COMPUTERIZATION AND COMPARATIVE ANALYSIS OF NUMERICAL INTEGRATION TECHNIQUES

by

Luis A. Flores Carrubio

An undergraduate thesis submitted in partial fulfillment
of the requirements for the degree of

Bachelor of Science

in

Computer Science

University of Puerto Rico at Bayamón

Abstract

The manifold techniques in numerical integration constrains the ability to determine which is the most efficient for finding the approximation of a definite integral. In theory, techniques like the Simpson's Rule provide the exact value of an integral under certain circumstances whereas other techniques involving Riemann sums can only provide approximations. These approximations, however, were obtained almost instantaneously by computerizing the mathematical formulas pertaining to each numerical integration technique. The creation of an open source computer application capable of providing detailed information was essential to determine the efficiency levels of several techniques. Furthermore, the performance of each technique was greatly influenced by the user's given parameters. Therefore, each technique behaved differently and if given the perfect parameters one technique was observed to succeed from the others. The Trapezium Rule for example, when given a function that its second derivative is zero, the technique yields the exact value of the integral. On the other hand, the Riemann sum's approximation was bound to the number of subintervals the user entered and a higher number of subintervals implies an increment in the technique's runtime. The data obtained from the created application answers various questions from which conclusions were made about each numerical integration technique. Furthermore, the obtained data about each technique is consistent with the mathematical theories and fundamentals.

Table of Contents

Introduction

Integral calculus is an eminent field of mathematics that studies the different aspects and techniques for solving integrals. It is a far-reaching field by cause of the many applications it has in the real world. Integration allows for the computation of various important quantities such as area, volume, length, probabilities, force, and mass. Numerical integration techniques are generally utilized when it is desired to obtain the approximated value of a definite integral having an elementary or non-elementary piecewise continuous function as the integrand. In furtherance of understanding numerical integration one must examine the analytical approach for solving Riemann integrals. This approach has the advantage of providing the exact value of an integral but only if the integrand is an elementary function, meaning that the integrand needs to have a known antiderivative. If this condition is not met or if solving the integral becomes an onerous task, then obtaining an approximation by using a numerical approach must suffice. For these reasons, it is of most importance to thoroughly analyze and computerize different numerical integration techniques with the objective of making detailed comparisons to determine which one of them efficiently offers the best approximation. Furthermore, the creation of an open source computer application capable of providing detailed information pertaining to these techniques will be of convenience to scientists or students wanting to reduce any knowledge gap.

Review of Literature

Integration is a key tool to calculate areas, volumes and many important quantities. There is no simple formula for calculating the areas of general shapes having curved boundaries but we can approximate these areas (Thomas 2014). Several numerical integration techniques such as Riemann sums, Taylor series, Newton-Cotes quadrature rules, adaptive quadrature, and extrapolation methods make possible the approximation of definite integrals. Although some of these techniques use a similar approach to approximate the value of an integral, they can differ in terms of implementation and efficiency. According to John Rice (1973), "a considerable amount of evidence has accumulated to indicate that adaptive quadrature is substantially superior to non-adaptive quadrature techniques" (p. 27). The main difference between adaptive and non-adaptive quadrature is that the latter will continue to subdivide intervals if it is found that the accuracy in the approximation will be higher by doing so.

The Riemann sum approach can also be expanded and be utilized for the approximation of multi-dimensional integrals by applying Fubini's theorem (Thomas' 2014). This means that for each increase in dimension another Riemann sum must be evaluated, hence assuring a decline of efficiency if no series acceleration techniques are considered in the implementation. A technique that is capable of approximating multi-dimensional integrals with more efficiency is known as Monte Carlo integration and is based on a stochastic approach (Dalquist 2008). Other techniques for approximating one-dimensional integrals are the Richardson and Romberg methods, known for efficiently achieving high accuracy results by incorporating extrapolation concepts. There are many numerical integration techniques but it is wise to study and evaluate the most recognized ones since these are constantly utilized by the academic and scientific communities.

I. Riemann Sums and the Definite Integral

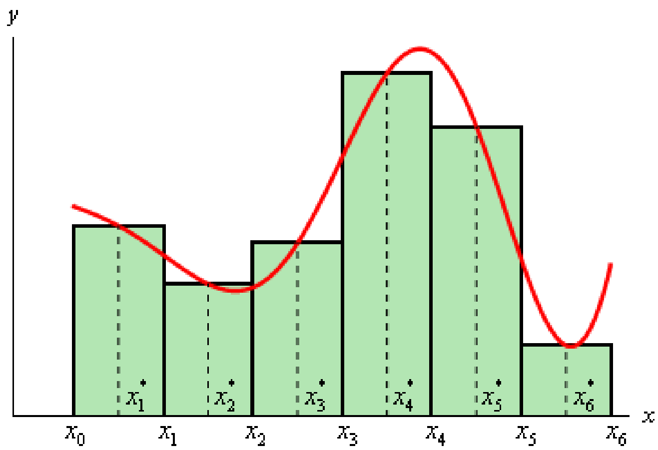

Figure 1. Geometric representation of the midpoint rule (Dawkins 2017).

