1. **Error in Symmetric Rounding vs. Chopping**

Here we show that the upper bound in the relative error for symmetric rounding is a factor of two smaller than the relative error in chopping. First, we must consider two cases separately: one where the first dropped digit is zero, and one where the first dropped digit is 1.

In base-2, we consider a real number $x = \pm 0.b_1 b_2 b_3 \ldots \cdot 2^e = \pm(\sum_{l=1}^{\infty} b_l 2^l)2^e$, and a corresponding machine number $rd(x) == \pm 0.b_1 b_2 b_3 \ldots b_p \cdot 2^e = \pm\left(\sum_{l=p+1}^{\infty} b_l 2^l\right)2^e$. For this proof, p represents the number of bits in the mantissa, and e represents the number of bits in the exponent. As mentioned above, since rounding a number has two cases, we will calculate $x - rd(x)$ for both.

Case 1: $b_{p+1} = 0$

$$\begin{aligned} x - rd(x) &= 0.00 \ldots 0 b_{p+1} \cdot 2^e \\ &= b_{p+1}.b_{p+2}b_{p+3} \ldots \cdot 2^{e-p-1} \end{aligned}$$

Since $b_{p+1} < 1 \Longrightarrow b_{p+1} = 0$, we get:

$$x - rd(x) = 2^{e-p-1}$$

Case 2: $b_{p+1} = 1$

$$\begin{aligned} x - rd(x) &= 0.00 \ldots 0\left(2 - b_{p+1}\right) \ldots \cdot 2^{e-p-1} \\ &= -\left(2 - b_{p+1}\right) \cdot 2^{e-p-1} \end{aligned}$$

Since $b_{p+1} \geq 1 \Longrightarrow b_{p+1} = 1 \Longrightarrow \left(2 - b_{p+1}\right) = 1$, we get:

$$x - rd(x) = -2^{e-p-1}$$

In both cases, the absolute value provides an upper bound: $|x - rd(x)| = \frac{2^{e-p}}{2}$. So, to find $\max\left|\frac{x-rd(x)}{x}\right|$, we look for $\frac{\max|x-rd(x)|}{\min|x|}$. The smallest nonzero number written in binary/fractional scientific notation is $2^{-1}2^e$. We plug this in:

$$\max\left|\frac{x - rd(x)}{x}\right| = \frac{\max|x - rd(x)|}{\min|x|} = \frac{2^{e-p-1}}{2^{e-1}} = 2^{-p}$$

So, this value of 2^{-p} is the machine epsilon, or ϵ, or eps.

2. **An Accurate Implementation of e^x**

   a. We approximate $e^{5.5}$ by working out the terms in this series up to $n = 30$, assuming a mantissa with five decimal digits. Summing together all those terms (while rounding to 5 decimal digits after each addition, division, and multiplication operation) we get $e^{5.5} \approx 244.71$ for $n = 30$. See the appendix for the code output.

   b. Partial sums can also be seen in the code output mentioned in part (a), denoted by S_k where $S_{k+1} = s_k + \frac{(5.5)^{k+1}}{(k+1)!}$. As we found out in part (a), the value of k where $e^{5.5}$ converges to 5 significant figures is S_{17}, because $S_{16} = S_{17} = 244.71$. Comparing this to the real value of $e^{5.5} = 244.6919$, the magnitude of the relative error is 0.4013%. The attached output of all values and partial sums for parts (a) and (b), the columns being n, $\frac{5.5^n}{n!}$, and S_n.

   c. Now we will do the same as part (b), but we will add the terms from right to left (starting with S_{30} and working backwards). These partial sums can be seen in the output. As can be seen by the output of the code (below the tables), the error is 0.007384%. The same value was reached, but it took more iterations to get to this value.

   d. We will now look at $e^{-5.5}$ four different ways:

      i. Adding from left to right
      ii. Within each partial sum, adding right to left
      iii. When computing S_k, add all the positive terms left to right, then all the negative terms left to right, then combine those results
      iv. When computing S_k, add all the positive terms contributing to that partial sum right to left, then add all the negative terms right to left, then combine those results.

      The outputs can be seen attached, but here we summarize a few results:

Algorithm	**Value of n of convergence**	**Relative Error**
L2R	$n = 25$	6.128%
R2L	$n = 25$	6.128%
L2R Sign Separate	$n = 16,17$	100%
R2L Sign Separate	$n = 16,17$	100%

      When the signs are separated, it seems that the two series cancel each other out when working in a 5-digit mantissa (this can be seen in the script outputs for $(d)(iii)$ and $(d)(iv)$. So, this results in an error of 100%, which is terrible. The lowest error out of these methods is *L2R* and *R2L*, which is much less error, but still terrible. All these errors are way worse compared to when we used this Taylor series to evaluate $e^{5.5}$, and it makes sense – when we were evaluating $e^{5.5}$,

we added many terms, so the error didn't suffer from subtractive cancellation. But now, when we evaluate $e^{-5.5}$ using a Taylor series, we start to subtract similar-sized terms.

