PHY407 Lab 2

Monday, October 14, 2024

Jace Alloway - 1006940802 - alloway1

This assignment was compiled in VS Code using the Python and LaTeX extensions. Matplotlib does not produce inline plots in terminal, so the `%matplotlib inline` command was commented out. **Collaborators for all questions: none.**

Equations:

$$\epsilon_N \approx I_{2N} - I_N \tag{1}$$

$$H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x) \quad \text{with} \quad H_0(x) = 1,\, H_1(x) = 2x \tag{2}$$

$$\psi_n(x) = \frac{1}{\sqrt{2^n n! \sqrt{\pi}}} \exp\left(-\frac{x^2}{2}\right) H_n(x) \tag{3}$$

$$\langle x^2 \rangle_n = \int_{-\infty}^{\infty} dx\, x^2 |\psi_n(x)|^2 \tag{4}$$

$$E_n = \hbar\omega\left(n + \frac{1}{2}\right) \tag{5}$$

$$v \equiv g(x) = c\left[\frac{k(x_0^2 - x^2)[2mc^2 - 0.5k(x_0^2 - x^2)]}{2[mc^2 + 0.5k(x_0^2 - x^2)]^2}\right]^{1/2} \tag{6}$$

$$T = 4\int_0^{x_0} \frac{dx}{g(x)} \tag{7}$$

$$x_c = c\sqrt{\frac{m}{k}} \tag{8}$$

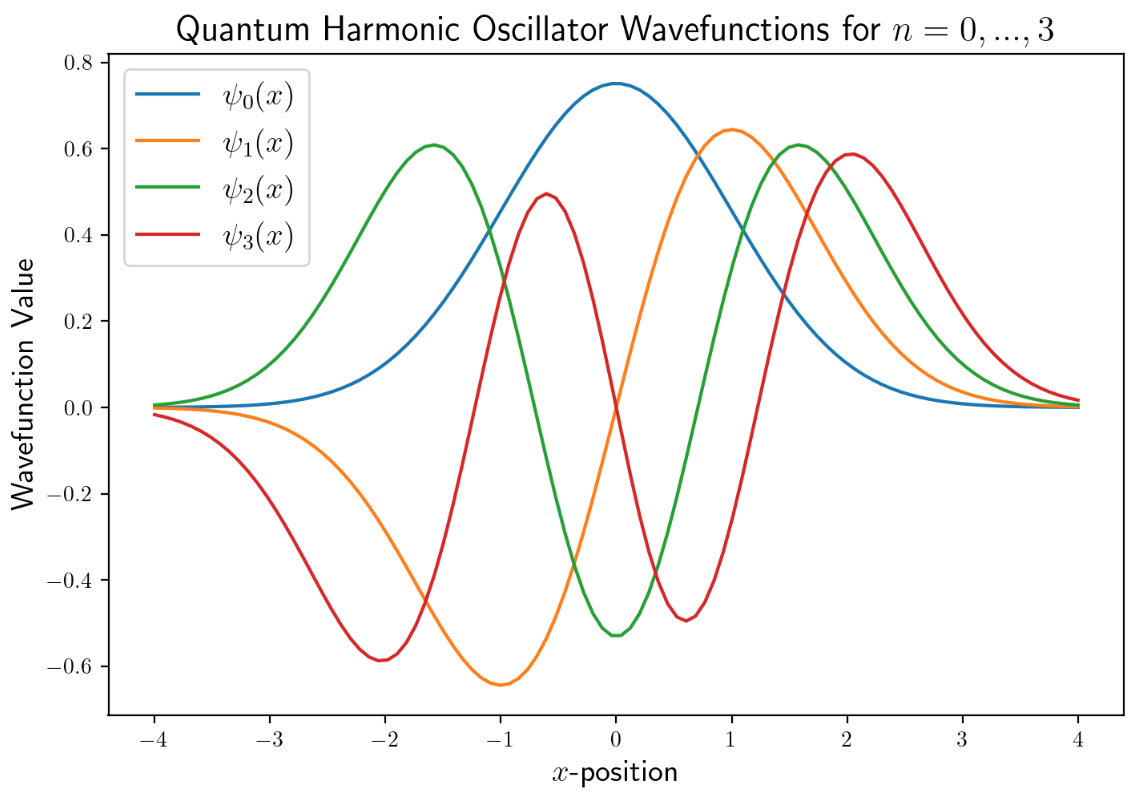
Problem 1

(a) We begin by writing a function `H_n(x)` which calculates the n-th physicists Hermite polynomial at x by invoking a recursion according to equation (2). We require n to be a non-negative integer, so we include ValueErrors if the user gives the function an n which is not non-negative or an integer. We write $H_n(x)$ as a list-operator function, which can operate on a list of x values at a time. This was done by using 2-dimensional arrays and filling them depending on each case of n. The code is included below:

```
def Hermite(
        n: int,
        x: list[int|float]
        ) -> int|float:
    """
    *n*: integer

    *x*: integer or float in list


    **Returns:** The n-th order physicist's Hermite polynomial at x.
    """
    if type(n) == float:            #make sure n is an integer
        raise ValueError('n must be of integer type!')
    if n < 0:                       #make sure n is >= 0
        raise ValueError('Input integer n must be non-negative.')

    M = len(x)                          #length of x input array
    output = np.zeros(M)                #set length of output array -
                                         input array axis x must be list
    H = np.zeros((n+1, M))              #set initial parameters on H
    for m in range(M):
        H[0, m] = 1
        if n > 0:
            H[1, m] = 2*x[m]            #only overwrite H[1] if n is not zero

        if n == 0 or 1:                 #first two initial conditions
            output[m] = H[n, m]         #write output as 1 or 2*x, depending on n
            pass                        #close the if statement and return output


        countlist = np.arange(2, n+1)   #list of k values to iterate over to
                                         determine the next H term
        for k in countlist:             #evalute the recursion relation
