Arithmetic on a computer

MATH1070: Numerical Mathematical Analysis

Location: http://people.sc.fsu.edu/~jburkardt/classes/math1070_2019/arithmetic/arithmetic.pdf

---

Enough tiny mistakes equal one big catastrophe.

Computer arithmetic

Computer arithmetic tries to simulate mathematics, but can break down:

- *there are limits on how big or small a number can be;*
- *there are numbers so small that adding them to 1 makes no difference;*
- *two formulas that are the same mathematically may give different results on the computer;*
- *adding $\frac{1}{3} + \frac{1}{3} + \frac{1}{3}$ doesn't get you 1;*
- *small errors can add up to a big problem;*

Most numerical computations are done with a version of IEEE arithmetic, which replaces the infinitely long and infinitely dense real number line with a finite collection of numbers of limited precision, and applies arithmetic operations whose results must be rounded back into this system.

While we rely on such computations, we need to remember that the numbers we start with and the results we obtain are approximations, that approximation means error, and that in some cases a calculation can be poisoned by the errors we start with.
In mathematics, we expect that, for any positive real number x,

- $x + 1$ is bigger than 1;
- $2 * x$ is bigger than x;
- $x/2$ is smaller than x;
- if x is not zero, then $\frac{x}{2}$ is not 0;
- $x + y - x$ equals y;
- $\frac{1}{x} \cdot x = 1$;

- between any two distinct numbers there is another distinct number;
- there is no smallest positive number;
- there is no largest positive number;

In computation, we probably also assume that

- legally rearranging a formula can't change its value;
- if your data has 8 correct digits so will your results;
- when approximating, we can always get more accuracy if we try harder;

1 Numbers on a computer

The following tiny MATLAB commands show how a few of these rules are broken when we work in computational arithmetic:

```
x = 1; while ( 1 + x > 1 ), x = x / 2, pause ( 0.05 ), end
x = 1; while ( x + x > x ), x = x / 2, pause ( 0.05 ), end
x = 1; while ( x + x > x ), x = x * 2, pause ( 0.05 ), end
```

Listing 1: rule_breakers.m

These results are evidence that:

1. some real numbers are so small you can't add them to 1;
2. some real numbers are so small they can't get smaller;
3. some real numbers are so big they can't get bigger.

MATLAB makes it easy to check some of the limits on numerical precision. In particular, `eps`, the smallest number that you can add to 1 and make a difference, comes up frequently in numerical work. If a number is at least as big as `eps`, you can add it to 1. If a number is at least as big as `eps*x`, you can add it to `x`. So, for example, `eps` gives you a way of checking whether the terms in a series have become negligible.

Consider the following formula:

$$\pi = \sqrt{6 * \sum_{k=1}^{\infty} \frac{1}{k^2}}$$

We could write a code that knows when to quit:

```
  s = 0;
  k = 0;
  while ( true )
    k = k + 1;
    t = 1 / k^2;
    if ( t < eps * s )
      break
    end
    s = s + t;
  end
  s = sqrt ( 6 * s );
  fprint ( 1, 'Estimate = %20.16g, absolute error = %g, next term was %g\n', s, abs ( s - pi
      ), t );
```

Listing 2: pi_series1.m

Notice that the estimate has only about 8 significant digits. That's because we computed an estimate of pi^2 to 16 digits, and took its square root. The square root reduces our accuracy by half.

In MATLAB, the largest and smallest positive real numbers have the names `realmax` and `realmin`. These are so extreme that they are rarely of importance. Because these numbers are at the very limits of the computational number system, though, many rules break down. For instance, you should know that, mathematically, e^x can never be zero, although it can get very small as x becomes very negative.

```
r = realmax;
y1 = exp ( -r )
y2 = r + 1
y3 = r + r
y4 = 2 * r
```

The values of $y3$ and $y4$ were returned as `Inf`, which is MATLAB's way of expressing infinity, which mathematically we write as ∞. MATLAB includes a value of infinity so that it can return a meaningful value to a numerical operation as often possible. When it encounters a fraction like $\frac{1}{0}$, it returns the reasonable, though non-numerical, value `Inf`. Similarly, evaluating `log(0)` would return a value of `-Inf`.