What I would expect to see here is a difference in output between the L2R and the R2L calculations via the code. The order in which operations are taken can change the error, so I'm not sure why I am getting the same values for L2R and R2L. When doing this by hand, the value for $(d)(ii)$ converges to 0.004, and the value for $(d)(iv)$ converges to 0.01. 0.004 isn't too far off from to $e^{-5.5}$ evaluated in the computer (which is 0.004086771438464067), but 0.01 has a terrible error.

e. Most of the errors seem to come in through subtractive cancellation due to the negative odd terms in the expansion of $e^{-5.5}$. Especially as the numbers start getting close together, the error can magnify greatly. To get rid of this exponent altogether, we can take the reciprocal of $e^{5.5}$ (calculated in parts a and b). This will yield us with a better, more accurate representation of $e^{-5.5}$ since there will be no subtractive cancellation.

To validate this, let's take the reciprocal of $e^{5.5}$ calculated from part (a), using the 5-digit mantissa. $rd\left(\frac{1.0}{244.73}\right) = 0.00408613$. The error of this compared to the actual value of $e^{-5.5}$ is $\left|\frac{x-rd(x)}{x}\right| = 0.00015695481720097166$, an error of 0.01% - which is much better than any of the errors from calculating $e^{-5.5}$ using the Taylor expansion.

3. **Error Propagation in Exponentiation**

   a. The whole-number exponent x^n could be calculated in two ways, repeatedly multiplying by x n-times or evaluating $e^{n \ln x}$. We conduct an error analysis for both, keeping in mind the ansatz: if " $\circ$ " represents one of the four elementary arithmetic operations, and x and y are machine numbers, then we say $fl(x \circ y) = (x \circ y)(1 + \varepsilon)$, where $|\varepsilon| \leq \text{eps}$ (ϵ).

      i. First, we say that $fl(x * x) = x^2(1 + \epsilon_1)$, assuming x is a perfect machine number. Now, we can compute $fl\big(x * fl(x * x)\big)$:

$$\begin{aligned} fl\big(x * fl(x * x)\big) &= fl\big(x * x^2(1 + \varepsilon_1)\big) \\ &= x^3(1 + \varepsilon_1)(1 + \varepsilon_2) \\ &= x^3(1 + 2\epsilon) \end{aligned}$$

      We then derive an expression for the error due to repeated multiplication:

$$fl(x^n) = x^n(1 + (n - 1)\epsilon)$$

      ii. For computing $e^{n \ln x}$, three operations occur. We assume that n, as well as x, are perfect machine numbers and do not result in rounding errors by introducing these numbers. We also assume the following:

$$\begin{aligned} fl(\ln x) &= \ln x\,(1 + \varepsilon), \qquad |\varepsilon| \leq \text{eps} \\ fl(\exp x) &= \exp x\,(1 + \varepsilon), \qquad |\varepsilon| \leq \text{eps} \end{aligned}$$

      With this, we derive the error in computing $e^{n \ln x}$:

$$\begin{aligned} fl(\ln x) &= \ln x\,(1 + \varepsilon_{lnx}) \\ fl\big(n * fl(\ln x)\big) &= fl\big(n * \ln x\,(1 + \varepsilon_{lnx})\big) \\ &= n \ln x\,(1 + \varepsilon_{lnx})(1 + \varepsilon_*) \\ &= n \ln x\,\big(1 + \varepsilon_2 + O(\varepsilon^2)\big) \text{ where } \varepsilon_2 = \varepsilon_{lnx} + \varepsilon_* \\ fl\left(\exp\left(fl\big(n * fl(ln\,x)\big)\right)\right) &= fl\big(\exp\big(n \ln x\,(1 + \varepsilon_2)\big)\big) \\ &= \exp\big(n \ln x\,(1 + \varepsilon_2)\big)\,(1 + \varepsilon_{ex}) \\ &= \exp(n \ln x)\exp(n\varepsilon_2 \ln x)\,(1 + \varepsilon_{ex}) \end{aligned}$$

      From this point onward, we want to match this expression above to the original function $e^{n \ln x}$ times one plus some expression in terms of ϵ. We will replace all ε with ϵ, the machine number, to establish an upper bound on this operation. We make use of exponent rules, and do a first-order Taylor expansion of the $e^{2\epsilon n \ln x}$ term

$$\begin{aligned} &= \exp(n \ln x)\,(1 + 2\epsilon n \ln x)(1 + \epsilon) \\ &= \exp(n \ln x)\,\big(1 + 2\epsilon n \ln x + \epsilon + O(\epsilon^2)\big) \end{aligned}$$

So repeated exponentiation results in a relative error of $(n-1)\epsilon$, and the log-exponential method results in a relative error of $(2\epsilon n \ln x + \epsilon)$.