            H[k, m] = 2*x[m]*H[k-1, m] - 2*(k-1)*H[k-2, m]
        output[m] = H[n, m]             #overwrite output as the last n-th
                                         term calculated

    return output                       #return array
```

This was particular, since the function required to also take inputs for $n = 0$ and 1, while also defining those initial conditions within the call. If n was greater than 1, then the previous array elements would be called recursively until n hit 2, from which we could just append the list onto the H_0 and H_1 terms. This would be repeated for each x in the input array (doing this meant that we could just feed the function whole arrays at a time, instead of manually having to write for-loops every time we wanted to call the function on a list in x).

(b) The function `QHO_psi(n, x)` was defined according to equation (3) by calling the Hermite function previously defined. Setting the x range as `np.linspace(-4, 4, 100)` (just 100 points felt appropriate) and feeding it into the $\psi_n(x)$ call, each of the wavefunctions could be calculated and displayed. This was done using `matplotlib.pyplot`, whose plots for $n = 0, \ldots, 3$ are shown below:



See code for in-depth review, however I will not list it here. This was just executed using a for-loop over a range of n values.

(c) Using equation (4) and Gaussian quadrature with $N = 100$ samples, the energy levels of the oscillator were calculated numerically. This was done by importing the functions `gausswx(N)` and `gausswxab(N, a, b)` to calculate the roots of the Legendre polynomials and the weights to perform the discrete integration. Since equation (4) is symmetric, only half required integration, since multiplying by two would yield the same value. A change of variables

$$x \to \frac{z}{z-1} \tag{9}$$

was implemented on x^2 and $|\psi_n(x)|^2$ to take the integral bounds between 0 and 1 instead of dealing with the improper integral directly. The integrand was defined as a function seperately to compute the integrand weights $w_k * \psi_n(x_k)$ individually. They were then summed. The function is included below:

```
def PE_integrand(                        #define the function
                                         we are integrating for
                                         potential energy
        n: int,
        z: list[int|float]
        ) -> list[int|float]:
    """
    Compute the integrand for the QHO potential energy.
    """
    if type(n) == float:             #make sure n is an integer
```

```
        raise ValueError('n must be of integer type!')
    if n < 0:                       #make sure n is >= 0
        raise ValueError('Input integer n must be non-negative.')

    x = u(z)        #compute variable change

    #return the function:   d(u(z)) * x^2 * |psi_n(x)|^2
    return (x**2)*(np.abs(QHO_psi(n, x))**2)*(1/((1-z)**2))
```