In calculus, we are taught how to analytically integrate continuous functions with known antiderivatives over closed intervals. This process is known as calculating the definite integral of a function and yields the exact value of the area between the function and a coordinate axis. The underlying theory of the definite integral was made precise by the German mathematician Bernhard Riemann in the 19th century. His work on the theory of limits of finite approximations provided a way of approximating the area under a curve using rectangles with widths approaching zero. According to Thomas' (2014),

> Any Riemann sum associated with a partition of a closed interval defines rectangles that approximate the region between the graph of a continuous function and the x-axis. Partitions in which all subintervals widths approach zero lead to collections of rectangles that approximate this region with increasing accuracy. A single limiting value is approached as the subinterval widths approach zero. (p. 315)

The single limiting value that the Riemann sum converges to as we make the widths of the rectangles infinitely smaller is known as the definite integral. Therefore, by computerizing a

Riemann sum we can obtain a valid approximation of an integral. The limitation of this approach is that the resources of a computer are not infinite hence we can only get an approximation that depends on the number of rectangles and their widths.

Estimating areas using finite sums can become quite cumbersome to do by hand when dealing with big intervals that contain many subintervals. Regardless, the process of establishing the sum formula is straightforward and we can translate this formula into code by using an appropriate programming language and integrated development environment. Thomas' (2014) explains the process involved in the elaboration of the Riemann sum formulae:

> We begin with an arbitrary function f defined on a closed interval $[a, b]$. We subdivide the interval into subintervals, not necessarily of equal widths, and form sums in the same way as for finite approximations. To do so we choose $n - 1$ points $\{x_1, x_2, \ldots, x_{n-1}\}$ between a and b satisfying $a < x_1 < x_2 < \cdots < x_{n-1} < b$. Denote $a = x_0$ and $b = x_n$ to form the set $P = \{x_0, x_1, \ldots, x_n\}$ called a partition of $[a, b]$. The partition P divides the interval into n closed subintervals $[x_0, x_1], [x_1, x_2], \ldots, [x_{n-1}, x_n]$. The width of the first subinterval is denoted by Δx, and the width of the kth subinterval is given by $\Delta x = x_k - x_{k-1}$. If all n subintervals have equal width, then the common width Δx is equal to $\frac{b-a}{n}$.
>
> In each subinterval, we select some point. The point chosen in the kth subinterval is calledC_k. Then on each subinterval we stand a vertical rectangle that stretches from the x axis to touch the curve at $(C_k, f(C_k))$. These rectangles can be above or below the x axis, depending on whether $f(C_k)$ is positive or negative, or on the x axis if $f(C_k) = 0$. On each subinterval, we form the product $f(C_k)\Delta x_k$. This product is positive, negative, or zero depending on the value of the function when we evaluate it at the chosen point.

Finally, we sum all the products to get $S_p = \sum_{k=1}^{n} f(C_k)\Delta x_k$. The sum S_p is called a Riemann sum for f on the interval $[a, b]$. There are many such sums, depending on the partition P we choose, and the choices for the points C_k in the subintervals. For instance, we could choose n subintervals all having equal width, and then choose the point C_k to be the right-hand endpoint of each subinterval when forming the sums. This choice leads to the Riemann sum formula $S_n = \sum_{k=1}^{n} f(a + \frac{k(b-a)}{n})(\frac{b-a}{n})$. Similar formulas can be obtained if instead we choose C_k to be the left-hand endpoint, or the midpoint, of each subinterval.

The definition of the definite integral was made possible by taking the limit of Riemann sums as the width Δx of the rectangles approach zero. Once established the notion of Riemann sums, Thomas' (2014) proceeds to give the formal definition of the definite integral:

Let $f(x)$ be a function defined on a closed interval $[a, b]$. We say that a number J is the definite integral of f over $[a, b]$ and that J is the limit of the Riemann sums $\sum_{k=1}^{n} f(C_k)\Delta x_k$ if the following condition is satisfied: Given any number $\in > 0$ there is a corresponding number $\delta > 0$ such that for every partition $P = \{x_0, x_1, \dots, x_n\}$ of $[a, b]$ with the largest width of all the subintervals less than δ and any choice of C_k in $[x_{k-1}, x_k]$, we have $|\sum_{k=1}^{n} f(C_k)\Delta x_k - J| < \in$. When the limit exists, we write it as the definite integral $J = \int_a^b f(x)dx = \lim_{||p|| \to 0} \sum_{k=1}^{n} f(C_k)\Delta x_k$ provided that the norm of the partitions approaches zero and the number of subintervals goes to infinity.

The theory underlying the definite integral provides a way of obtaining an approximation of the area between a function and the x axis. Furthermore, the definite integral of a function exists even if there is no analytic antiderivative (Kersalé). When using Riemann sums we are in control of how exactly the approximation is going to be computed since the rectangles and subintervals are created with the desired specifications. This allows for the computation of the left Riemann sum, Right Riemann sum and the midpoint rule (***Figure 1)***. It is worth mentioning some theorems pertaining to Riemann sums and the definite integral. Stewart (2015) provides the following two theorems:

1. If f is continuous on $[a, b]$, or if f has only a finite number of jump discontinuities, then f is integrable on $[a, b]$; that is, the definite integral $\int_a^b f(x)dx$ exists.
2. If f is integrable on $[a, b]$, then $\int_a^b f(x)dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i)\Delta x$, where $\Delta x = \frac{b-a}{n}$ and $x_i = a + i\Delta x$. The midpoint rule is given when $x_i = \frac{1}{2}(x_{i-1} + x_i)$.