b. Now, say we allow arbitrary exponents. Suppose x is positive and a is nonzero. We will show the propagated relative error ε in x^a for two cases: when either a or x is subject to relative error $\varepsilon_a, \varepsilon_x$, respectively, the other will be an exact machine number. To do this we will use Taylor's theorem to calculate the error, neglecting quantities above first-order: $f(x+h) = f(x) + hf'(x) + O(h^2)$. In these derivations, $a\varepsilon_a$ and $x\varepsilon_x$ will be the h in the Taylor expansion. See the attached work for these two expressions:

$$\begin{aligned} x^{a(1+\varepsilon_a)} &= x^{a+\varepsilon_a a} \\ &= x^a + a\varepsilon_a x^a \ln x + O(\varepsilon_a^2) \\ &= x^a(1 + a\varepsilon_a \ln x) \end{aligned}$$

$$\begin{aligned} \big(x(1+\varepsilon_x)\big)^a &= (x + x\varepsilon_x)^a \\ &= x^a + ax^{a-1}(x\varepsilon_x) + O(\varepsilon_x^2) \\ &= x^a(1 + a\varepsilon_x) \end{aligned}$$

For the first scenario, the propagated error might become substantial as $x \to 0$, because then $\ln x \to -\infty$. Also, in both scenarios, as $a \to \infty$, then the error will also approach infinity.

4. **Conditioning.** For this problem, we consider the function $f(x) = 1 - e^{-x}$ on $[0,1]$.

   a. The condition of $f(x)$ is $(\text{cond } f)(x) = \left|\frac{xf'(x)}{f(x)}\right| = \left|\frac{x(e^{-x})}{1-e^{-x}}\right| = \left|\frac{x}{e^{x}-1}\right|$. For $x = 1$, then $(\text{cond } f)(x) \approx 0.582$. $(\text{cond } f)(x)$ is undefined at $x = 0$, so we use L'Hôpital's rule to find that $\lim_{x\to 0}(\text{cond } f)(x) = 1$. Since there are no maximums or minimums on $f'(x)$ in $[0,1]$, then this function decreases from 1 to ~0.582, meaning it is less than 1 on $[0,1]$. So, the problem of determining $f(x)$, given x, is well-determined on this interval.

   b. Now we want to look for $(\text{cond } A)(x)$ of the algorithm for this function, which is flipping the sign bit, computing $\exp(-x)$, and then subtracting that result from 1. We will say that flipping the sign bit introduces no error, as well as x itself is a perfect machine number. We know that $fl(-x) = -x$, so we just need to consider $fl\big(1 - fl(\exp(-x))\big)$:

$$\begin{aligned} fl(e^{-x}) &= e^{-x}(1+\varepsilon_x) \\ fl\big(1 - fl(e^{-x})\big) &= fl(1 - e^{-x}(1+\varepsilon_x)) \\ &= \big(1 - e^{-x}(1+\varepsilon_x)\big)(1+\varepsilon_r) \\ &= 1 - e^{-x} - \varepsilon_x e^{-x} + \varepsilon_r - e^{-x}\varepsilon_r + O(\varepsilon^2) \\ &= 1 - e^{-x} - \varepsilon_x e^{-x} + (1-e^{-x})\varepsilon_r \\ &= (1-e^{-x})\left(1 - \frac{e^{-x}}{1-e^{-x}}\varepsilon_x + \varepsilon_r\right) \end{aligned}$$

   We want to now set this final expression to the true mathematical algorithm being evaluated at some other input, x_A, and then try to solve for $x - x_A$:

$$\begin{aligned} 1 - e^{-x_A} &= (1-e^{-x})\left(1 - \frac{e^{-x}}{1-e^{-x}}\varepsilon_x + \varepsilon_r\right) \\ 1 - e^{-x_A} &= (1-e^{-x})(1+\varepsilon_r) - e^{-x}\varepsilon_x \\ e^{-x_A} &= e^{-x} - \varepsilon_r(1-e^{-x}) + e^{-x}\varepsilon_x \\ x_A &= -\ln(e^{-x}(1 + \varepsilon_r + \varepsilon_x - \varepsilon_r e^{x})) \\ x_A &= x - \ln(1 + \varepsilon_r + \varepsilon_x - \varepsilon_r e^{x}) \end{aligned}$$

   It is here we will use the fact that for small values of δ, $\ln(1+\delta) \approx \delta + O(\delta^2)$

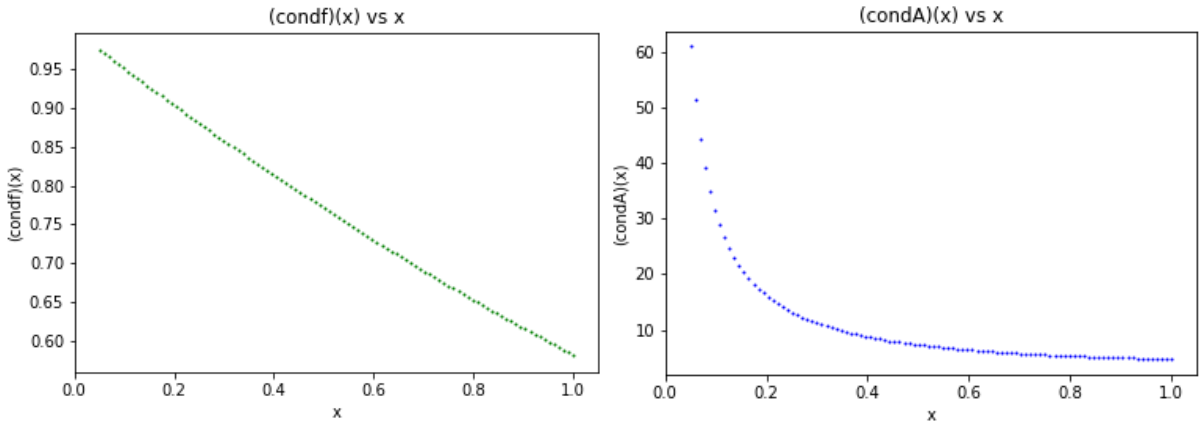
$$\begin{aligned} x_A - x &= -(\varepsilon_r + \varepsilon_x - \varepsilon_r e^{x}) \\ |x_A - x| &= \epsilon(2 + e^{x}) \end{aligned}$$