where $u(z)$ is as equation (9) states, and $\psi_n(x)$ was previously defined. After iterating through $n = 0, 1, \ldots, 10$ using a for-loop, the integral was taken out and the values were printed.

```
The average potential energy values of QHO are
    [0.25 0.75 1.25 1.75 2.25 2.75 3.25 3.75 4.25 4.75 5.25]
for n = 0, ..., 10.
```

We note that this is one-half the total energy of the oscillator, given in equation (5), which is equivalently the expected potential energy of the oscillator or the average energy of the oscillator. That is,

$$\langle U(x) \rangle = \frac{1}{2} m\omega^2 x^2 = \frac{1}{2}\hbar\omega \left(n + \frac{1}{2}\right) \tag{10}$$

where the last equality arises when the potential is transformed into raising/lowering operator-form. For $n = 0, \ldots, 10$, we have the analytical values identical to the energy values written out from the code (Python rounded the numbers when in an array, but they were correct on order 10^{-12}).

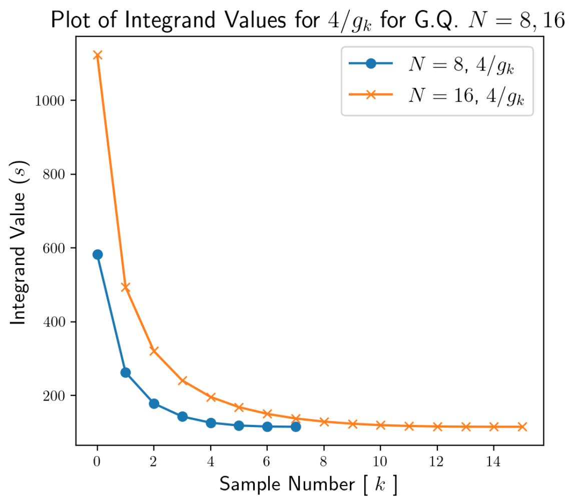
Problem 2

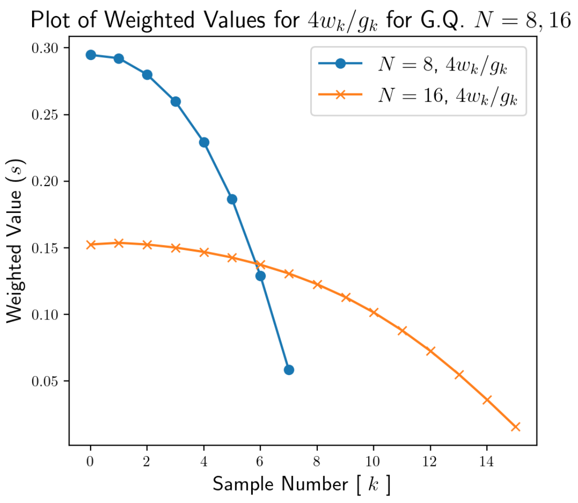
(**a**) Consider the relativistic particle on a spring. The period was calculated by integrating (6) over one cycle ($4 * x_0$), as shown in (7). This was taken out by writing the function $g(x)$ and integrating with Gaussian quadrature for both $N = 8$ and $N = 16$ points.

```
For N=8, T = 1.7301762343365565 with a
        fractional error estimate of 0.023430709023252592 %.

For N=16, T = 1.7707154902422433 with a
        fractional error estimate of 0.011978096533817853 %.
```

where the error was calculated by equation (1) with evaluating the $N = 32$ case as well and taking differences between integral values. Note that upon doubling the sample point number, the relative error approximately halved.

(**b**) With `matplotlib.pyplot`, the weighted values determined in part (a) were plotted.