Riemann sums do not provide the exact value of the integral regardless of which point you choose to evaluate the function. If the exact value of the integral is found analytically then it can be used to calculate the error in the approximation but sometimes the exact value cannot be found. Therefore, we turn to a mathematical theorem that gives the formula for the error bound of the midpoint rule. Suppose that $|f''(x)| \leq k$ for some $k \in R$ where $a \leq x \leq b$. Then $|E_t| \leq k\frac{(b-a)^3}{24n^2}$ (Leclair 2010). With this formula, we can know if it is needed to increment the number of subintervals by verifying the magnitude of the error. In theory, if we keep incrementing the number of subintervals then we will get a better approximation and the error will be very close to zero.

II. Trapezoid Rule

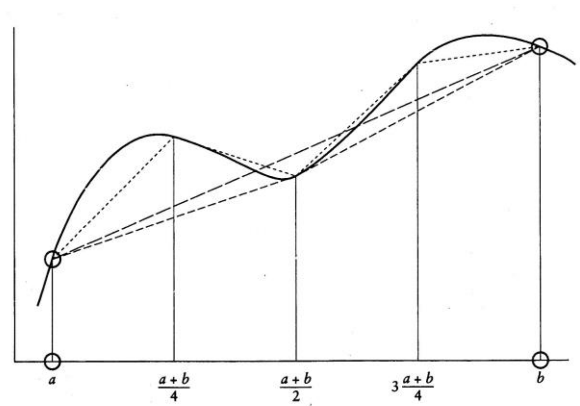

Figure 2. Geometric representation of the Trapezoidal rule (Paul 2010).

The trapezoid rule belongs to a family of formulas known as the Newton-Cotes formulas which are widely used in numerical integration. This technique uses interpolation to find an estimate of the area under a first order linear polynomial between the limits of integration. The definite integral can be then approximated using the polynomial interpolating $f(x)$ through n equispaced points x_k. Thus, the trapezium rule can be obtained by integrating the linear interpolation function over an interval (Kersalé). The resulting formula for the approximation using the trapezoid rule is $\int_a^b f(x)dx \approx (b-a)\frac{f(a)+f(b)}{2}$ (Balhoff).

With this technique, it is advantageous to use more trapezoids of smaller height to better fit the curvature of a graph. The number of trapezoids can be increased by subdividing the interval into many subintervals hence improving the accuracy of the approximation (Paul 2010). If more subintervals are used instead of only the endpoints of the interval then the technique is known as a composite or multiple segment rule ***(figure 2)***. The Newton-Cotes formulas may be "closed" if the interval $[x_0, x_n]$ is included or "open" if the points $[x_1, x_{n-1}]$ are used, or a variation of these two (Weisstein). Thomas' (2014) provides valuable information about the composite trapezoid rule:

> The trapezoidal rule for the values of a definite integral is based on approximating the region between a curve and the x axis with trapezoids instead of rectangles. The only requirement for this rule is for the function to be continuous over the interval of integration $[a, b]$. It is not necessary for the subdivision points $\{x_0, x_1, \ldots, x_n\}$ to be evenly spaced, but the resulting formula is simpler if they are. We, therefore, assume that the length of each subinterval is $\Delta x = \frac{b-a}{n}$. This length is called the step or mesh size. The area of the trapezoid that lies above the ith subinterval is $\frac{\Delta x}{2}(y_{i-1}, y_i)$ where $y_{i-1} = f(x_{i-1})$ and $y_i = f(x_i)$. The area below the curve and above the x axis (assuming $f(x) > 0$) is then approximated by adding the areas of all the trapezoids. This yields the following approximation formula: $\int_a^b f(x)dx \approx \frac{\Delta x}{2}(y_0, 2y_1, \ldots, 2y_{n-1}, y_n)$ where the y's are the values of f at the partition points $\{x_0 = a, x_1 = a + \Delta x, x_{n-1} = a + (n-1)\Delta x, x_n = b\}$.

The error in the trapezoid can be substantial depending on the order of the integrand. If the integrand is a function of first order then the trapezoid rule provides an exact approximation of the area since the second derivative of this function is always zero. When the integrand is a polynomial of higher order the exact value is harder to find given that the second derivative is itself another function and its values keep changing. Thomas' (2014) gives the formal formula for calculating the error in the trapezoidal rule:

> If $f''(x)$ is continuous and M is any upper bound for the values of $|f''(x)|$ on $[a, b]$, then the error E_t in the trapezoidal approximation of the integral of f from a to b for n steps satisfies

the inequality $|E_t| \leq \frac{M(b-a)^3}{12n^2}$. If the conditions are satisfied the error $E_t = \frac{b-a}{12}\ f"(c)(\Delta x)^2$ for some number c between a and b. Thus, as Δx approaches zero, the error also approaches zero.

III. Simpson's Rule

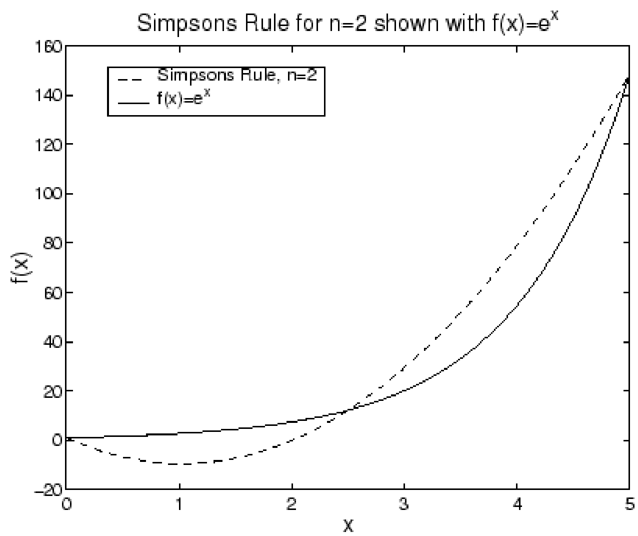

Figure 3. Geometric representation of the Simpson's rule using three points (Restrepo 2001).