   So, our expression for the condition of the algorithm is:

$$\frac{|x_A - x|}{|x|} \leq \epsilon\left(\frac{2+e^{x}}{x}\right)$$

   And this is everywhere greater than 1.

c. Below are two plots, the one on the left for $(\text{cond } f)(x)$ vs x, and the one on the right for $(\text{cond } A)(x)$ vs x.



The plot on the left shows that $(\text{cond } f)(x)$ is well-conditioned on the interval $[0,1]$, as all values fall below 1 on the interval. However, looking at $(\text{cond } A)(x)$, it becomes more and more ill-conditioned as $x \to 0$. The root cause of this error is in the algorithm itself.

Mathematically, looking at $(\text{cond } A)(x) = \left(\frac{2+e^x}{x}\right)$, as $x \to 0$, this function grows asymptotically (since it is undefined at $x = 0$). The root cause of this error is in the algorithm itself. Looking at $f(x) = 1 - e^{-x}$, as $x \to 0$, then $e^{-x} \to 1$. Subtraction of two close values can result in subtractive cancellation, which means that this algorithm is poorly condition, especially as $x \to 0$.

d. It turns out we can predict how many bits of significance we will lose in an operation $x - y$. Assuming $x > y > 0$, if you show that $2^{-b} \leq 1 - \frac{y}{x} \leq 2^{-a}$, then the number of sig figs lost is between a and b. In our problem, $y = e^{-x}$ and $x = 1$, so we are looking for an x that satisfies some b in $2^{-b} \leq 1 - e^{-x} \leq 2^{-a}$.

Bits Willing to Lose	**Condition**	**Least Value of x**
$b = 1$	$\frac{1}{2} \leq 1 - e^x$	$-\ln(1 - 2^{-1}) \approx 0.69314$
$b = 2$	$\frac{1}{4} \leq 1 - e^x$	$-\ln(1 - 2^{-2}) \approx 0.28768$
$b = 3$	$\frac{1}{8} \leq 1 - e^x$	$-\ln(1 - 2^{-3}) \approx 0.13353$
$b = 4$	$\frac{1}{16} \leq 1 - e^x$	$-\ln(1 - 2^{-4}) \approx 0.06453$

It can be seen from this table that if we are willing to lose more bits due to the subtraction, we can have lower and lower values of x (since the lower x is, the more ill-conditioned the algorithm is).

e. We will now estimate an upper bound on the relative error in the output for each of these values, i.e. we will give an upper bound for $1 - \frac{e^{-x}}{1-e^{-x}}\varepsilon_x + \varepsilon_r$, which (from above) is the relative error in $fl\left(1 - fl(e^{-x})\right)$.

We can rewrite this equation by plugging in $-\ln(1-2^{-b})$ for x in the relative error in $fl\left(1 - fl(e^{-x})\right)$.

$$\begin{aligned}\frac{e^{-(-\ln(1-2^{-b}))}}{1-e^{-(-\ln(1-2^{-b}))}}\varepsilon_x + \varepsilon_r &= \frac{1-2^{-b}}{1-(1-2^{-b})}\varepsilon_x + \varepsilon_r \\ &= \frac{1-2^{-b}}{2^{-b}}\varepsilon_x + \varepsilon_r \\ &= (2^b - 1)\varepsilon_x + \varepsilon_r\end{aligned}$$

So now we can simply plug in b for how many bits we are allowing ourselves to lose in the calculation, and then estimate an upper bound via $\varepsilon \leq \epsilon$ where ϵ is the machine epsilon, or eps.

Bits Lost	**Least Value of x**	**Upper Bound on Relative Error**
$b = 1$	$-\ln(1-2^{-1}) \approx 0.69314$	$1 + 2\epsilon$
$b = 2$	$-\ln(1-2^{-2}) \approx 0.28768$	$1 + 4\epsilon$
$b = 3$	$-\ln(1-2^{-3}) \approx 0.13353$	$1 + 8\epsilon$
$b = 4$	$-\ln(1-2^{-4}) \approx 0.06453$	$1 + 16\epsilon$

In general, if we are losing k bits, then our relative error in the output is $2^k\epsilon$.

f. The underlying math problem here is not ill-conditioned for small x, so we may use an alternate algorithm (one that perhaps avoids subtractive cancellation). For x small, we may approximate this as a Taylor series, as seen in problem 2. However, unlike problem 2, these values are on the interval $[0,1]$. So, calculating something like $1 - \left(1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots\right) = x(1 - \frac{x}{2} + \frac{x^2}{3!} - \cdots)$. For values of $x \in [0,1]$, there will be no subtractive cancellation between terns.

