Computer Simulation 1

ismisebrendan

October 2, 2023

1 Heron's Method for Finding Roots

1.1 Introduction

Heron's method is an iterative process for finding the roots of equations of the form $x^2 = a$ for $a > 0$. To find the value of a the iteration is

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \tag{1}$$

where x_0 is an appropriate starting value and n is the iteration number.

1.2 Assignment

The assignment was to calculate the square root of 2, plot the value of x_n against n as well as comparing x_n^2 to 2 and plotting the relative error.

1.3 Methodology

A function to iterate through Heron's method to find $\sqrt{2}$ for an inputted value of x_0 and number of iterations was defined. The function was then called with values of $x_0 = [2, 1, -1, 0.5, 10]$ and 10 iterations appeared to be more than enough to get the estimate for the root to a very accurate number for all values of x_0. The values of x_n against n were plotted together. The value $\sqrt{2}$ and $-\sqrt{2}$ (as calculated by numpy) were plotted as dotted lines on the plot to more easily see the convergence of the calculated values to the true value. The relative error between x_n^2 and 2 (ϵ) against n was plotted also.

1.4 Results

The method appeared to attain a highly accurate value for $\sqrt{2}$ within 10 iterations for all inputted values of x_0. It was also found that when the value of x_0 was negative the value converged to $-\sqrt{2}$ which is a root of the equation $x^2 = 2$ (the other being $\sqrt{2}$).

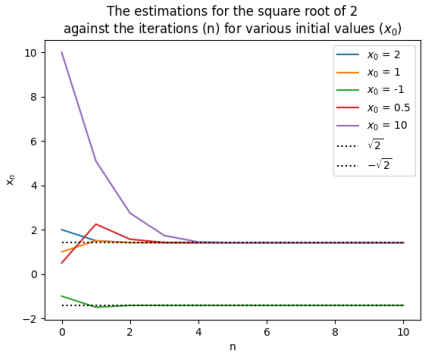

Figure 1: The graph of x_n against n. It can clearly be seen that the values converge quickly to an appropriate value within six iterations at most for these values of x_0.

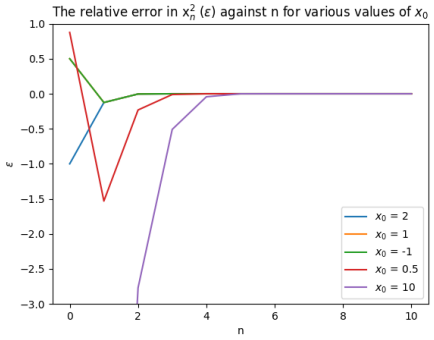

Figure 2: The graph of the relative error between x_n^2 (ϵ) against n. The graph has been limited to y values between -3 and 1 to better be able to see details. The error approaches 0 very quickly, within six iterations for these values of x_0.

The graph of x_n against n is shown in figure 1, it shows the quick convergence of x_n to the roots. The graph of ϵ against n can be seen in figure 2. In this again, the fast convergence can be seen. It has been limited to y values of $[-3, 1]$ to better see the details, as if it had not been limited the initial error for $x_0 = 10$ ($\frac{2-10^2}{2} = 49$) would make the detials of the graph next to unreadable.

2 Overflow and Underflow of Integers and Floats

2.1 Introduction

Computers have only a limited amount of memory to store numbers, as such when numbers get very close to 0 they are usually set equal to 0. This is an underflow and it occurs for variables which begin as both integers and floating point numbers (floats) in python. The opposite can also occur when a number becomes very large. In current versions of python it only occurs for floats as integers can have any arbitrary size while floats are 64 bit numbers [1]. Upon reaching the overflow value floats take on the value 'inf'[2],[3].

2.2 Methodology

In order to find the underflow value of each data type a variable *under* was declared to be 1 (2^0). Then in a loop the number was divided by 2 until python rounded it to 0 at which point the iteration number at which this occurred as well as the number at which this occurred ($2^{-iteration\ no.}$) were outputted. This gave the integer/float underflow value within a power of 2 as desired.