The Simpson's rule is another Newton-Cotes formula which uses higher order polynomials to get more accurate estimates. This technique uses a second order polynomial which requires the function to be evaluated at three points of a parabola ***(figure 3)***. The most basic form of the Simpson's rule is called the $\frac{1}{3}$ rd rule which requires for the interval to be broken into two segments or two subintervals. The resulting approximation formula is $\int_a^b f(x)dx \approx \frac{h}{3}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$ (Kaw & Barker). The composite form of the Simpson's rule can be obtained if we subdivide the interval into many segments like the trapezoidal rule. Thomas' (2014) shows how this rule is derived:

Another rule for approximating the definite integral of a continuous function results from using parabolas instead of straight-line segments that produce trapezoids. We partition the interval

$[a, b]$ into n subintervals of equal length $\Delta x = \frac{b-a}{n}$, but this time we require that n be an even number. On each consecutive pair of intervals, we approximate the curve $y = f(x) \geq 0$ by a parabola. A typical parabola passes through three consecutive points $(x_{i-1}, y_{i-1}), (x_i, y_i)$, and (x_{i+1}, y_{i+1}) on the curve. Computing the areas under all the parabolas and adding the result gives the approximation $\int_a^b f(x)dx \approx \frac{\Delta x}{3}\ (y_0 + 4y_1 + 2y_2 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n)$, where the y's are the values of f at the partition points $\{x_0 = a, x_{n-1} = a + (n-1)\Delta x, x_n = b\}$.

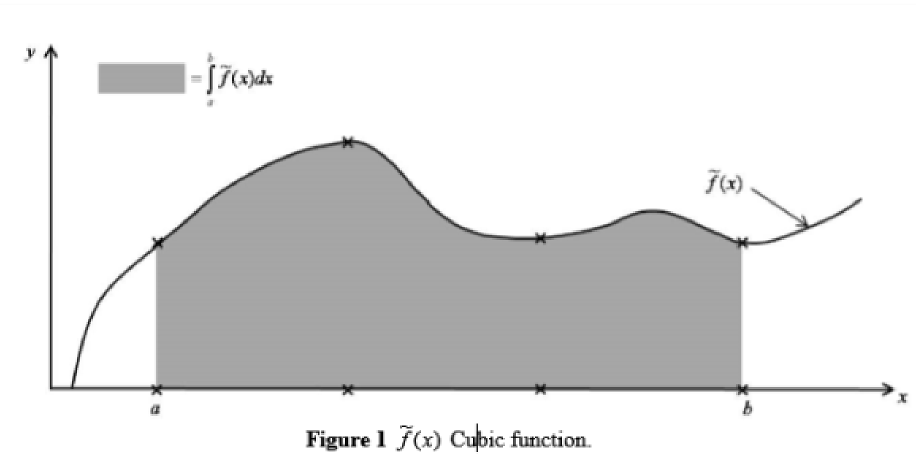

Figure 4. Geometric representation of the Simpson's 3/8 rule (Nguyen 2017).

The other form of the Simpson's rule is known as the 3/8 rule. This rule uses cubic interpolation instead of a quadratic interpolation and requires four points instead of three. The resulting formula for the approximation is $\int_a^b f(x)dx \approx \frac{3h}{8}\ (f(a) + 3f\left(\frac{2a+b}{3}\right) +\ 3f\left(\frac{a+2b}{3}\right) + f(b)$, where $3h = b - a$. This form of the Simpson's rule has an error of the same order as the 1/3rd rule (Balhoff). The error is given by $|E| = \frac{(b-a)f^{(4)}(c)\ h^4}{80}$, where $f(x)$ is sufficiently differentiable and c is an interior point of $[a, b]$ (Mathews 2002).

IV. Series Approximation

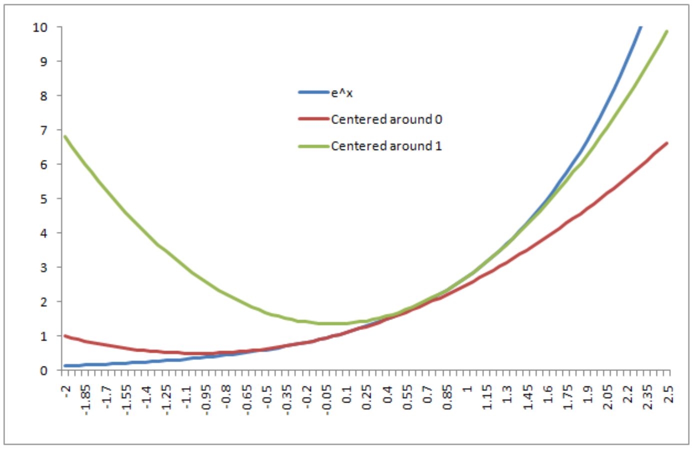

Figure 5. Geometric representation of Taylor polynomials (Tralie 2010).

