Md Rahman

CSC 30100

Section B

Spring 2021

Final Project: Differentiation

5/19/2021

**Table of Contents**

1. *Introduction …………………………………………………..… pg 3*
2. *Numerical Differentiation ………………………………….… pg 3*

2.1) *Forward Difference Formula …………………………... pg 4*

2.2) *Three Point Midpoint Formula ………………………… pg 4*

2.3) *Richardson’s Extrapolation ……………………………. Pg 6*

3) *Symbolic Differentiation …………………………………………. Pg 8*

4*) Automatic Differentiation …………………………………………pg 8*

5*) Comparisons of Differentiation Methods …………...………….pg 10*

5.1) *Example One: Quotient Rule ………………………….. pg 11*

5.2) *Example Two: Product Rule ………………………….. pg 14*

6) *Conclusion ………………………………………………………,, pg 16*

7) *Appendix* *………………………………………..……………….. pg 18*

8) *References ……………………………………………………….. pg 22*

**Final Project: Differentiation**

1. *Introduction*

Differentiation is arguably one of the most important mathematical concepts to date as it makes finding the behavior of a function possible. Using differentiation, we can find the instantaneous rate of change of a function, which is also referred to as the derivative. This derivative is utilized in a variety of fields such as medicine, business, engineering, physics, etc. Furthermore, its application extends to estimating functions, creating an infinite series, optimization, finding concavity, etc. Although the derivatives of trivial functions can be easily evaluated using standard derivative rules, certain functions cannot be accurately evaluated in simple closed forms. As a result, it is necessary to use differentiation techniques to approximate such values. Furthermore, it is vital to use a variety of differentiation techniques in order to observe which techniques are the most effective in terms of performance as well as accuracy. The three main types of differentiation techniques that will be observed are **Numerical Differentiation, Symbolic Differentiation and Automatic Differentiation.**

1. *Numerical Differentiation*

Numerical Differentiation is a form of differentiation that utilizes finite differences when approximating derivatives. It intuitively follows from the following limit definition: in which the numerator of the limit is the finite difference between the function being evaluated at some point of interest, x0 and some sample point h (represents step size) away from the point of interest. When evaluating the limit shown previously, one is essentially calculating the secant line of the function f(x) and in doing the approximation suffers from truncation error or the error present when trying to approximate a mathematical procedure. Note that this secant line is close in proximity to the tangent line of the function and when decreasing the step size in the limit definition, the secant line will get closer to the tangent line as its slope begins to match that of the tangent line until both lines eventually overlap producing the best possible approximation, while minimizing truncation error. However, a new issue arises in which minimizing truncation error will create rounding error, which is the difference between the actual and approximated value(rounded). Fortunately, there are numerous numerical differentiation techniques that help remedy this issue. The three Numerical Differentiation techniques that will be observed are the **Forward Difference formula**, **Three Point Midpoint Formula** and **Richardson’s Extrapolation**.

* 1. *Forward Difference Formula*

The Forward difference formula is the simplest of the three Numerical Differentiation technique since its implementation is identical to the limit definition mentioned previously. The formula for Forward Difference formula can be described by the following expression: . Note that when h is positive, forward difference is being performed in that the difference between the function evaluated at some point x and the function evaluated at some sample point ahead of x by h units is being calculated. Contrastingly, when h is negative, backward difference formula is being computed in which the difference between the function evaluated at some point x and the function evaluated at some sample point behind x by h units is being calculated. Forward difference formula is bounded by the following error bound: , in which M is bounded by |F(2)(x)|. The error term can be represented as .

* 1. *Three Point Midpoint Formula*

The Three Point Midpoint Formula is fundamentally similar to that of the Forward Difference formula in that we must also utilize finite differences to approximate the slope a tangent line. However, unlike the Forward Difference formula we will make use of two sample points rather than one. We will use one sample point that is h units to the right of the point of interest x and h units to the left of x. These same sample points are used in forward difference and backward difference respectively. Note that since we are utilizing two sample points, we will have to scale our division of the finite difference by 2 (now it is 2h). The final expression for the Three Point Midpoint Formula is defined by the following: . Note that since we are using more sample points, the truncation error will decrease faster as h decreases when compared to the Forward Difference formula and indicates that the slope of the secant line will quickly match the slope of the tangent line. The formula is bounded by the following error bound: and its error term can be described by the following expression: .

