CS-2363: Assignment 1

Santripta Sharma

September 8, 2024

Tutorial 1

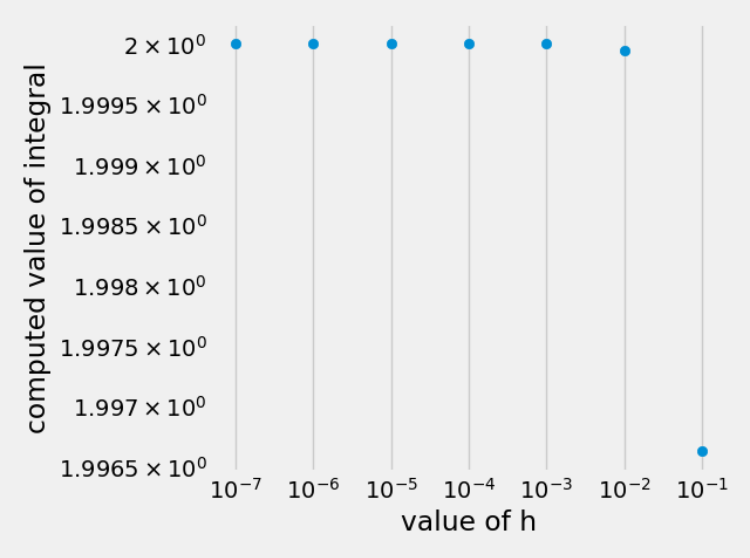
1.1

Analytically, this integral resolves to $[-\cos x]_0^\pi = [\cos x]_\pi^0 = 1 - (-1) = 2$.

Upon implementation, the following results are observed:

h	computed	iterations
0.1	1.9966292525412175	32
0.01	1.9999479920714234	315
0.001	1.999999750367962	3142
1e-04	1.9999999983052663	31416
1e-05	1.9999999999476539	314160
1e-06	1.99999999998949	3141593
1e-07	2.000000000570725	31415927

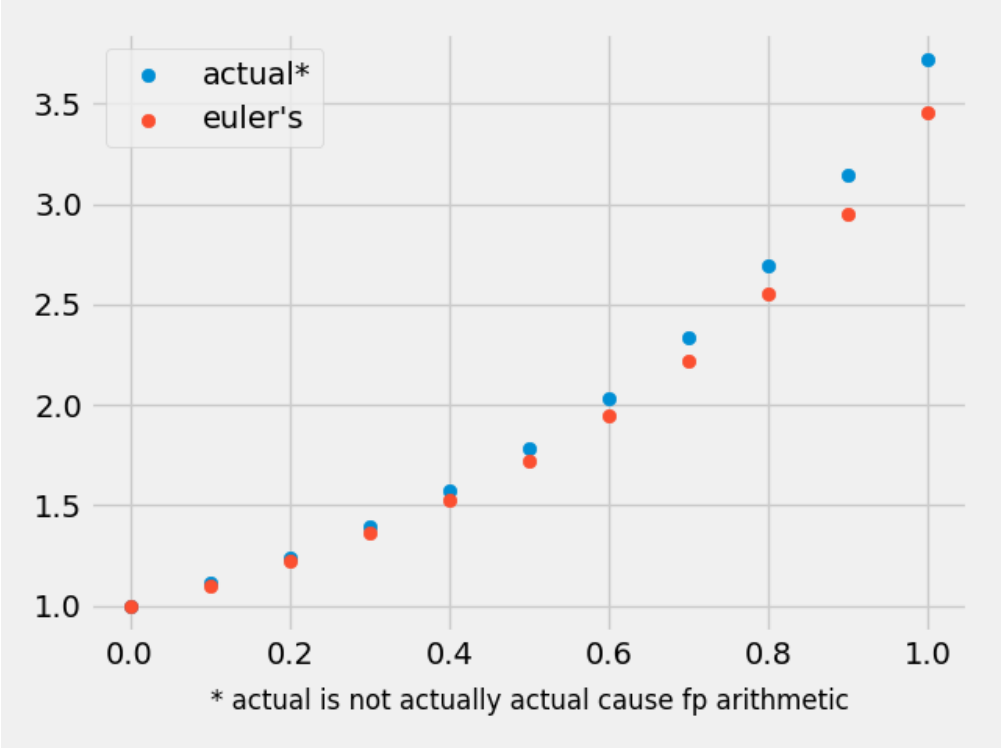
As expected, the number of iterations approaches $\pi \times 10^i$ as h approaches 10^{-i}. Since each tenfold increase in the number of iterations leads to a two more digits of precision (taking 1.9999... to be the target), we have linear convergence (error decreases 100-fold for each 10-fold increase in iterations).