A Taylor series is a representation of a function as a sum of powers in one of its variables, or by a sum of powers of another function about a point (Weisstein). The Taylor series is named after the English mathematician Brook Taylor even though representing functions as sums of powers series goes back to Newton (Stewart 2008). If a function can be represented by a series expansion then it is possible to use term by term integration to obtain a series representation of the antiderivative of the function. The term by term integration theorem states that:

Suppose that $f(x) = \sum_{n=0}^{\infty} C_n (x-a)^n$ converges for $a - R < x < a + R$ for $R > 0$. Then, $\sum_{n=0}^{\infty} C_n \frac{(x-a)^{n+1}}{n+1}$ converges for $a - R < x < a + R$ and $\int f(x)dx = \sum_{n=0}^{\infty} C_n \frac{(x-a)^{n+1}}{n+1} + C$ for $a - R < x < a + R$ (Thomas 2014).

By convergence it is understood that the partial sums of a series get closer to a value when incrementing the number of terms. The concept can also be understood by reading the definition of the definite integral. Thomas (2014) formally defines the Taylor series:

Let f be a function with derivatives of all orders throughout some interval containing a as an interior point. Then the Taylor series generated by f at $x = a$ is:

$$\sum_{k=0}^{\infty} \frac{f^k(a)}{k!}(x-a)^k = f(a) + f'(a)(x-a) + \frac{f'(a)}{2!}(x-a)^2 + \cdots + \frac{f^n(a)}{n!}(x-a)^n + \cdots$$

Given that Taylor series can be used to express nonelementary integrals in terms of series, we can obtain an approximation of a function's antiderivative by using the above theorems. This process involves finding the function's Taylor polynomial of degree n and evaluating it at an interior point of a given interval $[a, b]$. By applying term by term integration to the polynomial we obtain an approximation of the function's antiderivative. The accuracy of the approximation is affected by the number of terms used. Also, the upper bound error of the Taylor polynomial is given by $|E(x)| \leq \frac{M(b-a)^{n+1}}{(n+1)!}$, where $f(x)$ is a continuous function and M is the function's maximum value over the interval (Khan Academy). This method provides a Taylor polynomial that approximates the antiderivative ***(figure 5)***.

V. Boole's Rule

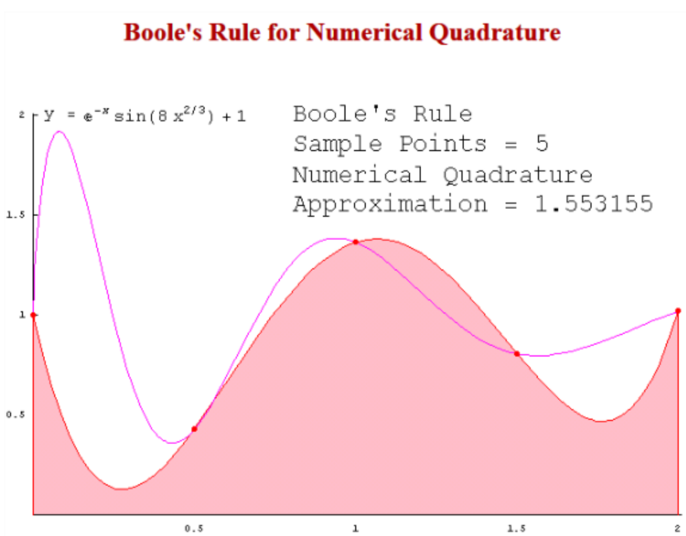

Figure 6. Geometric representation of Boole's rule (Mathews 2002).

Boole's rule is named after George Boole and it is also within the family of the Newton-Cotes formulae. It approximated an integral by using the values of f at five equally spaced points ***(figure 6)***. It is derived by putting n = 4 in the general quadrature formula. Having n = 4 means that f(x) can be approximated by a polynomial of 4th degree so that fifth and higher order differences vanish in the general quadrature formula (Mathews 2002). The composite formula can be obtained by subdividing the interval [a, b] into $4m$ subintervals giving $\int_a^b f(x)dx \approx \frac{2h}{45}\sum_{k=1}^{m} 7f(x_0) + 32f(x_1) + 12f(x_2) + 32f(x_3) + 7f(x_4)$, where $h = \frac{b-a}{4m}$ and $x_k = x_0 + kh$ for $k = 0,1,2 \dots ,4m$. The error of the Boole's rule is given by $|E| = \frac{2(b-a)f^6(c)}{945}\ h^6$ (Mathews 2002).

Solution and Objectives

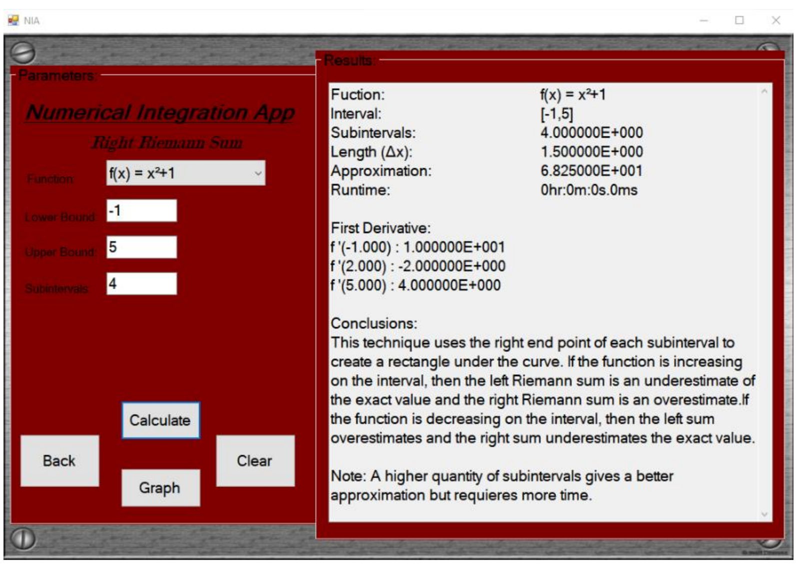

Figure 7. Application's window showing the results of the right Riemann sum.