Notice that as the sample number increases (that is, as the x_0 upper bound is approached), the integrand values become smaller and smaller. Likewise, the weighted values (which are summed) become smaller as well. This implies that the accuracy of the integral could be affected, since rounding error offers an inconvenient solution to managing small numbers. More plainly, rounding error will accumulate more as we sum smaller terms, since they are being rounded prior to summation. This could mean that the integral value could be slightly different than the analytical value for each x_0.

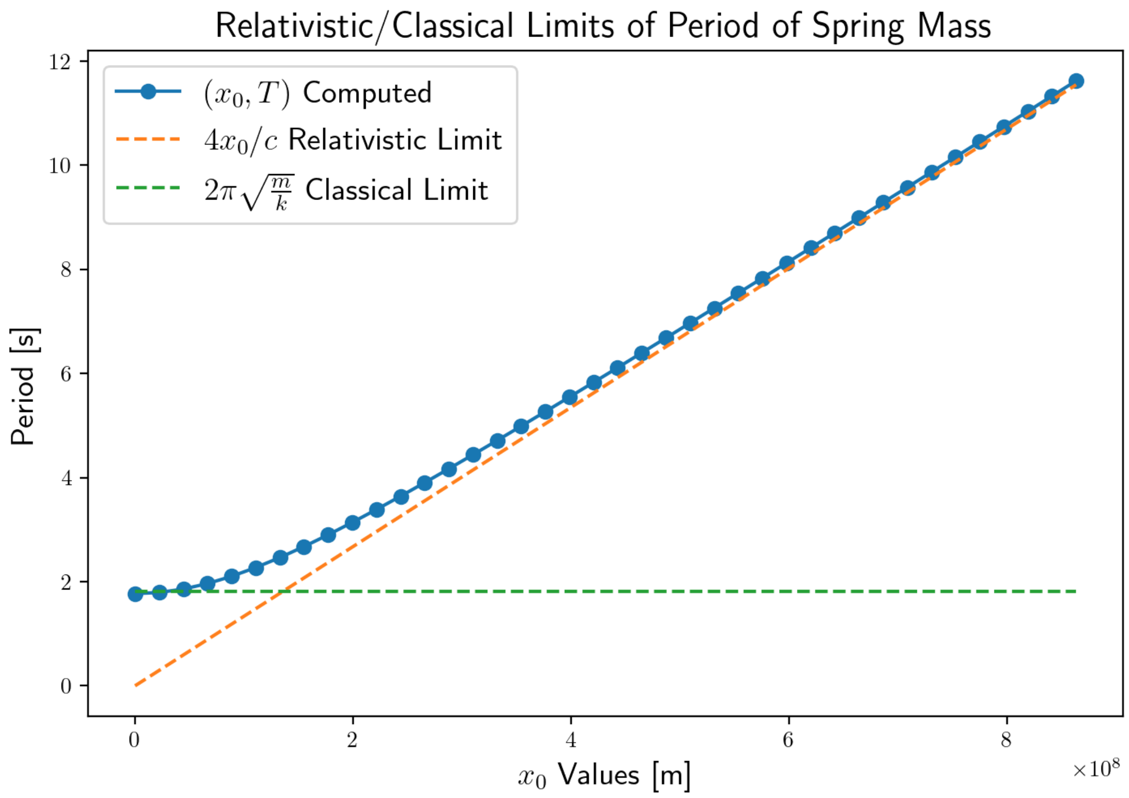
(**c**) x_c was defined as in equation (8), with $c = 2.99e8$ m/s, $m = 1$ kg, and $k = 12$ N/m. A range of x_0 values were chosen as `np.linspace(1.1, 10*xc-0.1, 40)`, with 40 sample points (this was just an arbitrary choice that seemed to give a smooth curve when plotted). A for-loop was used to iterate through `x_range` values at $N = 16$ and to calculate the periods using the integration methods previously outlined. They were then plotted against the `x_range` list. The "relativistic" limit approximation for the period

$$T = \frac{4x_0}{c} \tag{11}$$

was plotted as a sloped line, and the "classical" limit

$$T = 2\pi\sqrt{\frac{m}{k}} \tag{12}$$

was plotted as a horizontal line to compare the $0.5k(x_0^2 - x^2) \gg mc^2$ and $0.5k(x_0^2 - x^2) \ll mc^2$ limits, respectively.