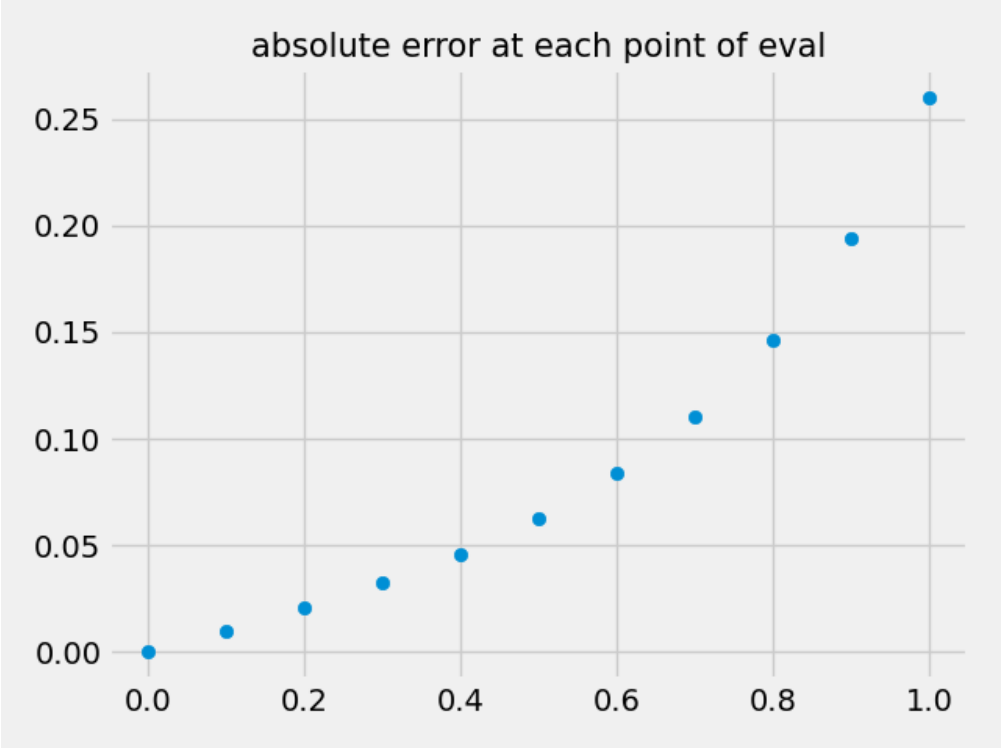


1.2

Performing the iteration with the given x_ns is equivalent to getting the solutions in the interval $[0, 1]$ with a step of $h = 0.1$.

This gives us the following results:

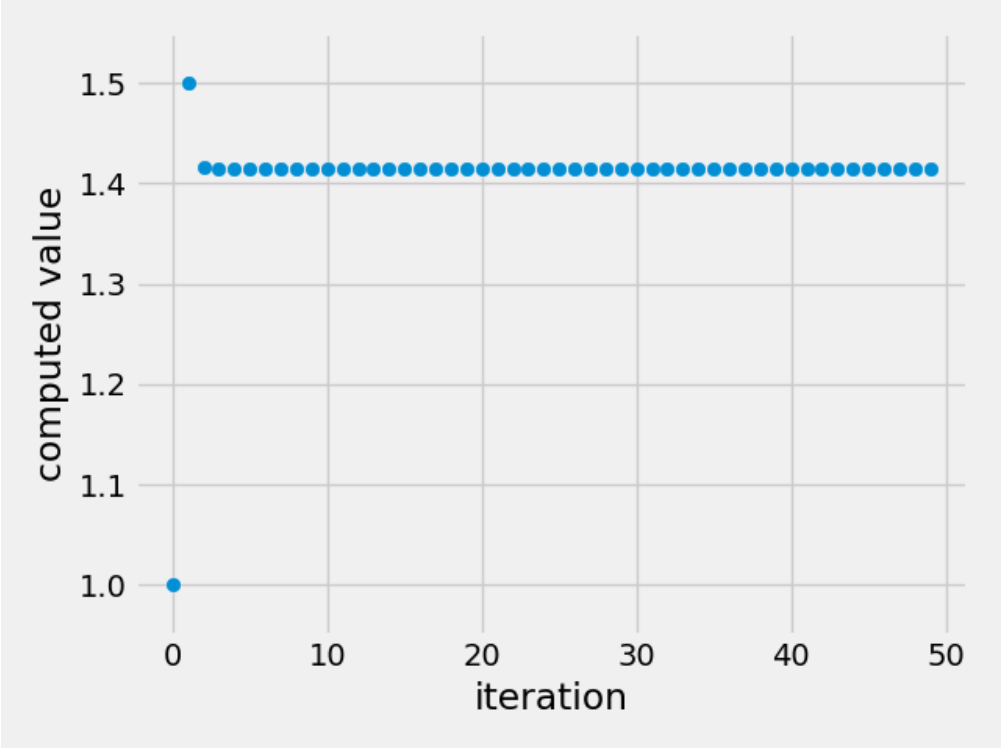




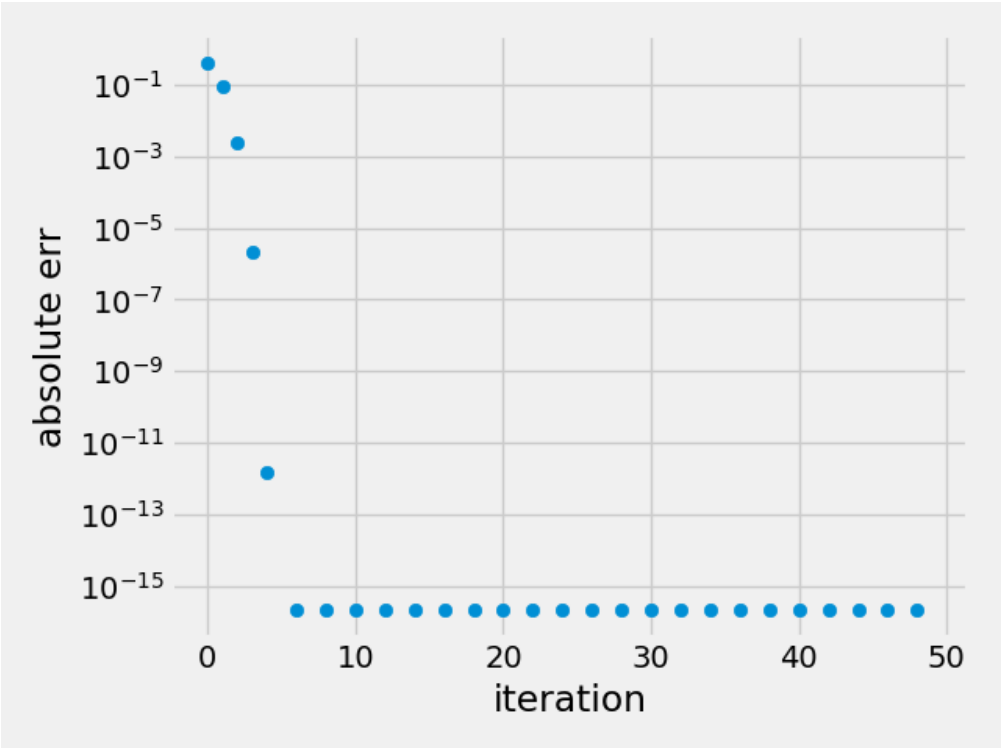
As we can see (and perhaps anticipate), the errors begin mounting as x increases. This is because this method essentially assumes that the given function is locally linear at any given x_i. This would be true for an infinitesimal divided difference approximation (i.e $h \to 0$), but is not true for the given $h = 0.1$.

1.3

We can see the results of the iteration here:



and the abs error (against the best approximation of $\sqrt{2}$ available on my computer):



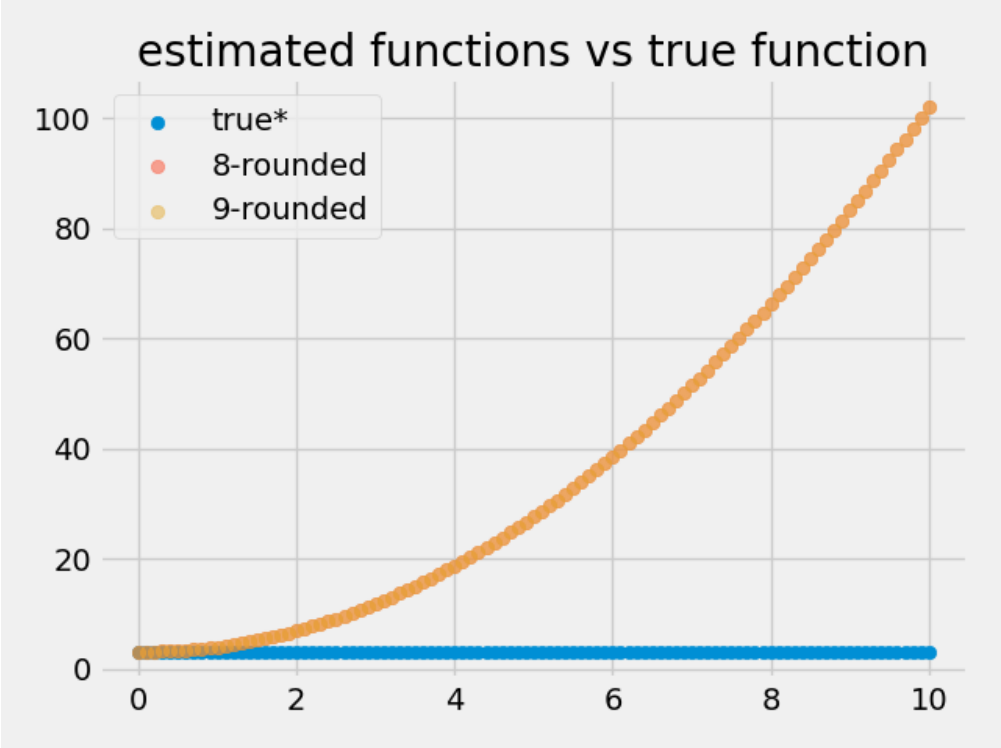
This figure confirms, and gives us a better intuition for, the quadratic convergence rate. We see that the error term is squared at each iteration (due to the derivative of the FPI being 0 at the fixed point, $\sqrt{2}$).