Another special value is `NaN`, for "Not-A-Number", which is the value returned when there really is no value that can be assigned to the requested computation. The classic example is evaluating `0/0` , but you can also get this by `sin(1/0` or `Inf-Inf`.

2 A Taylor series on a computer

In mathematics, very smooth functions like $\sin(x)$ and e^x can be approximated to any desired accuracy using a Taylor series. In particular, we have the following series for $\cos(x)$:

$$\begin{aligned}\cos(x) &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ... \\ &= \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}\end{aligned}$$

Since we can't actually compute an infinite sum, it might seem reasonable, when using the Taylor series, to continue to add terms until the next term is so small in magnitude that it doesn't actually change the current estimate. A code to do this would be:

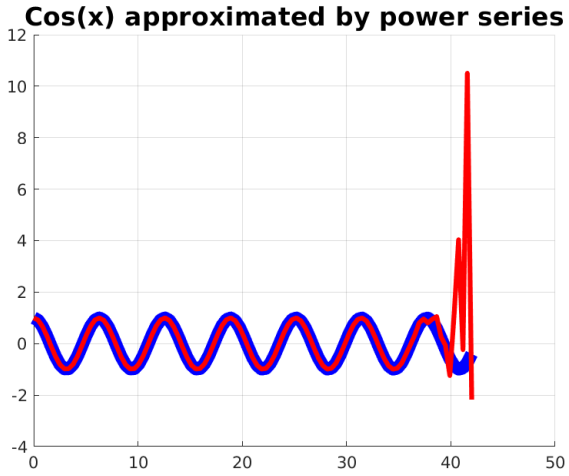
```
function c = cos_taylor ( x )

  n = 0;
  c = 0.0;
  while ( true )
    if ( n == 0 )
      t = 1.0;
    else
      t = - x.^2 / n / ( n - 1 ) .* t;
    end
    if ( ( c + t ) == c )
      break
    end
    c = c + t;
    n = n + 2;
  end

  return
end
```

Listing 3: cos_taylor.m

Notice that the terms we are summing alternate in sign. Since our result will be about the order of 1, if these terms are very large, then many digits of our computation will be cancelled out as we go along. And as x gets bigger, we must expect that the terms in the Taylor series do become very large for a while, before the factorial divisor gradually drives them towards zero. Here's what happens if we use the Taylor series too far:



The Taylor series (red) for $\cos(x)$ *(blue) fails beyond* $x = 40$.

Here is how the computation goes for $x = 1$:

n	c	t
0	0	1
2	1	-0.5
4	0.5	0.04166666666666666
6	0.5416666666666666	-0.001388888888888889
8	0.5402777777777777	2.48015873015873e-05
10	0.5403025793650793	-2.755731922398589e-07
12	0.5403023037918870	2.087675698786810e-09
14	0.5403023058795627	-1.147074559772972e-11
16	0.5403023058680920	4.779477332387385e-14
18	0.5403023058681398	-1.561920696858622e-16

The exact value is 0.540302305868140. But when we move to $x = 40$, the computation is no longer well behaved. The terms grow enormously for 50 steps, before gradually decreasing.

n	c	t
0	0	1
2	1	-800
4	-799	106666.6666666667
6	105867.6666666667	-5688888.888888889
8	-5583021.222222222	162539682.5396825
10	156956661.3174603	-2889594356.261023
...	...	...
50	2570694634194567	-4167971051515082
...	...	...

```
100          -150.0872900558725           172.1850646803727
...                 ...                          ...
134          -1.154002094667014       -2.37981948266972e-14
136          -1.154002094667038        2.073916760496488e-15
138          -1.154002094667036      -1.755139541306665e-16
```

The answer is obviously wrong; the correct value is about -0.6669. As x increases beyond 40, the estimate breaks down completely.

It's important to note that, mathematically, the formula is correct. If we were using exact arithmetic, then summing up to term #138 would have given us a very accurate answer. But for $x = 40$, our formula requires us to add and subtract numbers of very different magnitudes, and because we can only do this to a limited number of digits, our result fails.