Finding the float overflow value was accomplished similarly, starting with a variable *over* defined initially as 1 it was doubled until its value was saved by python as 'inf', giving the float overflow value.

2.3 Results

The float underflow values were both found to be, within a factor of two, 2^{-1075}. The 'integer underflow value' does not in fact exist, as once an integer is divided to give a number with a decimal point it is converted to a float by python.

The float overflow value was found to be, within a factor of two, 2^{1024}.

3 Machine precision

3.1 Introduction

The machine precision ϵ_m of a computer system is the largest possible number which can be added to 1.0 and return 1.0. This would occur as floats, which are 64 bit numbers, (and other number types with a fixed size are similar) can only store 15-16 decimal points of precision. Below this threshold addition to 1 is treated the same as $1 + 0$.

3.2 Methodology

In order to determine ϵ_m for floats in python a variable *precision_float* was initialised to 1. This was added to 1 and the total was printed. If the calculated total was equal to 1 the last number which, when added to 1 gives a number larger than 1, was printed and the loop was stopped. Otherwise the current value of *precision_float* is stored in *old_pf* in case that was the last value which added to 1 to give a number other than 1, and *precision_float* is halved. This is looped until the total is equal to one as described above.

The same was carried out for complex numbers to find ϵ_m for complex numbers in python.

3.3 Results

The value of ϵ_m for floats in python was found to be 2.22×10^{-16}, within an error of a factor of two.

The value of ϵ_m for complex numbers in python was found to be 5×10^{-324}, within an error of a factor of two.

4 Numerical differentiation

4.1 Introduction

There are a number of ways of differentiating functions numerically, two of which are the forward difference method (see equation 2) and the central difference method (see equation 3).

$$\frac{dy(t)}{dt}_{FD} \simeq \frac{y(t+h) - y(t)}{h} + O(h) \tag{2}$$

$$\frac{dy(t)}{dt}_{CD} \simeq \frac{y(t+h/2) - y(t-h/2)}{h} + O(h^2) \tag{3}$$

Where O(h) and O(h^2) are the order of the leading error term in each method respectively. Clearly then when $h < 1$, as is usually the case, the central difference method is more accurate than the forward difference method.

4.2 Methodology

Functions were defined to calculate the forward and central differences for a given function $f(t)$, at any value of t and with any step size, h. A function was then defined to find, print and plot the derivative of e^t for a given value of t as well as the relative error between the calculated value and the analytic result of the derivative, $\frac{d}{dt}e^t = e^t$ (as calculated by numpy) (ϵ), using both numerical methods. Starting from h

set to 1 the step size was halved each time until it became smaller than the previously calculated machine precision (2.22×10^{-16} for floating point numbers). A similar function was defined to do the same for $\cos t$, whose analytic derivative is $\frac{d}{dt}\cos t = -\sin t$. The functions were called for values of t = 0.1, 1 and 100.