We note that the computed values of T with Gaussian quadrature seem reasonable, since they approach the corresponding asymptotes in either limit in the smooth way as $x_0 \to x_c$ and as $x_0 \to 0$ in `x_range`.

Problem 3

(**a**) To compute the central difference derivative, a `central_difference` function was defined to take in a callable parameter, x and h values and to return the value of the approximated derivative at x. Explicitly,

```
def central_difference(
        func: callable,
        h: int|float,
        x: int|float
        ) -> int|float:
    """
    Computes the single-variable numerical derivative of a function by
    using the method of central differences.

    **Returns:** Approximated central derivative of func at point
    x using width h.
    """
    return (func(x+h) - func(x-h)) / (2*h)
```

(I did this inefficiently. I later had to return to define the "delta" (Δ_h) operator in part (f) to compute higher-order central difference calculations, which, when looking back, I could've used all along.) The range of h values was computed using `np.logspace(lower = -16, upper = 0, base = 10)`. Each derivative was then calculated by iterating through this h list. The h values were returned and printed:

```
    Range of h values:
    [1.e-16 1.e-15 1.e-14 1.e-13 1.e-12 1.e-11 1.e-10 1.e-09 1.e-08 1.e-07
    1.e-06 1.e-05 1.e-04 1.e-03 1.e-02 1.e-01 1.e+00]
```

(**b**) The derivatives of $f(x) = e^{-x^2}$ were calculated using the central difference function defined in (a). They were returned numerically (kind of a long list...) as

```
Respective derivative values: [-1.1102230246251565, -0.7771561172376095,
    -0.7771561172376096, -0.7788214517745473, -0.7788214517745473,
    -0.7787992473140548, -0.7788009126485917, -0.7788008016262893,
    -0.7788007849729439, -0.7788007827524979, -0.7788007830300536,
    -0.7788007830022979, -0.7788007765813232, -0.7788001340710005,
    -0.7787358856669813, -0.7723373144759016, -0.33670077925477027]
```

where we note that the values tend to diverge from the fixed '$-0.778800...$' value at the edges of the h list. The actual derivative was calculated to be

$$f'(x) = -2xe^{-x^2} \implies f'(0.5) \approx -0.778800783071. \tag{13}$$

and was compared against the numerically computed values using relative error (I did the same thing with the approximation error, but obviously the lowest error will be that with the lowest h since the error is proportional to h^3 - hence I didn't include it here). By applying `np.min()` and `np.where()` to locate the index of the smallest relative error value, the respective h value to the smallest relative error was printed.

```
Most accurate h value central approximation (relative) is 1e-06.
```

This value of h is not what is expected according to the expected h to minimuze total error, which is given as

$$h_{\text{min}} = \left[48C\frac{|f(x)|}{|f'''(x)|}\right]^{1/3} \tag{14}$$

(see (5.100) in Newman) which was computed to be

```
The expected h to minimize error (central) is 9.86484829732188e-06.
```

This is therefore relatively consistent with my h value computed above (1e-6), with the exception that I am of course discretizing h steps by factors of 10.

(c) Parts (a) and (b) were repeated by defining a forward difference method to compute the derivative, similar to that of my central difference function noted in (a). The same code was copied and relabelled appropriately. The outputs were given as