3 Adding 3 million numbers accurately

Isaac Asimov noted that the fraction $\frac{355}{113}$ gives a pretty good approximation to π. He also knew of an infinite series:

$$\pi = 4 * \sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} \cdots$$

and he asked the question: how many terms would I have to add up in the series in order to get an answer as accurate as that simple fraction? For exact arithmetic, the answer is 3,748,630. We can try this challenge computationally, but can we expect a reasonably accurate result?

```
1   x1 = 355.0 / 113.0;
2   e1 = abs ( pi - x1 );
3   %
4   %  Sum terms until error is less than e.
5   %
6   k = 0;
7   s = 1.0;
8   x2 = 0.0;
9   while ( true )
10    t = s * 4.0 / ( 2 * k + 1 );
11    x2 = x2 + t;
12    e2 = abs ( pi - x2 );
13    s = - s;
14    k = k + 1;
15    if ( e2 <= e1 )
16      break;
17    end
18  end
```

Listing 4: pi_series3.m

Somewhat surprisingly, our computation stops on the same term as the exact arithmetic calculation. Can you see why we do not see to suffer a huge loss of significant digits even though we are adding and subtracting so many terms?

4 Three ways to evaluate a polynomial

Mathematically, we can rearrange the terms in a formula in many legal ways, without altering the result. Computationally, this is not always true. We have already seen that adding and subtracting values of comparable size can reduce the accuracy of our result. Let's look at what this means in terms of a very simple polynomial $p(x)$.

We can write this polynomial three ways:

- $p_1(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$ (standard form)
- $p_2(x) = ((((((x - 7) * x + 21) * x - 35) * x + 35) * x - 21) * x + 7) * x - 1$ (Horner's form)
- $p_3(x) = (x - 1)^7$ (factored form)

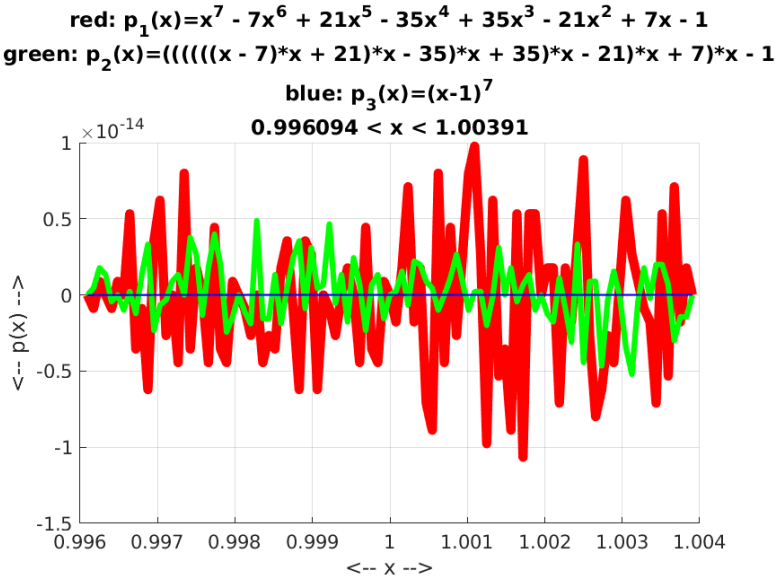
These forms are equivalent mathematically, but not computationally. And especially near $x = 1$, we can see that there are noticeable differences in the results. If we were looking to solve $p(x) = 0$, it looks like we could find many (wrong) answers instead of the single correct result.

```
xmin = -1.0;
xmax = +3.0;

xstep = ( xmax - xmin ) / 100;
x = xmin : xstep : xmax;
y1 = x.^7 - 7*x.^6 + 21*x.^5 - 35*x.^4 + 35*x.^3 - 21*x.^2 + 7*x - 1;
y2 = ((((((x - 7).*x + 21).*x - 35).*x + 35).*x - 21).*x + 7).*x - 1;
y3 = ( x - 1 ).^7;
hold ( 'on' );
plot ( x, y1, 'r-', 'linewidth', 5 )
plot ( x, y2, 'g-', 'linewidth', 3 )
plot ( x, y3, 'b-', 'linewidth', 1 )
hold ( 'off' );
```

Listing 5: plot_poly.m

When we reduce the range to $[0.996, 1.004]$, we see a big discrepancy in the results of the three formulas:



The red, green and blue formulas are mathematically the same.