To formalise this, define the order of an error term e_i as the smallest power of 10, 10^o, such that $e_i < 10^{o+1}$.

If the initial error was of the order of 10^{-1} (which is the case for $x_0 = 1, e_0 = 0.41\ldots$, since they differ in all digits 10^{-1} onwards), the error after t iterations, e_t, is of the order 10^{-2^t}. If d is the desired digits of precision, making the replacement $t = \log d \Rightarrow e_t$ is of the order $10^{-2^{\log d}} = 10^{-d}$. Therefore, after $\log d$ iterations, the error is bounded above by 10^{-d+1}, which gives us (roughly) the desired precision.

2.1

Solving the differential equation using euler's method, setting y_0 in all cases to be the best available approximation to π, with $x_0 = 0, x_n = 10, h = 0.1, \pi_8 = 3.14159265, \pi_9 = 3.141592654$, we observe the following results:

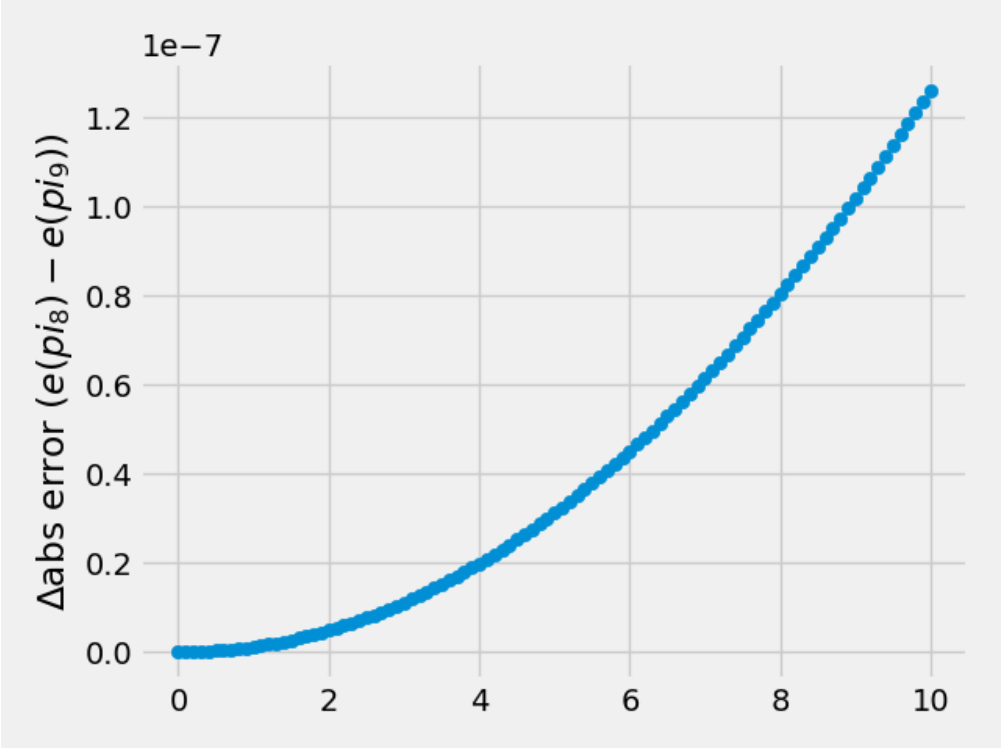


We see that while both the y_8 and y_9 approximations are close to each other, they do not estimate the true y value at any of the given points too well. The explanation here is quite simple. In the actual function, $\pi \cdot y_0/[y_0 + (\pi - y_0)e^{x^2}]$, when $y_0 := \pi$, we get:

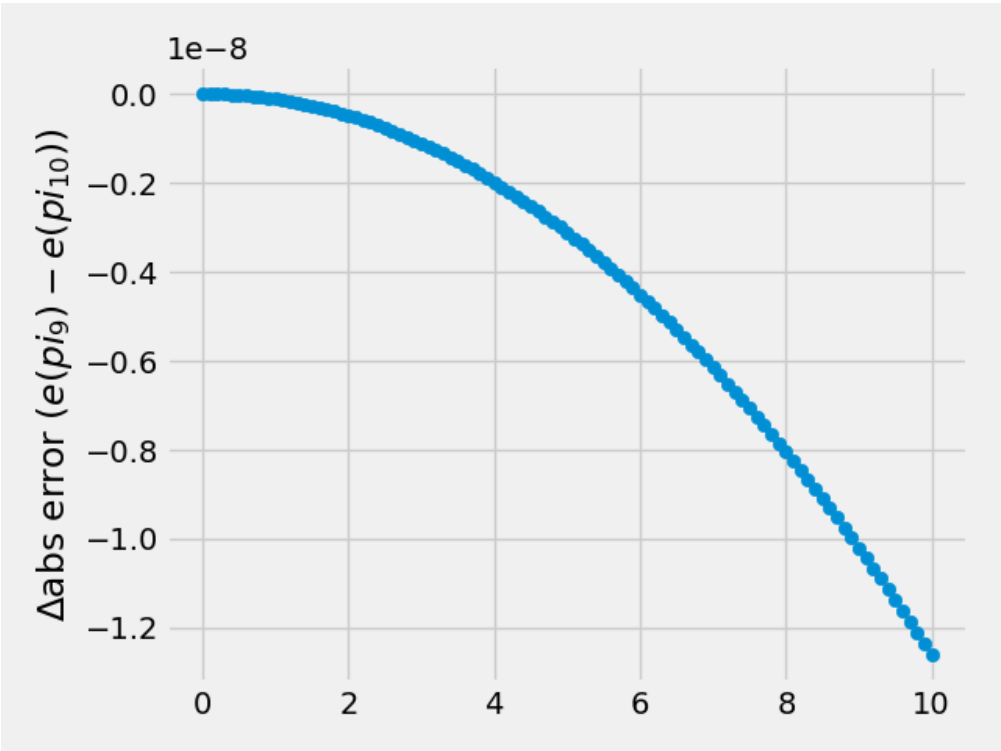
$$= \frac{\pi^2}{\pi + 0 \cdot e^{x^2}}$$
$$= \pi$$

Giving us the constant, true (where both π and y_0 are the best computer approximations to π), function y. However, when we use the rounded estimates of π in $y'_8 = (2/\pi_8) \cdot x \cdot y_8(y_0 - \pi)$, both the $\exp(\ldots)$ term in the denominator (within y_8) and the $x \leftarrow y_0 - \pi$ term in the argument to y_8 do not vanish, giving us the faulty approximation.

We can now plot the difference in the errors as a function of x, using $|y_8 - y| - |y_9 - y| = e(\pi_8) - e(\pi_9)$ as the error term:



Besides informing us that both approximations yield similar functions (Δerr is of the order of $1e-7$), this also tells us that the error is always higher for the less precise (8-digit) approximation of π than the more precise (9-digit) approximation, which aligns well with our intuition of what should happen. To put this intuition to the test, we can also repeat this comparison for π_9 and π_{10}, a 10-digit rounded approximation to π. With this, we see the following error difference:



...which now seems counterintuitive. One possible reason for this is that the gains in accuracy from accuracy from one more digit of precision here are counteracted by the increased

accumulation of error during the many $(y_0 - \pi)$ subtractions in the process. This subtraction is problematic because of our choice of y_0 as the best available approximation to π. Then, when π is replaced by π_9 or $\pi_1 0$, we have a subtraction which ends up dropping most of our significant digits.

Since the loss in precision is roughly positively correlated with how close y_0 is to our approximation of π, $\pi_1 0$ accumulates more error due to this issue, which **could** be the cause of the observed inversion of ordering.

2.2

Solving the first system using the typical LU decomposition method, we get the solution $x_1 = (1400.06794774, 699.78397387)$. After the slight perturbation, we see that the solution changes quite drastically to $x_2 = (1749.91822625, 874.70911312)$.

Clearly, this problem is poorly conditioned/unstable, since we see a larger relative change in output (consider taxicab norm between solution vectors) for a tiny relative change in the input system matrix (consider sum of element-wise absolute differences as the distance metric for the matrices).

Putting numbers to it, we have: $d(f(x'), f(x))/||f(x)|| = (|1400.06794774 - 1749.91822625| + |699.78397387 - 874.70911312|)/(1749.91822625 + 874.70911312) = 524.77541776/2624.62733937 \approx 0.1999$ as the relative error in the output and $d(x', x)/||x|| = 10e-3/(2+4+2.998+6.001) \approx 1/15000$ as the relative error in the input. Then, for this input output pair, the ratio of relative errors in the input to the output is $0.1999 \times 15000 \approx 3000$. This shows that at minimum, the condition number of this problem is 3000 (since there might be input-output pairs with higher relative error ratios). Therefore, we have shown that this is an ill-conditioned problem.

Another explanation for this is that the original matrix for this system is nearly dropping rank ($detA \approx 0.01$), as can be seen by multiplying the first equation by -1.5:

$$\begin{aligned} -3x &+ 6y &&= 1.5 \\ -2.998x &+ 6.001y &&= 2 \end{aligned}$$

2.3

Solving the system after changing 201 to 200, we see that two of the three solutions become imaginary ($0.9949 \pm 0.10037i$), which means there is no coherent distance metric between 1 and them, so 1 is not "even close" to them, by vacuity.