An open source computer application was created with the purpose of obtaining detailed information pertaining to relevant numerical integration techniques. The data obtained from the application can be helpful to students who are interested in the subject. In ***(figure 7)*** the results of the right Riemann sum numerical technique are presented to the user in an organized manner. In the application, all the necessary data from each studied technique is available and efficiency was determined by calculating the runtime of each technique. This data was used to make conclusions and comparisons with other techniques within the application. Providing the raw data and detailed analysis in an organized manner to the user, allows for the comparison with other techniques that are not incorporated in the application hence increasing its usability. The computerization of several techniques is an essential part of this investigation since the information it provides is of most importance to answer various questions concerning numerical integration. Various closed-source computer applications incorporate some of the techniques earlier mentioned but do not provide any comparative analysis. Therefore, an uninformed user might choose a technique that does not necessarily provide the best approximation. There are several aspects to consider when measuring the efficiency of a numerical integration technique. Some of these aspects include implementation factors, error analysis, runtime, scope, and success rate.

Methodology

The numerical integration techniques that were studied are the Riemann sums, midpoint rule, trapezoidal rule, Simpson's rule, Taylor series, and Boole's rule. The mathematical theory underlying each technique was carefully studied since it is required to code the techniques. The application was created using the programming language Visual Basic .NET and the Visual Studio integrated development environment. Other applications such as Mathematica were used to compare and to make sure that the obtained data from each technique was correct. To calculate a technique's running time the stopwatch class available in the .NET Framework was used. The error in the approximations was calculated by using mathematical theorems. The analytical approach of solving definite integrals also served as a tool to find errors in the approximations since this approach provides the exact value. The application provides a list of elementary and non-elementary integrands from which the user can select. These include exponential, polynomial, logarithmic, and other types of functions. Once the function has been selected, the user will enter any necessary parameter required for a numerical integration technique to begin processing. Furthermore, several tests were conducted to determine how the different techniques behaved. Once the techniques were successfully computerized the data gathering and analysis began. In this phase, a rigorous analysis of all the data obtained from the different techniques took place. Through this analysis, it was determined which technique excels from another and conclusions were made based on the gathered data. These conclusions are provided to the user and be utilized to answer several questions pertaining to numerical integration.

Findings and Conclusions

Technique	Approximation	Error	Runtime
Left Riemann Sum	**0.8961791** n=60	-	0ms
Right Riemann Sum	**0.7987968** n=60	-	0ms
Midpoint Rule	**0.8334809** n=60	**0.027** Max Bound	0ms
Composite Trapezoidal Rule	**0.8408703** n=60	**6.86%** Relative	0ms
Trapezoidal Rule	**2.127358** n=1 (fixed)	**101.11%** Relative	0ms
Taylor Series	**0.8381447** Terms=37	10^{-28} Max Bound	1ms
Boole’s Rule	**0.8381856** n=60	**0.000%** Relative	0ms
Composite Simpson’s Rule	**0.83756611** n=60	**0.024%** Relative	0ms
Simpson’s 1/3	**-2.318091** n=2 (fixed)	**98.6%** Relative	0ms
Simpson’s 3/8	**0.8381698** n=60	**0.001%** Relative	0ms
Wolfram Mathematica	**0.838185**	**-**	0ms
Scientific Calculator	**0.7232084**	**-**	4s

Table 1. Summary of the data obtained from the different techniques approximating $\int_{-1}^{5} \sin x^2 \, dx$.

Through the completion of this investigation it was determined which of the studied technique offers the best approximation in the most efficient manner. It was found that the numerical integration techniques behave accordingly to the underlying mathematical theories. In ***(table 1)*** the approximations of each numerical technique is given along with the found error. It was observed that the Riemann sum approximations increase in accuracy as the number of subintervals become large. For example, the left Riemann sum for the same parameters with $n = 4$ subintervals gives an approximation of 0.03142964 which is far from the more accurate approximation of 0.8381856 given by Boole's rule. It can be observed that the midpoint rule approximation stays between the left and right Riemann sums. This behavior indicates that the midpoint rule approximation must be closer to the exact value in comparison to the Riemann sums approximations. The riemann sums also tend to overestimate or underestimate the exact value of the integral depending on the function.