The Three Point Midpoint formula is effective for approximating derivatives between the intervals of the function, however, issues will arise when approximating points near the intervals of the function, since the formula will try to extract sample points that will be out of range. A solution to this problem is to use the Three Point Endpoint formula which is similar to its midpoint counterpart except it can only use sample points near the end of the intervals. The following expression represents formula for the endpoint version of the Three Point Midpoint formula, . Note that apart from modifying the Three Point Midpoint formula to cater to certain exceptions, we can also extend its formula to include more sample points, thus achieving a more accurate approximation. The only necessary adjustment would be to scale the denominator when dividing by the step size and finding the appropriate finite differences by evaluating the function at the correct positions. The number of evaluations performed depends on the number of points used in the approximation. For example, if we use n points to approximate a derivative, we will perform n-1 approximations. Furthermore, we can expect the error terms to change to depending on the number of points used to approximate the derivative.

* 1. *Richardson’s Extrapolation*

Of the three Numerical Differentiation techniques three Richardson’s Extrapolation is arguably the most complex. Despite its complexity, it produces the most accurate approximation when compared to the other formulae and requires lower order formulas. The reason for Richardson’s Extrapolation being the more accurate formula is due to formulas like the Three Point Midpoint Formula having samples points with positions of x0-h and x0+h that can be almost identical when the value of the step size h is really small leading to possible subtractive cancellation when calculating the finite difference. Note that this form of Numerical Differentiation is only applicable when using an approximation technique whose error term has a predictable form, which is a trait that is present in all forms of extrapolation techniques.

Using Taylor approximation, the formula for Richardson’s Extrapolation can represented by the following expression: Note that the first term of this expression is identical to the formula of Three Point Midpoint formula and the terms that follow are the error terms that are to be reduced. The goal of Richardson’s Extrapolation is to cancel out higher order error terms by accumulating different approximations at different step sizes, which is done iteratively. The approximation at each iteration can be described by the following equation: , where N represents the function that computes the first term of the expression shown previously (this term is identical to the Three Point Midpoint Formula) and j represents the current iteration. The number of iterations is dependent on the number of levels of extrapolation. Note that on each iteration, the current step size h will be scaled by half and the expected final value of the step size is the original h value multiplied by 1/2x, in which x represents the total number of iterations performed. Figure 1 of appendix A shows the Python implementation of Richardson’s Extrapolation. A 2D numpy array is used to store the current approximation at different iterations and the dimensionality of the array is n by n, in which n represents the level of extrapolation. Nested for loops are used to store and compute the appropriate approximation and to update the step size. The best possible approximation which is returned in this function is found in the last cell of this 2D array (nth row and nth column).

3. *Symbolic Differentiation*

As seen previously, Numerical Differentiation techniques can be used to find approximations for derivatives and the accuracy of the Numerical Differentiation technique will vary depending on its implementation. Unlike Numerical Differentiation, symbolic differentiation does not produce a numerical output when evaluating a derivative. Rather it will use standard derivative rules such as chain rule, product rule, quotient rule, etc. to produce an expression (in LaTeX format) when given an expression as an input and such a process requires large quantities of memory. Many programming libraries such SymbolicC++ 3 found in C++ and Sympy found in Python contain these derivative rules within their source code. The Sympy library is used in this project to evaluate derivatives and the main file used to evaluate finite derivatives is function.py, which is found under core folder. Within this file a derivative class is declared and it contains various property decorators, member variables and member functions that keep track of the amount of variables, the symbolic representation of a variables (whether it be x,y or some other form), etc.

Although the implementation of Python’s symbolic differentiation library may seem complex, it is user friendly and simple to use. A variable that is to be used to define a differentiable expression can be symbolically declared using the sy.symbols method whose argument is an equivalent string. After defining an input expression using the symbolically declared variable, sy.diff can be used to differentiate the function to return the correct ouput expression. Finally, evalf can be used to evaluate the output expression at some point of interest to obtain the best possible approximation to a certain numerical precision.