4.3 Results

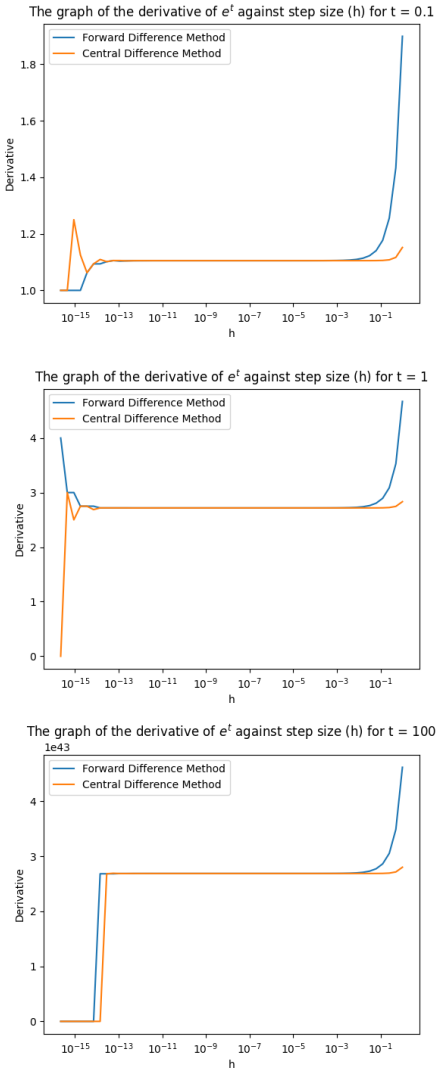

Figure 3: The graph of the derivative of e^t for t = 0.1, 1 and 100 against h. The analytical results are $e^{0.1} = 1.105...$, $e^1 = 2.718...$, $e^{100} = 2.688... \times 10^{43}$. It can be seen that both methods converge to these values in the middle but diverge for very large or very small values of h

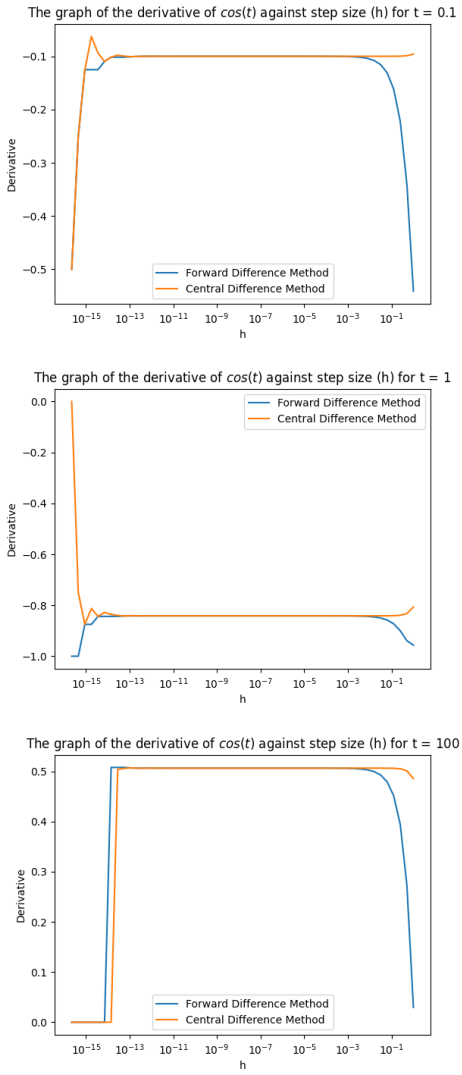

Figure 4: The graph of the derivative of $\cos t$ for t = 0.1, 1 and 100 against h. The analytical results are $-\sin 0.1 = -0.099...$, $-\sin t = -0.841...$, $-\sin t = 0.506...$. It can be seen that both methods converge to these values in the middle but diverge for very large or very small values of h

In all cases the central difference method lost precision at much higher values of h than the forward difference method did, despite consistently having a smaller error at its lowest extent.

The error in the central difference method began to increase as h decreased at values smaller than approximately 10^{-5}. This also occurred for the forward difference method at smaller values of h, around 10^{-8} or 10^{-9}. These observations can be seen in the graphs in figures 5 and 6.

From this one can clearly see that we cannot attain an accuracy for the numerical derivative which

is near the machine precision (ϵ_m) using these methods. Using the central difference method the lowest attainable error is on the order of 10^{-13}, significantly larger than the machine precision of 2.22×10^{-16}. In most cases the error is in fact two or three orders of magnitude larger even than this, while the forward difference method reaches errors on the order of 10^{-9}, again significantly larger than the machine precision. Perhaps with a different differentiation method higher accuracy approaching the machine precision can be obtained.