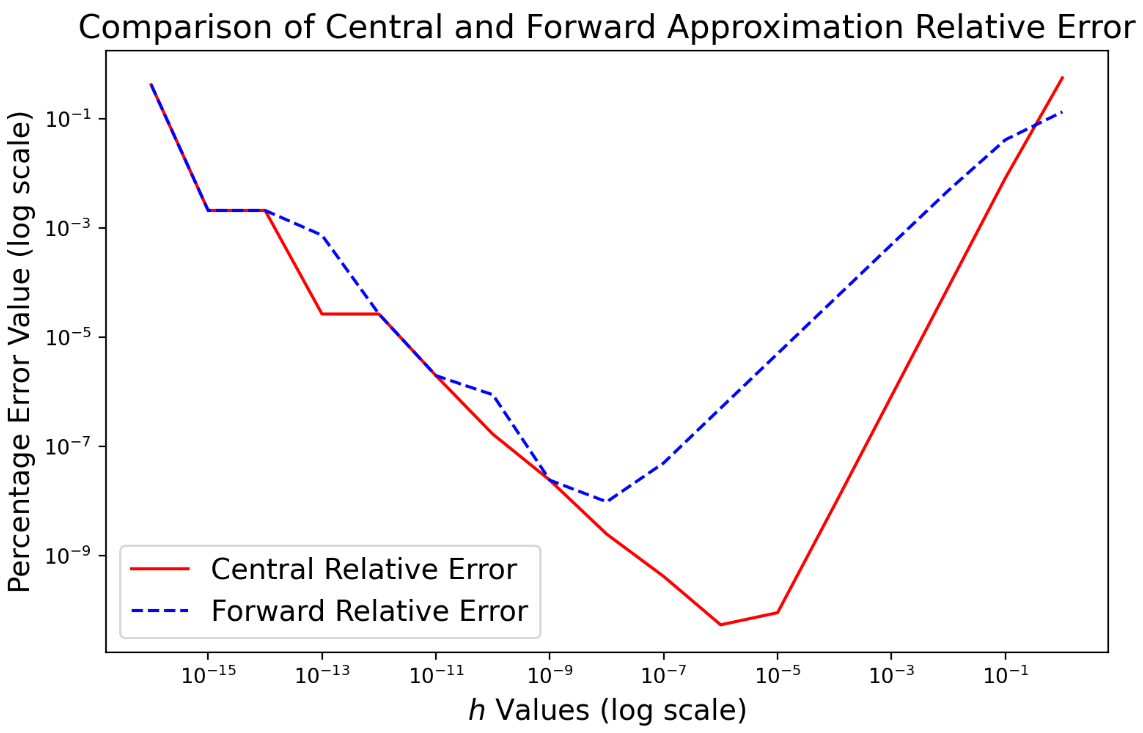
```
Respective derivative values: [-1.1102230246251565, -0.7771561172376095,
-0.7771561172376096, -0.7793765632868599, -0.7788214517745473,
-0.7787992473140548, -0.7788014677601041, -0.7788008016262893,
-0.778800790524059, -0.7788008216103037, -0.7788011724407795,
-0.7788046770040856, -0.7788397166208494, -0.7791895344301247,
-0.782629857128958, -0.8112445700037385, -0.6734015585095405]

Most accurate h value forward approximation (relative) is 1e-08

The expected h to minimize error (fwd) is 2e-08
```

where we have imposed

$$h_{\text{min}} = \sqrt{4C\frac{|f(x)|}{f''(x)}} \tag{15}$$

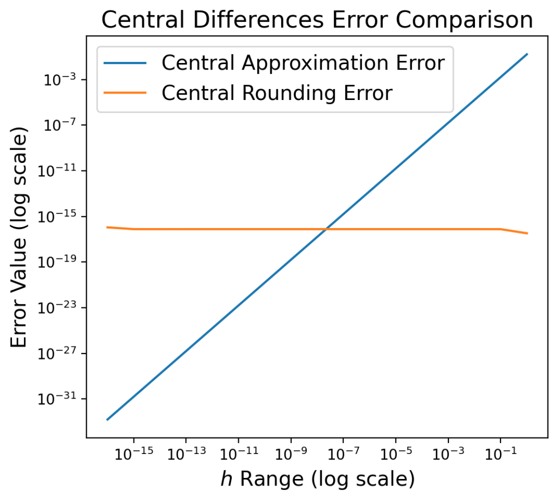
to evaluate the expected minimum h for the forward difference method. Again, this value is close to the actual value of $-e^{-0.25}$, with the h on the same order as the expected value of h to minimize total error according to C.