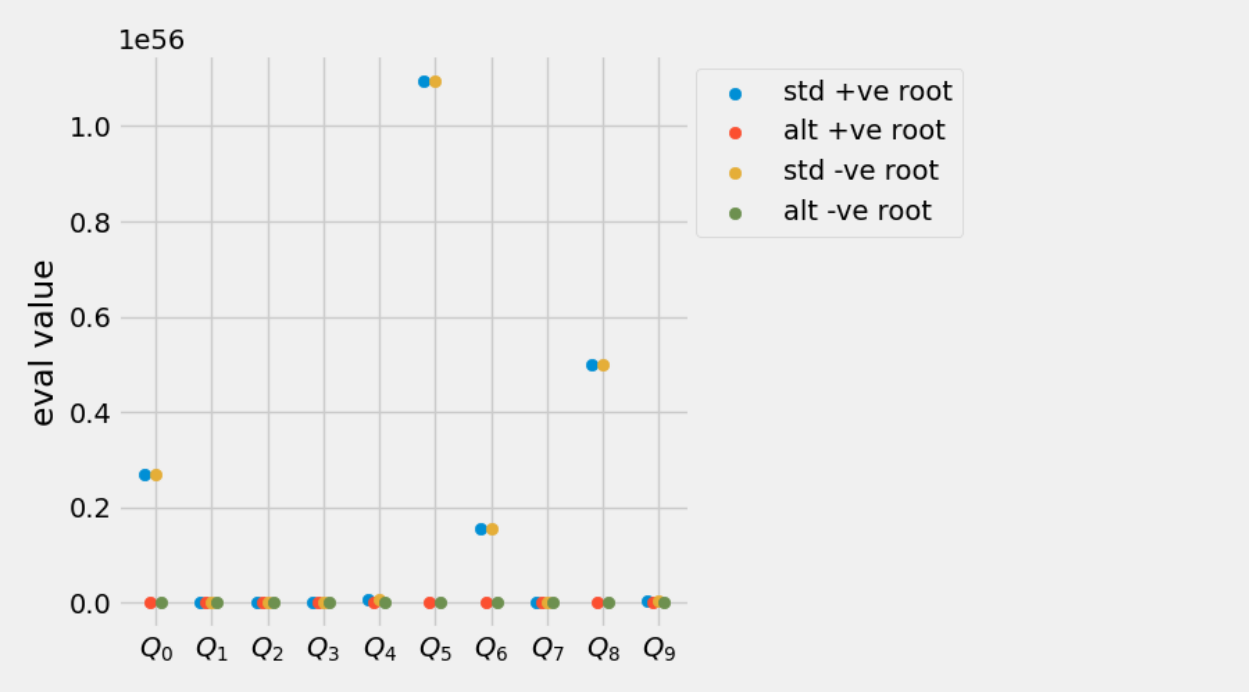
The third solution is real, but is 100.0102, which is also not really close to 1. Then, a slight change in the input (coefficients) lead to a major change in the output (solution), so this problem is ill-conditioned.

3.2

We begin by generating 10 (a, b, c) pairs, each using a uniform sample from the ranges $([10^3, 10^4], [10^{12}, 10^{15}], [10^3, 10^5])$, giving us $Q_1, \ldots, Q_{10}$:

Q_0	5890.6444761186885	891430632357951.8	43738.71418264905
Q_1	3505.3244658441654	209992919995072.4	94062.95214261509
Q_2	4820.658316742198	186142891330524.97	81947.28849889601
Q_3	8602.985190879133	109268513573790.88	34275.083061968966
Q_4	1042.4697057187532	220477795132367.16	18365.634920491328
Q_5	2094.122087048028	978645160922662.2	37910.372582702395
Q_6	7036.741762541008	811871465940234.0	1563.162227904769
Q_7	8432.67479594543	172769071719861.62	25990.208991039202
Q_8	2230.3593071645764	816408523977114.1	79770.58833885545
Q_9	6175.83996484525	274799673294657.47	2510.242153387562

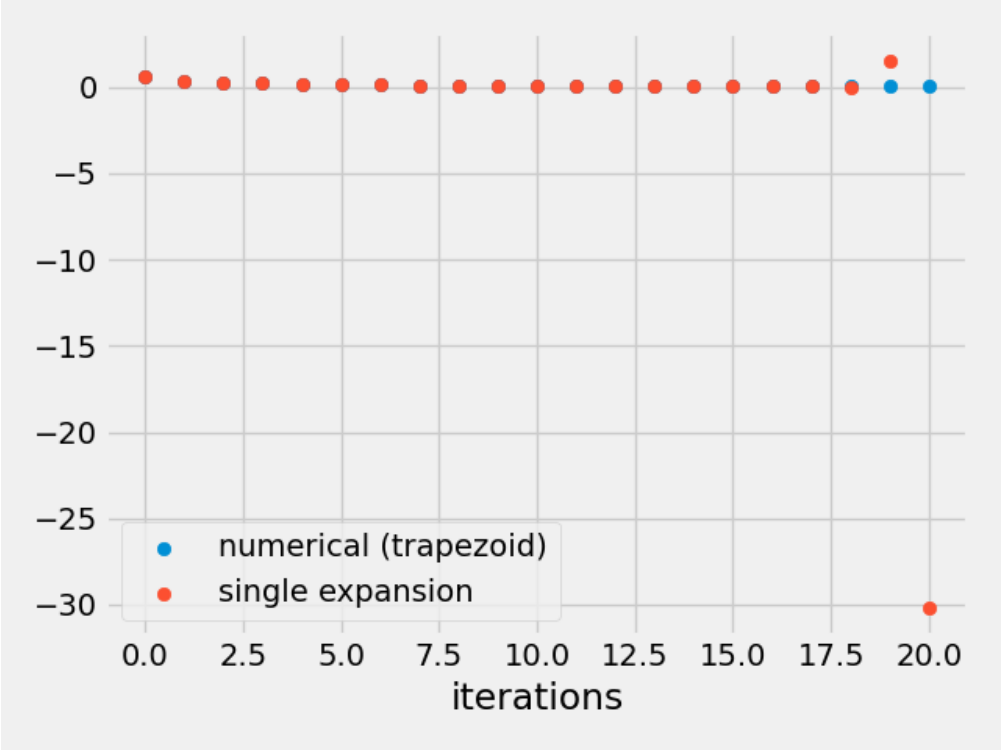
We can then compute their roots using both methods, substitute them into the equations, and check how far they are from the expected value of 0:



We can see how in the cases that the standard root solver fails (evaluation $\neq 0$), the alternate method still holds stable. This is because the standard method has potential to incur catastrophic cancellation due to the $-b$ term in the quadratic formula, which can lead to a loss of precision. The alternate method, however, does not have this term, and is therefore more stable.

3.3

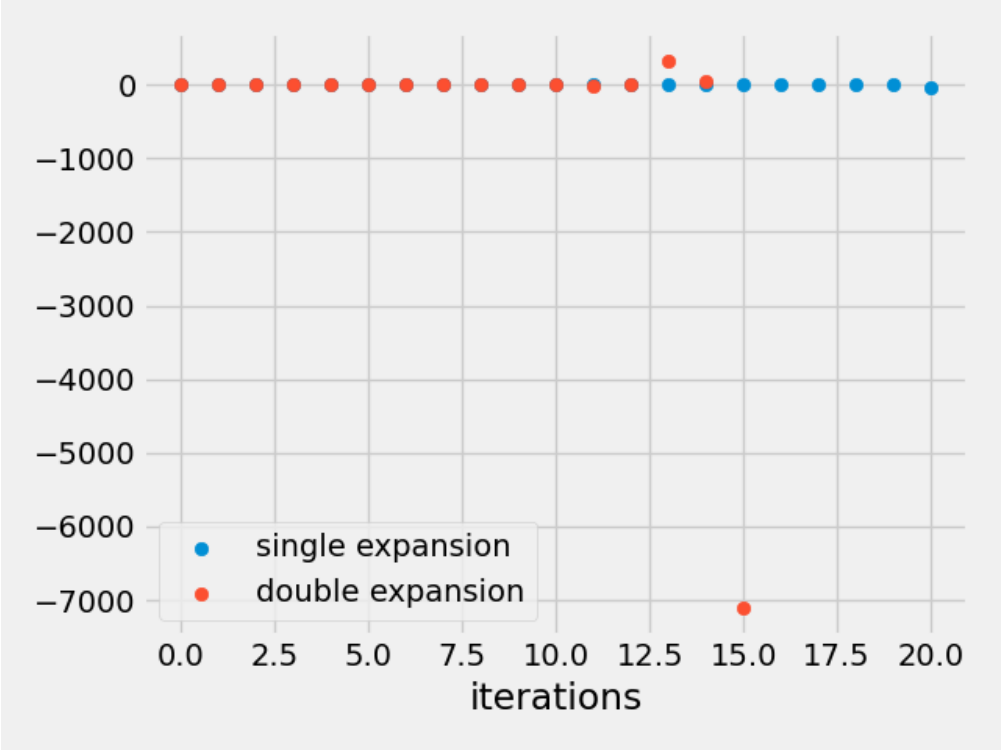
Here is the plot of iteration i vs I_i:



We can see that, although some noise exists before this point, there is a sudden deviation at $i = 19$. To investigate why this happens, we step back and find that for all iterations $i < 18$, the term $X_i = i * I_{i-1} < 0.95$. However, at $i = 18$, this term becomes 1.0294536707515363. Note that this should never happen, since this is a completely positive integral, but it likely happens due to the error accumulation over the previous iterations.

Once it happens, $I_{18} := 1 - X_{18} = 1 - 1.0294536707515363$, which, due to being a subtraction of two nearby values, loses precision (catastrophic cancellation), netting the value of $I_{18} = -0.029453670751536265$ which appends some junk digits to the number. This causes problems further down the chain, as X_{19}, also negative gets reflected about 1, resulting in the I_{19} becoming greater than 1, setting off an immediate explosion due to the X term in future iterations, leading to the kind of error we see.