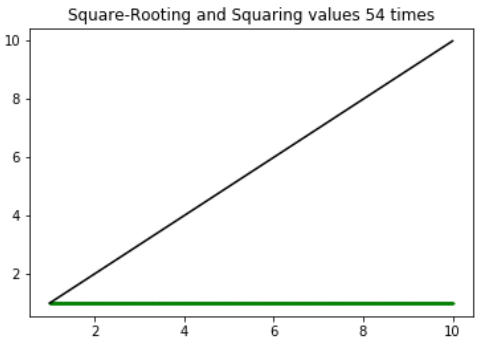
5. **Limits in $\mathbb{R}(\boldsymbol{p}, \boldsymbol{q})$.** The base e of the natural logarithm can be defined analytically by the limit $e \equiv \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$, which defines a sequence. We consider the subsequence of entries for which n is an integer power of 10. Attached is the code, *q5_limits.py*. Output of the code shows the value of n_{stop} where we've achieved 12-sigfig convergence, the final value, and then a table of all intermediate values. This value of n_{stop} is 10^{14}, which gives a final value of 2.716110034087023.

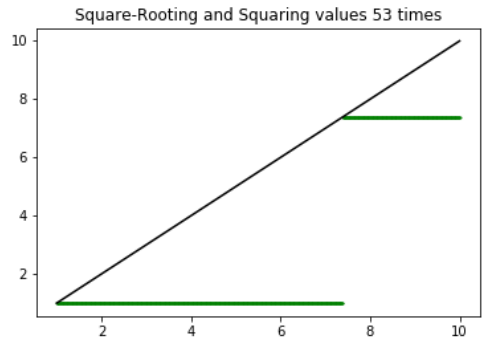
```
-----------------------------------
n                    f(n)
-----------------------------------
1                    2.00000000000
10                   2.59374246010
100                  2.70481382942
1000                 2.71692393224
10000                2.71814592682
100000               2.71826823719
1000000              2.71828046910
10000000             2.71828169413
100000000            2.71828179835
1000000000           2.71828205201
10000000000          2.71828205323
100000000000         2.71828205336
1000000000000        2.71852349604
10000000000000       2.71611003409
100000000000000      2.71611003409
1000000000000000     3.03503520655
10000000000000000    1.00000000000
100000000000000000   1.00000000000
```

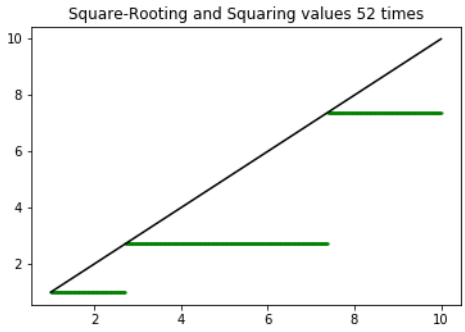
The value of n_{stop} where 13 sig-fig convergence is achieved is 10^{16} and 10^{17}, with a final value of 1.0. This is likely due to the machine epsilon, which is $2^{-52}\sim 2.220446049250313 * 10^{-16}$. When we plug in values on the order of 10^{16}, it is evaluated as $e = \left(1+\frac{1}{10^{16}}\right)^{10^{16}}$. Doing a simple test in Spyder, we find that $\left(\frac{1}{10^{16}}\right)^{10^{16}}$ is evaluated as zero. We also see that evaluating $1+\left(\frac{1}{10^{15}}\right)$ results in 1.000000000000001, while $1+\left(\frac{1}{10^{16}}\right)$ results in 1.0. To further confirm this, when just evaluating 1.0000000000000001 (the true mathematical value of $1+\frac{1}{10^{16}}$) into Spyder, the output itself is 1.0, meaning that this number cannot be stored in the computer. This is because the extra value of 10^{-16} is less than $\epsilon \approx 2.22 * 10^{-16}$. So, the sum before taking the exponent (in the computer) is 1.0, and then this value that is taken to the power of 10^{16} is just 1.0 again.

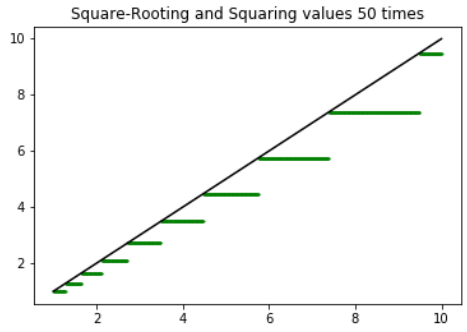
6. **Fun with Square Roots.** See attached script $q6\_sq\_roots.py$ as well as attached data and plots. I created a vector from 0 to 1 with 1001 points, and then took the square root of each value n times and then squared each value n times (where $n \in [49,54]$) . the expected result of the line $y = x$ is not returned, only some values are (close to) their original value. Some attached analysis uses the $logspace()$ function: for each x-value that returned to its original state, I took the natural logarithm of it to see what would happen.