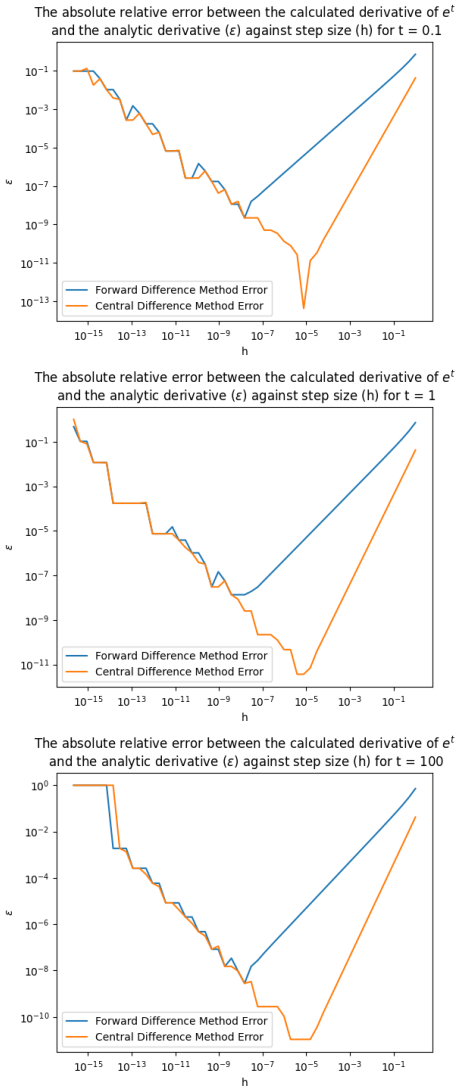

Figure 5: The graph of ϵ against h for the derivative of e^t where t = 0.1, 1, 100. The fact that the error in the central difference begins to increase for h smaller than approximately 10^{-5}, which does not happen for the forward difference method until around 10^{-8} is obvious. The initial smaller error in due to the central difference method than due to the forward difference method is also obvious.

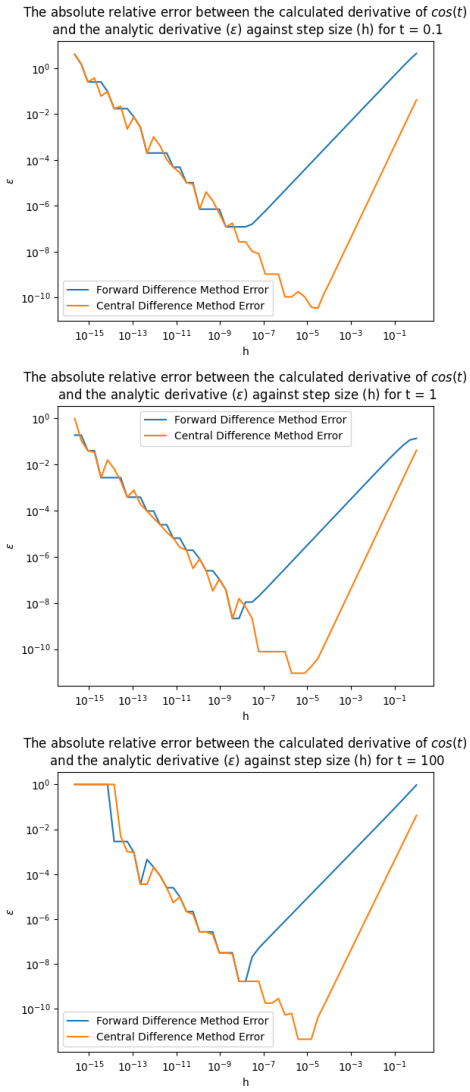

Figure 6: The graph of ϵ against h for the derivative of $\cos t$ where t = 0.1, 1, 100. The fact that the error in the central difference begins to increase for h smaller than approximately 10^{-5}, which does not happen for the forward difference method until around 10^{-8} is obvious. The initial smaller error in due to the central difference method than due to the forward difference method is also obvious.

References

[1] Numpy Documentation. (2023, Sep. 28). *Data types - NumPy v1.26 Manual* [Online]. Available: https://numpy.org/doc/stable/user/basics.types.html

[2] Python Documentation. (2023, Sep. 27). *Integer Objects - Python 3.11.5 documentation* [Online]. Available: https://docs.python.org/3/c-api/long.html

[3] Q. Kong, T. Siauw, A. Bayen. (2020). *Floating Point Numbers - Python Numerical Methods* [Online]. Available: https://pythonnumericalmethods.berkeley.edu/notebooks/chapter09.02-Floating-Point-Numbers.html