Although Symbolic Differentiation is a highly accurate method of differentiation, it suffers from a few disadvantages. One disadvantage is that it requires closed form expressions, which prevents the use of control flow mechanism such as if else statements, for loops and recursion. Another con of Symbolic Differentiation, arguably the most significant, is that it is susceptible to expression swelling, a phenomenon in which the number of terms in an expression increases dramatically as a calculation (in this case the derivative) progresses. Expression swelling due to Symbolic Differentiation can be portrayed by the following example. Assume there are two functions a(x) and b(x) defined by the expressions x6 + sin(x) and tan(x5) respectively. Some function f(x) is equal to their product and another function c(x) is equal to log(2x2+1). Although we have many initial terms prior to finding the derivative of f(x)\*c(x), we will get a larger number of terms in our output expressions since derivative rules like the product rule will naturally increase the number of terms. Expression swelling makes the process of obtaining derivative approximations slower and only gets worse when trying to approximate higher order derivatives.

1. *Automatic Differentiation*

Automatic Differentiation is a form of differentiation that reaps the benefits found in symbolic and numerical advantages, while steering clear of their faults. It approximates the exact value when evaluating a finite derivative, similar to Symbolic Differentiation and does so by avoiding the costly computation finite differences used in Numerical Differentiation, which are considered numerically unstable. Unlike Symbolic Differentiation, Automatic Differentiation has a time complexity of O(n), assuming the amount of number of inputs matches the number of outputs. The time complexity can be presumably worse when obtaining partial derivatives and the number of independent variables in the expression to differentiated.

Primitive operations are an essential component used in Auto Differentiation to approximate variables. These are intermediate variables that are present in the original function’s implementation. These primitive operations are stored within the nodes of a computational map, with the initial node or nodes being the value or values in which we are trying to evaluate our derivative. The number of nodes will continue to grow as we evaluate more terms from the original function. The last node of the computational map will contain the evaluation of the original function at the specified value or values. Note that while evaluating terms at each node, we are simultaneously keeping track of the tangents of each primitive operation. Therefore, by keeping track of the tangent of the final node, we have successfully approximated the finite derivative.

An example of a computational map is shown in figure 2 of appendix B, which approximates the derivative of ) + at 1.5 and 0.5. Note that the initial nodes are the points at which we want to evaluate the function at, which contains the values 1.5 and 0.5. These two terms form a node that contains the following primitive operation . The derivative of this node is simultaneously kept track of using the quotient rule and is evaluated, giving a result of 3, when evaluated using 1.5 and 0.5. The primitive operation is evaluated within the original function’s sin term so a naturally this leads to an adjacent node, which will store sin (3) and keep track of the derivative of this primitive operation, which is 3cos(3). We continue to reuse previous nodes to form and evaluate new ones until we reach the final node whose value is 2.017(original function evaluated at 1.5 and 0.5) and whose derivative approximation is 3.012(tangent of final node).

Note that in this project, an Automatic Differentiation Python API named auto-diff was utilized to approximate derivatives. Its source code makes use of the concept of the computational map discussed previously. The derivatives of the various primitive operations are stored within a Jacobian Matrix, which contains the approximation of the derivative. Note that this API is compatible with functions with numerous variables, but for the purpose of this project we will only use a single variable, with a single expected output(approximation).

1. *Comparisons of Differentiation Methods*

Using Python programming, two separate tests were performed on all of the differentiation techniques mentioned previously to evaluate them based on speed and accuracy. Each test approximates a different function and requires different derivative rules. The speed of the differentiation is represented by the average time needed to run the technique. This average time is found by evaluating the function 1000 times using a for loop, accumulating the total times of each evaluation and dividing the accumulation of time by 1000. The accuracy of the differentiation technique is made by finding the absolute error of the resulting approximation, which can be found by finding the absolute difference between the actual value and the approximation. Furthermore, comparisons are made between Forward Difference formula and Three Point Midpoint formula using separate Plotly scatterplots to observe how the absolute error of approximation made from both techniques converges to zero as the step size, h decreases.