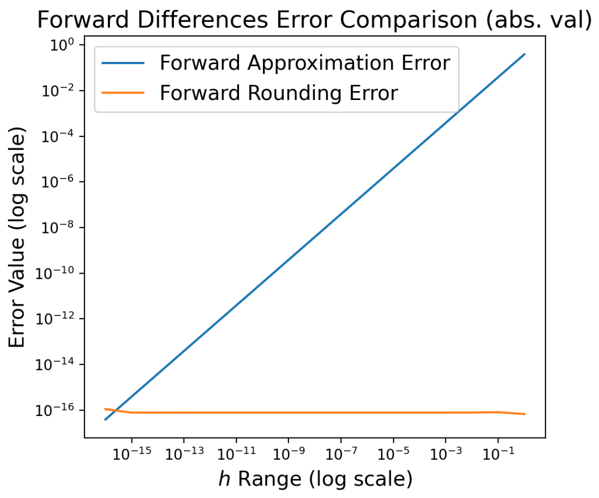
(d) The relative error value differences were portrayed by plotting the relative error outputs against the h axis, using a logarithmic scale on both x and y:



Note the h values which yield minima for both methods. For small h's, both methods produce the same values, however there are values of h which best approach the true integral value (these are shown as minima in the relative error on the plot). Furthermore, note that for all h value, the relative error computed by the central difference method is always less than or equal to that of the forward difference method - except when $h = 1$. This makes sense, because it is generally assumed that the smaller h value approximates the derivative better, and the best choice of h is that which minimized the error.

(e) At high values of h, then, we expect the approximation error to dominate over rounding error, since the rounding error is small while the h values are large. Similarly, at small h values, approximation error is significantly reduced since the actual value of the derivative is approached, hence rounding error will dominate.
Below is plotted these relationships for both the central difference method and the forward difference method, where one may clearly observe where rounding error and approximation error dominate in terms of h.





Here, the rounding error was just computed by multiplying C by each of the numerically computed derivative values at $x = 0.5$.

(f) Lastly, considerthe fucntion $g(x) = e^{2x}$. The analytical derivatives are easy to compute:

$$\begin{aligned} g(x) &= e^{2x} \\ g'(x) &= 2e^{2x} \\ g''(x) &= 4e^{2x} \\ &\vdots \\ g^{(n)}(x) &= 2^n e^{2x}. \end{aligned} \tag{16}$$

These derivatives were computed numerically by defining the Δ_h function (I told you I did this inefficiently...):

```
def delta(        # calculate the m-th derivative of f at x using recursion.
        func: callable,
        x: int|float,
        m: int,
        h: int|float
        ) -> int|float:
    """
    Calculate the *m*-th derivative of a function *func*
    at a point *x* with width *h*.

    **Returns:** [integer or float] value of derivative.
    """
    if m > 1:         #compute value of m-th derivatve with central differences
        return (delta(func, x + h, m - 1, h) - delta(func, x - h, m-1, h))/(2*h)

    elif m==1:        #if m==1, just apply central differences once
        return (func(x + h) - func(x - h))/(2*h)

    elif m==0:        #if m==0, do not do anything
```

```
        return func(x)
```

Hence this could just be implemented in a while-loop with a counter which accumulates until it approaches the desired number of iterations.

```
iterations = 5+1      #set the number of iterations (iterations - 1)
counter = 0          #set counter
g_derivs = []       #set the blank g derivative array
while counter < iterations:                   #repeat for number of iterations
                                               until counter reaches maximum
    g_derivs.append(delta(g, 0, counter, 10.**(-6)))    #append deriv array with
                                                         the m-th derivative
    counter = counter + 1   #up the counter
```

The values of the first 5 derivatives were then printed:

```
    The numerical values of the derivatives of g at x=0 with h=10e-6 are:
    [1.0, 2.000000000002, 3.999994779846361,
    0.0, 13877787.807814457, 6938893903907.229]
```

Note that the first three derivatives are accurate according to (16), but quickly diverge to infinity and are unreasonable. This is most likely because the numerical derivative at high-order is unstable, since the step size is very small. Thus large values tend to "blow up" when divided by $2h$.