We see a similar pattern in the case with the alternate iteration, as shown here:



Here, we've excluded the last 5 terms, as the error there blows up towards a scale of $10e6$, where it becomes hard to see the trend. However, we see a similar pattern, although it occurs much earlier. We then claim that the first form is more numerically stable. This is possibly because the alternate form has a k^2 term in the product with I_{i-2}, which means that once things start blowing up, they blow up much faster.

3.4

Consider the set $X = \{1e43 + i, 0 \leq i \leq 10\}$. Using the alternative formula here returns the variance as $-2.7606985387162255e + 70$, which is negative, therefore clearly incorrect. The actual value, which is correctly returned by the first method is $1.532495540865889e + 54$.

The reason this happens is because we have a set of large numbers which are very bunched up. So, the $\sum x_i^2$ term can essentially be approximated by $n \cdot x_1^2$. Similarly, in the mean, the $\sum x_i$ term can be replaced by $n \cdot x_1$, therefore, $\overline{x}^2 = \left(\frac{n \cdot x_i}{n}\right)^2 = x_1^2$. Similarly, the first term, when divided by n, yields x_1^2 as well.

So, we are performing a subtraction of two numbers a, b, both of which are very close to x_1^2, which is very big. This leads to a catastrophic cancellation, a large drop in precision, leading to our erroneous answer.

Tutorial 2

1.1

1. $765.4567 = 7.654567 \times 10^2 = 7.654 \times 10^2$
2. $765.4567 = 7.654567 \times 10^2 = 7.655 \times 10^2$
3. $765.499 = 7.65499 \times 10^2 = 7.655 \times 10^2$
4. $100.05 = 1.0005 \times 10^2 = 1.001 \times 10^2$

1.2

To perform any addition, we first need to align the numbers by shifting the number with the smaller exponent to align its exponent with the larger number.
Forward Pass

1. $x_1 + x_2$:
   - **Shift** $0.3429 \times 10^0 = 0.03429 \times 10^1$
   - **Chop** to 0.0342×10^1
   - **Add** $10^1 \times (0.1234 + 0.0342) = 0.1576 \times 10^1$
2. $(x_1 + x_2) + x_3$:
   - **Shift** $0.1289 \times 10^{-1} = 0.001289 \times 10^1$
   - **Chop** to 0.0012×10^1
   - **Add** $10^1 \times (0.1576 + 0.0012) = 0.1588 \times 10^1$
3. $(x_1 + x_2 + x_3) + x_4$:
   - **Shift** $0.9895 \times 10^{-3} = 0.00009895 \times 10^1$
   - **Chop** to 0.0000×10^1
4. $(x_1 + x_2 + x_3 + x_4) + x_5$:
   - **Shift** $0.9763 \times 10^{-5} = 0.0000009895 \times 10^1$
   - **Chop** to 0.0000×10^1

Therefore, we get the result 0.1588×10^1. Notice how the last two values vanish.
Backward Pass

1. $x_5 + x_4$:
   - **Shift** $0.9763 \times 10^{-5} = 0.009763 \times 10^{-3}$
   - **Chop** to 0.0097×10^{-3}
   - **Add** $10^{-3} \times (0.0097 + 0.9895) = 0.9992 \times 10^{-3}$
2. $(x_5 + x_4) + x_3$:
   - **Shift** $0.9992 \times 10^{-3} = 0.009992 \times 10^{-1}$
   - **Chop** to 0.0099×10^{-1}
   - **Add** $10^{-1} \times (0.0099 + 0.1289) = 0.1388 \times 10^{-1}$
3. $(x_5 + x_4 + x_3) + x_2$:
   - **Shift** $0.1388 \times 10^{-1} = 0.01388 \times 10^0$
   - **Chop** to 0.0138×10^0
   - **Add** $10^0 \times (0.3429 + 0.0138) = 0.3567 \times 10^0$
4. $(x_5 + x_4 + x_3 + x_2) + x_1$:
   - **Shift** $0.3567 \times 10^0 = 0.03567 \times 10^1$

- **Chop** to 0.0356×10^1
- **Add** $10^1 \times (0.0356 + 0.1234) = 0.1590 \times 10^1$

Which gives us the result 0.1590×10^1. Here, no values vanish, meaning that the result is expected to be more accurate.

The exact answer is 1.590789263, and as expected the backwards pass is more accurate. This is because, in the forward pass, since we always align to the higher exponent to maintain normalisation, the small values added later leave the window of the positional number system, leading to their values not being accounted for in the final sum.

1.4

With the given formulation, we expect an error to arise by way of catastrophic cancellation near $x = 0$, since there $\cos x$ is close to 1, and the subtraction of two nearby numbers will lead to a loss in precision. In the worst case, this leads to the denominator being 0, which makes the whole expression uncomputable. We can substitute $1 - \cos x = \frac{\sin^2 x}{1+\cos x}$, which avoids this cancellation.

Implementing this in code, we see that the first expression cannot be computed for $x = 1e{-}9$, due to the aforementioned division by 0. However, the second expression correctly evaluates to 2 for $1e - 9$.