* 1. *Example One: Quotient Rule*

The function to be differentiated in the first example to test the speed and accuracy of each differentiation technique is defined by the following expression: . The shape of this function can be seen in figure 3.1, which is under appendix C. When using Numerical Differentiation, we are trying to estimate the slope of the blue line, which represents the tangent line of the function. The derivative of this function will be evaluated at 1.5 and different differentiation techniques will be used to approximate the derivative at this same value. Before using any differentiation technique, the expression of this derivative of this value will be found by hand, relying heavily on the concept of the **quotient rule.** Furthermore, the derivative will be evaluated at 1.5 manually using Python. The expression of the derivative of can be described by the following expression: **- –**. Using Python’s math library to evaluate trigonometric functions and basic arithmetic operators we can evaluate the previous expression at 1.5, resulting in an actual value of **-0.540194403156311**, which is represented by 16 significant figures.

When approximating the derivative using Forward Difference formula and Three Point Midpoint Formula, for loops are used (separately) to perform both functions using different step sizes. The for loop of the corresponding differentiation technique is broken once the corresponding absolute error is less than 10-8, which will serve as the tolerance for most differentiation techniques that will be tested. The approximations generated from both Forward Difference and Three Point Midpoint Formulas are saved for different step sizes in two different arrays and these arrays are used to create the scatterplots shown in figures 3.2 and 3.3 contained in appendix c. The scatterplots are of the form of a log-log plot, meaning that both the x and y axes are on a log scale. If a log-log plot depicts a straight line it indicates that the behavior of the plotted data adheres to the power law relationship defined by the following expression: Y =kxn, in which n is the slope of this straight line.

Based on the Forward Difference scatterplot result shown in figure 3.2, the absolute error converges towards zero as the step size decreases and this can be observed by following the plot from the right most step size to the left most step size. Note that the relationship between the absolute error and step size is not entirely depicted as a straight line. There is a steep drop from h = 2 to h = 0.25, which is almost linear. From h=0.25 to the left most h value of 2.384185791015625e-07, there is a straight line indicating that during this interval the power law relationship is satisfied. Additionally, since 2.384185791015625e-07 is the smallest h value that produces an error below 10-8, we consider this to be the ideal step size that meets the requirement of our approximation. Based on the Three Point Midpoint Difference scatterplot result shown in figure 3.3, much like the scat plot of the Forward Difference Formula, the absolute error converges towards zero as the step size decreases. Unlike the previous graph, the graph of the Three Point Midpoint formula seems more stable and is linear from h = 0.5 to h= 0.0001220703125, the smallest h that produces an absolute error less than 10-8.

Using Sympy we were able to create an input expression to be differentiated (identical to the one found by hand). The derivative of this expression was done using sy.diff and was evaluated using evalf and supplying 1.5 as an argument for this function. The final result for the approximation is −0.54301944031563, which includes 15 significant figures, one less than the actual result. Using the auto diff API we were able to supply the example function as an input and obtain a final result of -0.5430194403156311, which is equivalent to the actual value that was computed manually. Note that since the “real” value of the evaluation is a non-terminating decimal value, we cannot obtain a definitive exact result, however, the result from the manual evaluation and Automatic Differentiation is still accurate considering its high numerical precision.

Based on the results shown the table in figure 3.4 of appendix C, Automatic Differentiation was shown to have obtained the most accurate approximation since it has an absolute error of zero since its result matches that of the manual evaluation of the derivative. The symbolic differentiation result, 1.11022302462516 e -16, is also extremely accurate since its absolute error is almost zero, but it still doesn’t have as much numerical precision as the Automatic Differentiation result and manual evaluation. The approximations of all three Numerical Differentiation techniques are less accurate than that of the other two differentiation techniques. Richardson’s Extrapolation was proven to be the most effective of the three Numerical Differentiation techniques with an absolute error of 1.784065117860223e-10.