The trapezoidal rule with one subinterval gives an error of 101.11% while the composite rule using 60 subintervals has a 6.86% of error. This drastic difference in error can also be seen when comparing the composite simpson's rule with the Simpson's $1/3^{rd}$ rule. The difference in error is so drastic because one approach uses only one subinterval while the other uses many more. It is evident that the composite version of the trapezium rule is far more efficient since it provides a better approximation. This rule is also superior to the Riemann sums approach and its approximation is close to the midpoint rule in accuracy. In contrast, the taylor series approximation depends not in the number of subintervals but in the number of terms. The approximation obtained by using taylor series requiered the calculation of its first 37 terms to obtain a valid approximation within one millisecond, making it the slowest of the techniques. The Boole's rule provided an approximation which appaears to be close to the exact value when comparing it to the approximation obtained from the Mathematica application. Simpson's 3/8 rule also gave a valid approximation of the integral since it is also a composite rule.

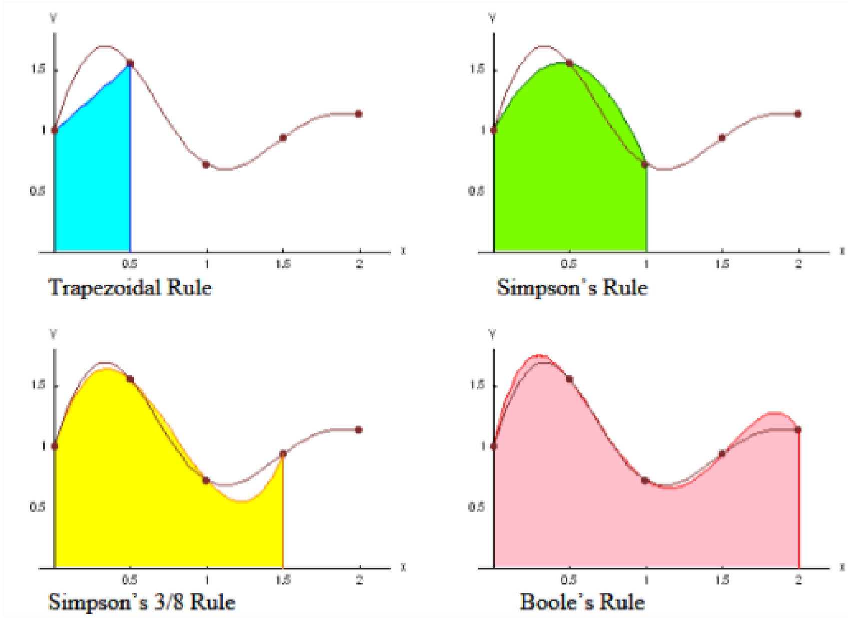

Figure 8. Geometric representation of different numerical techniques. (Mathews 2002).

Testing each technique and observing how they behave under different circumstances allows for conclusions to be made about each one of them. The least efficient techniques are generally the non-composite techniques followed by the Riemann sum technique. Although they provided their approximations instantly when using one million subintervals (***table 2)***, their runtime also depends on the computer being used. Furthermore, other techniques are shown to provide better approximations without the need for many subintervals. The trapezoid rule provides an exact value if the integrand is a polynomial of second order. The same can be said about the Simpson's rule but the polynomial can be of the fourth order or lower. Simpson's 3/8 rule is an improvement over the composite variation because it takes four points instead of three ***(figure 8)***. Therefore, its approximation will be closer to the exact value. Boole's rule uses even more points to calculate the area under a 4th degree polynomial. Most of the techniques were implemented by coding their composite forms. This way, the user can enter $n = 1$ subintervals and obtain the approximation for one subinterval like is shown in ***(figure 8)***.

Technique	Approximation	Error	Runtime
Left Riemann Sum	**17, 217.64** n=1,000,000	N/A	6ms
Right Riemann Sum	**17, 217.70** n=1,000,000	N/A	6ms
Midpoint Rule	**17,217.67** n=1,000,000	10^{-9} Max Bound	6ms
Composite Trapezoidal Rule	**17, 217.70** n=1,000,000	0.000% Relative	6ms
Trapezoidal Rule	**34, 521.50** n=1 (fixed)	100.5% Relative	0ms
Taylor Series	**17, 217.67** Terms=3	10^{7} Max Bound	0ms
Boole's Rule	**17,217.67** n=1	**0.000%** Relative	0ms
Composite Simpson's Rule	**17,217.67** n=4	**0.000%** Relative	0ms
Simpson's 1/3	**17, 217.67** n=2 (fixed)	**0.000%** Relative	0ms
Simpson's 3/8	**17,217.67** n=1	**0.000%** Relative	0ms
Wolfram Mathematica	**17, 217.7**	N/A	0ms
Scientific Calculator	**0.7232084**	N/A	1s

Table 2. Summary of the data obtained from the different techniques approximating $\int_{-10}^{37} x^2 dx$.

Future Work

Several numerical integration techniques are worth studying in the future. These include the Gaussian quadrature, Monte Carlo method, Romberg Integration, and adaptive quadrature. Furthermore, the adaptation of these techniques and a parser to the application is something that will greatly enhance the application's usability. The parser will allow for the user to enter any function and obtain a valid approximation from the desired technique. There are various open source mathematical parsers like UCalc Software which allow for the program to evaluate expressions that are defined at runtime. Also, by implementing a more robust grapher in the application the user will be able to see the area of integration, the subintervals, and any other geometric data that might be helpful in understanding each numerical integration technique.

References

Stewart, J. (2008). “Calculus Early Transcendentals”. Belmont, CA: Brooke/Cole, Cengage Learning.