Especially when we look at the standard form of the polyomial, it is clear that we are adding and subtracting terms of roughly equal magnitude and losing our precision.

5 Trying to harder to approximate can make things worse

In calculus, we learn that if the derivative $f'(x)$ exists, it can be thought of as a limit:

$$f'(x) = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$

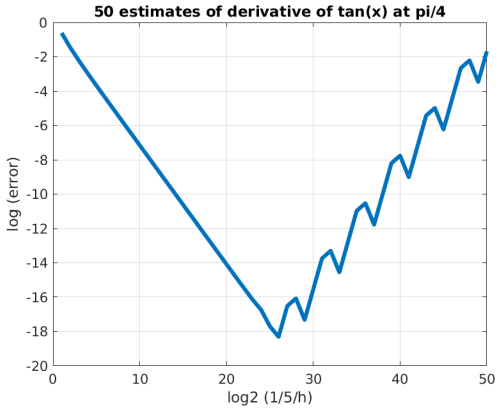
Let's approximate the derivative of $\tan(x)$ at $x = \frac{\pi}{4}$, whose exact value is 2. If we repeatedly divide h by 2, we expect to get ever better answers.

```
h = 0.2;
x = pi / 4;
fp = 2.0;

for i = 1 : 50
  fd = ( tan(x+h) - tan(x) ) / h;
  fprintf ( '  %2d  %14.8g  %14.8g  %g\n', i, h, fd, abs ( fd - fp ));
  h = h / 2.0;
end
```

Listing 6: tan_difference.m

But our error plot reveals that our results become junk as h gets too small:



Trying too hard to estimate f'(x), our answers get worse.

Our estimate is based on the difference $f(x + h) - f(x)$. As h gets very small, these two numbers get very close, and so we begin to lose many significant digits in the result.

6 Missile Defense

In February 1991, an American military base was expecting a Scud missile attack. It was protected by the Patriot missile system. The operators had been warned that the system was not designed to run continuously for "excessively long times", whatever that meant. The system had been running for about 100 hours when the attack occurred, at which time, the Patriot did not correctly intercept the attacking missile, which exploded on target.

Subsequent investigation focussed on the Patriot's timing system. Once the system was started, there was a "tick" every tenth of a second, stored as an integer `T`. The intercept program needed the time in seconds, so this was stored in a real number `S`. Instead of dividing by 10, the value `T` was multiplied by 0.099999905, which is ... "pretty close. However, as time passed, these tiny errors increased. At 100 hours, the system's time was wrong by about $\frac{1}{3}$ of a second, which was not noticeable to the operator. But a Scud travels at 1,676 meters per second, and this meant that at the time of attack, the Patriot's aim was off by more than half a kilometer.

```
one_tenth = 0.099999905;
v = 1676.0;

fprintf ( '      h              t          r              s          r-s        v*(r-s)\n' );
```

```
5
6  for h = [ 0, 1, 8, 24, 48, 72, 96 ]
7    t = h * 60 * 60 * 10;  % T is the number of ticks counted in H hours;
8    r = t / 10.0;          % R = T / 10 (correct time in seconds)
9    s = t * one_tenth;     % S = T * "1/10" (slightly wrong)
10   ds = r - s;            % DS = error in time!
11   dx = v * ds;           % DX is the error in the estimated missile position
12   fprintf ( '    %3d   %12d   %12.4f   %12.4f   %12.4f   %12.1f\n', h, t, r, s, ds, dx );
13 end
```

Listing 7: patriot.m

Moral: This problem could have been avoided by dividing T by 10, rather than multiplying by an approximate value of 1/10. And that "insignificant error" would never have been noticed except that the system stayed on so long that the error was allowed to grow. The high speed of the Scud missile multiplied the "moderate error" so that it became a "catastrophic error".