In regard to speed, all of the Numerical Differentiation techniques displayed better performances when compared to Symbolic and Automatic Differentiation. The Forward Difference formula was shown to have been the quickest of the three Numerical Differentiation techniques, with an average time of 5.74 e-06 seconds (as seen on figure 3.4). This is expected since its implementation has the fewest number of operations. Nevertheless, the average times of Three Point Midpoint Formula and Richardson’s Extrapolation, which are 7.68 e-06 and 6.73e-05 seconds, are still incredibly small and not too far off from the quickest average time. Symbolic Differentiation ran incredibly slow with an average time of 0.000842 seconds, which was noticeably slower than the average time of Automatic differentiation, which had an average time of 0.000348 seconds. The average time of Symbolic differentiation was a little more than double the average time of Automatic differentiation, further displaying the latter’s superior performance.

* 1. *Example Two: Product Rule*

The same testing procedure used in the previous example was reused in the second example when evaluating the different differentiation techniques based on speed and accuracy. The expression to be differentiated is described by the following expression: f(x)g(x), where f(x) = 5x3sin(x) (product of two functions) and g(x) =tan(x). The graph shown in figure 4.1 in appendix D, depicts the plot of f(x)g(x) as well as the tangent line (in blue) that is to be approximated using Numerical Differentiation. Before using any differentiation technique, the expression of this derivative of this value will be found by hand, relying heavily on the concept of the **product rule.** After obtaining the expression of the derivative, the output expression will be evaluated manually using Python’s math library and basic arithmetic operations and will evaluated at x = 2. The expression of the derivative of f(x)g(x) is and its actual value found from manual evaluation at x = 2 is **127.18668307049617,** which has 17 significant figures.

Like the previous example, the absolute errors of the Forward Difference Formula and Three Point Midpoint Formula at different step sizes are saved in two separate arrays. Note that for this example, Forward Difference formula could not obtain an absolute error below 10-8 as the step size ended up being too small, leading to subtractive cancellation. The best possible approximation of the expression using Forward Difference had an absolute error less than 10-6. This issue did not arise when using the Three Point Midpoint formula. As seen in figure 4.2 in appendix D, the log log plot displaying the relationship between absolute error and step size for the second example is linear from h=0.5 to h = 1.4901161193847656e-08, indicating that during this interval the conditions of the power law relationship are met, although the step size with the best possible approximation for Forward Difference is h = 7.450580596923828e-09. As seen in figure 4.3, the log log plot of the Three Point Midpoint formula is linear from h= 0.25 to h = 3.814697265625e-06 (best step size for this formula), indicating that this interval satisfies the conditions of the power law relationship.

Using Sympy, we were able to differentiate f(x)g(x), which resulted in a formula identical to the one found by hand. When evaluating this expression using evalf with 2 as an argument, we obtain a result of 127.186683070496, which is an accurate approximation with 15 significant figures. This is the same number of significant figures found in the result of the previous example. The output of the differential evaluation from using Automatic Differentiation is 127.18668307049617, which is exactly the same as the output of the manual evaluation.

Based on the results of the chart shown in the table found in figure 4.4 in appendix D, Automatic Differentiation was once again the most accurate form of differentiation, obtaining an absolute error of zero. Symbolic differentiation also had a very low absolute error of 1.4210854715202 e -14, which is almost zero. Due its result having less numerical precision than that of Automatic Differentiation, Symbolic Differentiation can be considered the second most accurate differentiation technique. All three Numerical Differentiation techniques had higher absolute error than the other two forms of differentiations, with Forward Difference having the greatest absolute error at 5.842889834184462e-07. This is especially poor considering it could not obtain an absolute error less than 10-8, which was achieved by every other form of differentiation. Richardson’s Extrapolation was once again the more accurate of the other two Numerical Differentiation techniques, obtaining an absolute error of 1.9019381625184906e-09.