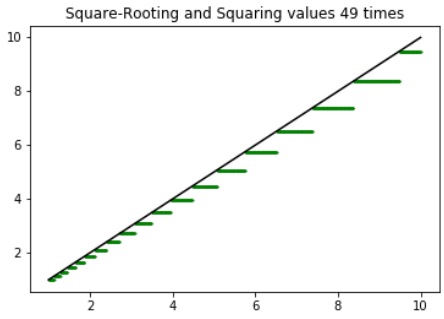
   But now, we will change the spacing from 0.0 to 1.0 to 1.0 to 10.0. Below are some plots that I get for different values of amount of times a square root is taken:











And here are the tables that show the xy-pairs for which the original value is returned:

Number of Iterations	x-values	y-values
52	1.000000 2.719000 7.390000	1.0000000000000000 2.7182818081824731 7.3890559886957758
53	1.000000 7.390000	1.0000000000000000 7.3890559886957758
54	None	None
49	1.000000 1.135000 1.288000 1.459000 1.657000 1.873000 2.125000 2.404000 2.719000 3.088000 3.493000 3.961000 4.483000 5.086000 5.761000 6.526000 7.390000 8.380000 9.496000	1.0000000000000000 1.1331484520102597 1.2840254142932479 1.4549914132630868 1.6487212645509468 1.8682459487159784 2.1170000126693145 2.3988752827774129 2.7182818081824731 3.0802168115967237 3.4903429248936662 3.9550766750613335 4.4816890536418779 5.0784190324237350 5.7546026223204123 6.5208190656088156 7.3890559886957758 8.3728973552260761 9.4877356064430867
50	1.000000 1.288000 1.657000 2.125000 2.719000 3.493000 4.483000 5.761000 7.390000 9.496000	1.0000000000000000 1.2840254142932479 1.6487212645509468 2.1170000126693145 2.7182818081824731 3.4903429248936662 4.4816890536418779 5.7546026223204123 7.3890559886957758 9.4877356064430867

We note that, in the 52 iterations case, the exponents returned are 1, 2.718, and 7.389. These values are very close to e^0, e^1, and e^2. For 53 iterations, we only have e^0 and e^2. For 53 iterations, we get some extra values in between the three values from 52 iterations, namely: $e^{0.5}$ & $e^{1.5}$. For 50 iterations, we have more values appear, and so forth. So, clearly these values are related to some aspect of scientific computing, as well as values of e^x for certain x.

Now I will think about this problem mathematically; to do so, we recall a few facts. For small x, $\sqrt{1+x}$ can be approximated to $1+\frac{x}{2}$. This is the binomial approximation, which states that when δ is small, then $(1+\delta)^{\alpha} \approx 1+\alpha\delta$. So, consider some quantity $(1+\delta)$ that is raised to the power of 2^{-52} (square rooted 52 times) and then raised to the power of 2^{52} (squared 52 times). We can estimate $(1+\delta)^{2^{-52}} \approx 1+2^{-52}\delta$. Then we can take this quantity and estimate $(1+2^{-52}\delta)^{2^{52}} \approx 1+2^{52}2^{-52}\delta = 1+\delta$. So, for these operations, the value $1+\delta$ (whatever that value may be) is returned to its original value.

We know that $e^{x} = \left(1+\frac{x}{n}\right)^{n}$, so when $n=-52$, any remainder from this value is cut off (due to machine precision of 2^{-52}), so we are only left with values that are whole-number exponents of e, i.e. e^{0}, e^{1}, e^{2}, etc. When we bump up n to 2^{-53}, we now shift the mantissa another bit, losing another remainder value. This means that we only get even-values of e, i.e. e^{0}, e^{2}, etc. Now let's go in the other direction – if we do 51 iterations, this is one less bit than the machine epsilon – meaning that we are able to retrieve numbers with "tens-place accuracy." In base-2, this means "0.1 accuracy" (since 0.5 is represented as 0.1 in base-2). In short, this means that we will retrieve values in base-2 that are $0.0, 0.1, 1.0, 1.1, 10.0, 10.1$, etc. – so in base 10, the values we get back are $0.0, 0.5, 1.0, 1.5, 2.0$, etc. – and since these are represented as values in the expression above for e^{x}, we obtain the values back such as $e^{0}, e^{0.5}, e^{1.0}$, etc.

Change machine epsilon in base 2 is 2^{-52}, and in base e, $\text{eps} = e^{-52\ln 2} \approx e^{-36.0436}$. Not sure if this helps, just a conversion I did to see if I got anywhere.

