 1
i = 2
i = 3
i = 4
i = 5
End of break test
```

The `break` statement forces the loop to exit once the value of `i` exceeds 5.

In a situation where the `break` occurs in a series of nested loops, the `break` statement will terminate processing of the innermost loop that the break is in. For example,

```
i = 0;
while (i < 2)
    i = i  + 1
    j = 0;
    while (j < 4)
       j = j + 1
       if(j > 1)
          break
       end
    end
end
disp('End of break test')
```

The output from this code would be

```
 i = 1
 j = 1
 j = 2
 i = 2
 j = 1
 j = 2
 End of break test
```

The `break` statement causes the `j` loop to terminate once `j` exceeds 1, but the `i` loop continues to execute.

5 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).

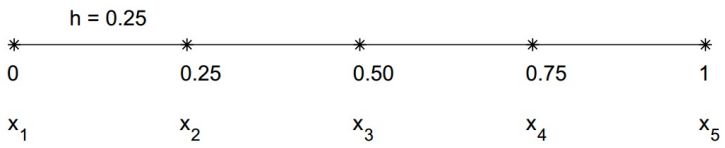

Figure 1: Subdividing the interval $[0, 1]$ into 4 segments.

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

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

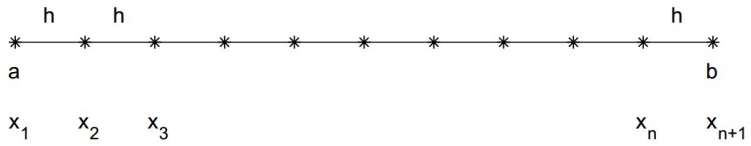

Figure 2: Subdividing the interval $a, b]$ of the x-axis into n equally spaced segments of length h (or $n+1$ points spaced h units apart).

5.1 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.

5.2 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
...
...
```

5.3 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.