In terms of speed, each Numerical Differentiation technique was proven to be faster than the other two forms of differentiation, with Three Point Midpoint Formula proving to having the quickest average time of 7.97 e-06 seconds. The average times for Forward Differentiation and Richardson’s Extrapolation, which are 1.1e-05 seconds and 2.69e-05 seconds, are still impressive considering the increased complexity of differentiating f(x)g(x). Once again, Symbolic Differentiation had the slowest average time at 0.00171 seconds, which was slower than average time it took to evaluate the first example. The drastic fall in speed is most likely due to expression swell since we are approximating a function with more terms. We start with three terms initially and end up with six terms, which is double the number of initial terms. Since Automatic Differentiation avoids expression swell it’s able to obtain an average time of 0.000408 seconds, indicating that this form of differentiation is about 3.5 times faster than Symbolic Differentiation.

1. *Conclusion*

Based on the results of the test performed previously, it’s apparent that the differentiation technique that obtained the most accurate approximation was Automatic Differentiation, since in both examples it showed an absolute error of zero, indicating that it had the same value as the actual value that was manually evaluated with the same numerical precision. The differentiation technique that had the quickest average time was Numerical Differentiation. The specific Numerical Differentiation technique with the fastest average time was Forward Difference when approximating the first example and Three Point Midpoint formula when approximating the second example. Note that the difference between the average times amongst the three Numerical Differentiation techniques are very minimal. It seems as though Automatic Differentiation is the best overall form of differentiation since it provides high accuracy in a short period of time. In order to make more stronger case or verdict for which differentiation technique is truly the best, future experimentation will be done by approximating more unique functions that utilize other derivative rules such as substitution rules and utilizing other Numerical Differentiation techniques such as Five Point Midpoint Formula, Differential Quadrature, etc.

**Appendix**

Appendix A

**Graphical user interface, text, application

Description automatically generated**

Figure 1:Python Implementation of Richardson’s Extrapolation

Appendix B

A picture containing schematic

Description automatically generated

Figure 2: Example of Computational Map

Appendix C

Chart

Description automatically generated

Tangent line

Figure 3.1: Example Function 1 with accompanying tangent line shown in blue

Chart, line chart

Description automatically generated

Figure 3.2: Log - Log plot of Example 1 showing the relationship between step size and absolute error when using Forward Difference Formula

Chart, line chart

Description automatically generated

Figure 3.3: Log - Log plot of Example 1 showing the relationship between step size and absolute error when using Three Point Midpoint Formula

Table

Description automatically generated with medium confidence

Figure 3.4: Table depicting the average time of each differentiation technique and their corresponding absolute error for the first example

Appendix D

Chart, line chart

Description automatically generated

Tangent line

Figure 4.1: Graph of function to be evaluated in second example with accompanying tangent line shown in blue

Chart, line chart

Description automatically generated

Figure4.2: Log - Log plot of Example 2 showing the relationship between step size and absolute error when using Forward Difference Formula

Chart, line chart

Description automatically generated

Figure 4.3: Log - Log plot of Example 2 showing the relationship between step size and absolute error when using Three Point Midpoint Formula

Table

Description automatically generated

Figure 4.4: Table depicting the average time of each differentiation technique and their corresponding absolute error for the second example

**References**

1. *Grosse, R. (n.d.). CSC321 Lecture 10: Automatic Differentiation. https://www.cs.toronto.edu/~rgrosse/courses/csc321\_2018/slides/lec10.pdf.*
2. *Harder, D. W., & Rodgers, J. (n.d.). Background. Topic 12.3: Richardson Extrapolation. https://ece.uwaterloo.ca/~dwharder/NumericalAnalysis/12Differentiation/richardson/complete.html.*
3. *Margossian, C. C. (2019, January 14). A Review of Automatic Differentiation and its Efficient Implementation. https://arxiv.org/pdf/1811.05031.pdf.*
4. *Burden, R. L., Faires, J. D., & Burden, A. M. (2016). Numerical Analysis. Cengage Learning.*
5. *Wang, R. (n.d.). Differentiation by Richardson Extrapolation. http://fourier.eng.hmc.edu/e176/lectures/ch6/node9.html.*
6. *Baydin, A. G. (2018, April 18). Automatic Differentiation in Machine Learning: a Survey. https://www.jmlr.org/papers/volume18/17-468/17-468.pdf.*