7. **The Issue with Polynomial Roots.** Using Wilkinson's Polynomial up to 20 terms, $w(x) = \prod_{k=1}^{20}(x-k)$, we will try to realize how the problem of finding the roots of an n-degree polynomial is, in general, ill-conditioned.

    a. The factors of Wilkinson's Polynomial are as follows (computed using the attached python script): $x^{20} - 210x^{19} + 20615x^{18} - 1256850x^{17} + 53327946x^{16} - 1672280820x^{15} + 40171771630x^{14} - 756111184500x^{13} + 11310276995381x^{12} - 135585182899530x^{11} + 1307535010540395x^{10} - 10142299865511450x^{9} + 63030812099294896x^{8} - 311333643161390640x^{7} + 1206647803780373360x^{6} - 3599979517947607200x^{5} + 8037811822645051776x^{4} - 12870931245150988800x^{3} + 13803759753640704000x^{2} - 8752948036761600000x + 2432902008176640000$

    b. I coded the function in two ways. At first, I decided to test it with all function coefficients and inputs as integer values, called $w_{int}(y)$. When testing this function for the roots $1-20$, the function returns 0 (as we might expect), and when 0 was plugged into the function, the final coefficient of $20!$ was returned. But, when everything was stored as a float, and the roots were evaluated as floats, the right numbers were not showing up. For root finding, I used the Newton-Raphson method ($optimize.newton$) with an initial guess of 21, the NumPy Polynomial package root finder ($poly.polyroots$) and the general NumPy root finder ($np.roots$). The highest root given for NR 20.00001176025919 – but it should be 20 mathematically. The NumPy polynomial package gave me 20.000542093702702, and the NumPy roots package gave 19.999809291236637. The best estimation of the highest root so far is the Newton-Raphson method.

    c. See the code output for the roots obtained, and the error of largest roots values for changing the coefficient of a_{20} from 1 to $1+\delta$. Perturbing this root even pushes the largest root onto the imaginary plane! The Newton-Raphson method gave a real number for the largest root the entire time (with a max relative error of $\sim 5.88 * 10^{-7}$). But the other two methods gave imaginary roots for the largest root for $\delta = 10^{-8}$ and higher values of δ.

    d. Now we will change a_{20} back to the original value of $20!$ and change a_{19} from -210 to $-210 - 2^{-23}$. Roots 16 and 17 are also pushed to the imaginary plane – which is certainly not the correct root.

    e. We look at a restricted-case error analysis. We define a general monic degree-n polynomial as:

$$\sum_{k=0}^{n} a_k x^k = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + x^n$$

We also consider Ω, a map of the space of n free coefficients of $p(x)$ to the space of n roots, so $\Omega = \Omega(a_0, a_1, a_2, \dots, a_{n-1}), \Omega_k = k^{\text{th}}$ root of $p(x), k = 1 \dots n$. We define a condition matrix Γ_{kl}, as a condition for the k^{th} root, given changes in the l^{th} coefficient. Imposing an L_1 norm, we say $(\text{cond}\ \Omega_k)(\vec{a}) \sum_{l=0}^{n-1}(\Gamma_{kl})(\vec{a})$.

i. We now want to find an expression for $(\text{cond}\ \Omega_k)(\vec{a}) \equiv \sum_{l=0}^{n-1}(\Gamma_{kl})(\vec{a})$, which can also be written as:

$$\sum_{l=0}^{n-1} \frac{a_l \left(\frac{\partial \Omega_k}{\partial a_l}\right)}{\Omega_k}$$

We want to change the partial derivative to be some term in the form of $p'(\Omega_k)$. Taking the derivative of Ω_k with respect to some change in a_l, we analyze this using differentials:

$$a_l x^l \Longrightarrow a_l \to a_l(1 + \varepsilon_a) = a_l + \varepsilon_a a_l = a_l + \delta a_l$$

The roots vary due to this change as well, so $\Omega_k = \Omega_k + \delta\Omega_k$. We know that Γ_{kl} is a square matrix, since $k \in [1 \dots n]$ and $l \in [0 \dots n-1]$. We also know that $p(\Omega_k) = 0$, so taking the derivative, $0 = p'(\Omega_k)\delta\Omega_k + \delta a_l \Omega_k^l$. The first term here is just due to the chain rule in calculus. The second term is calculated as a residual for when the root a_l varies.

How did we get this root? When the derivative at some root Ω_k is taken, the terms that form $p(\Omega_k)$ appear in the expression, so they get cancelled out. Rearranging said expression leaves remaining terms that look like $p'(\Omega_k)$ via the power rule. There are also higher order terms, but we can ignore them here. This leaves us with the expression mentioned above, $0 = p'(\Omega_k)\delta\Omega_k + \delta a_l \Omega_k^l$. Rearranging, we have $\Gamma_{kl} = \left|\frac{a_l \delta\Omega_k}{\Omega_k \delta a_l}\right|$. Therefore, the condition number for roots of a monic polynomial is:

$$\sum_{l=0}^{n-1} \left|\frac{a_l \Omega_k^{l-1}}{p'(\Omega_k)}\right|$$

ii. We will now evaluate this for the roots $r = 14,16,17,20$. The attached script does this, so we will present the results here:

```
Condition number for root r =  14  =  53952975806400.0
Condition number for root r =  16  =  35406640372950.0
Condition number for root r =  17  =  18132120329399.996
Condition number for root r =  20  =  137846528820.0
```

These are extremely large condition numbers for these roots, on the order of about 10^{14}, for the Wilkinson polynomial. As the condition number

approaches infinity, the problem becomes ill-posed, which essentially means that no algorithm can be used to reliably find a solution around the chosen root. This is evident from the beginning parts of this exercise, where a slight deviation of a polynomial coefficient was enough to push the roots onto the imaginary plane.

iii. A sufficiently clever algorithm would therefore probably not be able to help us here. This shows us that the conditioning of polynomial root-finding is, in general, an ill-conditioned problem. A major underlying issue here is the difference in storing the coefficients as integers versus storing them as floats. The coefficients for the 7^{th} through 3^{rd} coefficients cannot be stored as an exact number, which already introduces an error before the root-finding process even begins!

The entire point of this exercise is to prove that the problem itself is poorly conditioned. No matter what the algorithm is, a small perturbation in any of the roots gives will magnify in any algorithm. An algorithm may be well-conditioned, but nothing would help this poorly-conditioned problem.

8. **Recurrence in Reverse.** Here we will work with the sequence of integrals,

$$y_n \equiv \int_0^1 x^n e^x dx, \quad n \geq 0$$

which satisfies the recurrence relation:

$$y_{n+1} = e - (n+1)y_n$$

a. Reversing this relation, we get $y_n = \frac{e - y_{n+1}}{n+1}$. First let's imagine this as a map g_k from y_N to y_k, $k < N$. For some N, then

$$y_{N-1} = \frac{e - y_N}{N}$$

And we can then write the first few values of this relation

$$\begin{aligned} y_{N-2} &= \frac{e - y_{N-1}}{N-1} \\ &= \frac{1}{N-1}\left[e - \frac{1}{N}(e - y_N)\right] \\ &= \frac{1}{N(N-1)}[Ne - e + y_N] \end{aligned}$$

$$\begin{aligned} y_{N-3} &= \frac{e - y_{N-2}}{N-2} \\ &= \frac{1}{N-2}\left[e - \left(\frac{1}{N(N-1)}[Ne - e + y_N]\right)\right] \\ &= \frac{1}{N(N-1)(N-2)}[N^2 - 2Ne + e - y_N] \end{aligned}$$

We see that for some $k < N$, the map g_k from y_N to y_k is a linear function of y_N. For the condition number, we want to find $g_k{}'(y_N)$, which is just the coefficient of y_N. We write out the first few derivatives, using the pattern above, remembering that N and e are constants:

$$\begin{aligned} y'_{N-1} &= g'_k(y_N) = -\frac{1}{N} \\ y'_{N-2} &= g'_k(y_{N-1}) = \frac{1}{N(N-1)} \\ y'_{N-3} &= g'_k(y_{N-2}) = -\frac{1}{N(N-1)(N-2)} \\ y'_{N-4} &= g'_k(y_{N-3}) = \frac{1}{N(N-1)(N-2)(N-3)} \end{aligned}$$

Since $k < N$, we can say $k = N - d$, where d is the difference between N and k. The general expression for $y'_{N-d} = g'_k(y_{N-d+1})$, based on the pattern above, can be written in terms of d:

$$y'_{N-d} = \frac{(-1)^d (N-d)!}{N!}$$

And then since $N - d = k$, we rewrite as:

$$y'_k = g'_k(y_{k+1}) = \frac{(-1)^{N-k} k!}{N!}$$

Now that we have the derivative of this map for $g'(k)$, we can now calculate an upper bound for the condition number for this recurrence relation. In doing so, we note that y_N is always less than y_k for $k < N$ (since $y_n \equiv \int_0^1 x^n e^x dx$ for $n \geq 0$ is a decreasing sequence):

$$\sup(\text{cond } g_k)(y_N) = \sup \left| \frac{y_N (-1)^{N-k} k!}{N!\, y_k} \right| = \sup \left| \frac{y_N k!}{y_k N!} \right| = \frac{k!}{N!}$$

b. Suppose we want a relative error of ε in y_k, and we accept a 100% relative error in y_N, which is the same as taking $y_N^* = 0$ as the starting value of y_N.

$$\frac{\varepsilon_k}{\varepsilon_N} = \varepsilon_k = \frac{k!}{N!} \Longrightarrow N! = \frac{k!}{\varepsilon_k}$$

We leave our answer in terms of $N!$

c. In terms of ε and k, we determine the minimum value of N needed to achieve this target relative error in y_k, and by performing trial-and-error, we get $N = 31!$.

d. See the attached script $q8\_recurrence.py$. Using this backwards relation, we get a value of $y_{20} = 0.12380383076256998$. Comparing this to other sources:

Source	**Estimate of y_{20}**
Q8 Recurrence Algorithm	0.12380383076256998
$scipy.integrate.quad$	0.12380383076256998
Wolfram Alpha	0.12380383076256995

Not bad! This algorithm works pretty well, even assuming a 100% error in N.