Weisstein, Eric W. "Newton-Cotes Formulas." From *MathWorld*--A Wolfram Web Resource. Retrieved from http://mathworld.wolfram.com/Newton-CotesFormulas.html

E. Kersalé. *Composite Numerical Integration*. Personal Collection of E. Kersalé, University of Leeds, UK. Retrieved from http://www3.nd.edu/~zxu2/acms40390F11/sec4-4-Composite-integration.pdf

Thomas, G. B., Weir, M. D., Hass, J., Heil, C. (2014). *Thomas' Calculus Early Transcendentals.* Boston, MA: Pearson Education.

Balhoff. *Numerical Integration.* [PDF document]. Personal Collection of Balhoff, University of Texas, Austin.

Paul, A. (2010). *Numerical Integration.* [PDF document]. California State University, Northridge.

Kaw, A., Barker, C. *Integration*. [PDF document]. Personal Collection of Kaw, Holistic Numerical Methods Institute. University of South Florida. Retrieved from http://nm.mathforcollege.com/#sthash.W2VEVLR2.dpbs

Khan Academy. (2017). *Taylor and Maclaurin Polynomials.* [Video file]. Retrieved from https://www.khanacademy.org/math/calculus-home/series-calc/taylor-series-calc/v/generalized-taylor-series-approximation

Leclair, A. (2010). *Error Bound Theorems*. [PDF document]. Personal collection of Leclair, Carnegie Mellon University, Pittsburgh, PA. Retrieved from http://math.cmu.edu/~mittal/Recitation_notes.pdf

Mathews, J., Fink, K. (2002). *Numerical Analysis.* Personal collection of Dr. Mathews, California State University, Fullerton, CA. Retrieved from http://mathfaculty.fullerton.edu/mathews/n2003/NumericalUndergradMod.html

Dawkins, P. (2017). *Approximating Definite Integrals.* Personal collection of Dawkins, Lamar University, Beaumont, Texas. Retrieved from http://tutorial.math.lamar.edu/Classes/CalcII/ApproximatingDefIntegrals.aspx

Weisstein, Eric W. "Taylor Series." From *MathWorld*--A Wolfram [Web Resource]. Retrieved from http://mathworld.wolfram.com/TaylorSeries.html

Restrepo, J. (2001). *Composite Numerical Integration.* Personal collection of Restrepo, The University of Arizona, Tucson, Arizona. Retrieved from http://www.physics.arizona.edu/~restrepo/475A/Notes/sourcea-/sourcea.html

Tralie, C. (2010). *Taylor Series.* [Blog post]. Retrieved from http://www.ctralie.com/Teaching/taylor/

Dalquist, G., Bjorck, A. (2008). *Numerical Methods in Scientific Computing*. [PDF document]. Society for Industrial and Applied Mathematics.

Nguyen, D. (2017). *Simpson's 3/8 Rule for Integration*. [PDF document]. Personal collection of Nguyen, University of South Florida, Tampa, FL. Retrieved from http://mathforcollege.com/nm/mws/gen/07int/mws_gen_int_txt_simpson3by8.pdf

Appendix

Gantt Chart

ID	Task Name	Duration	Start	Finish	Predecessors
1	**Start**	**6 days**	**Fri 2/3/17**	**Fri 2/10/17**	
2	Schedule Creation	6 days	Fri 2/3/17	Fri 2/10/17	
3	Schedule Review and Approval	0 days	Fri 2/10/17	Fri 2/10/17	
4	**Planification**	**6 days**	**Fri 2/17/17**	**Fri 2/24/17**	
5	**Description of Investigation**	**6 days**	**Fri 2/10/17**	**Fri 2/17/17**	
14	Literature Revision	4 days	Fri 2/17/17	Wed 2/22/17	
15	Investigative Hypothesis or Questions	4 days	Fri 2/17/17	Wed 2/22/17	
16	Identify Possible Solutions or Techniques	4 days	Fri 2/17/17	Wed 2/22/17	
17	Software Design Ideas	3 days	Wed 2/22/17	Fri 2/24/17	
18	Expected Findings	3 days	Wed 2/22/17	Fri 2/24/17	
19	New Discoveries	3 days	Wed 2/22/17	Fri 2/24/17	
20	Data Analysis	3 days	Wed 2/22/17	Fri 2/24/17	
21	Conclusions	3 days	Wed 2/22/17	Fri 2/24/17	
22	Future Work	3 days	Wed 2/22/17	Fri 2/24/17	
23	Proposal	3 days	Wed 2/22/17	Fri 2/24/17	
24	Review and Approval	1 day	Fri 2/24/17	Fri 2/24/17	
25	**Analysis**	**26 days**	**Fri 2/24/17**	**Fri 3/31/17**	
26	**Data Gathering**	**14 days**	**Fri 2/24/17**	**Wed 3/15/17**	
27	Research and Existing Projects	6 days?	Fri 2/24/17	Fri 3/3/17	
28	Literature Revision	14 days?	Fri 2/24/17	Wed 3/15/17	
29	**Data Analysis**	**17 days**	**Fri 3/3/17**	**Mon 3/27/17**	
30	**Analysis of Techniques**	**9 days**	**Fri 3/3/17**	**Wed 3/15/17**	
31	Mathematical Formulas	2 days?	Fri 3/3/17	Mon 3/6/17	

Project: Project1
Date: Fri 7/14/17

Task, Split, Milestone, Summary, Project Summary, Inactive Task, Inactive Milestone, Inactive Summary, Manual Task, Duration-only, Manual Summary Rollup, Manual Summary, Start-only, Finish-only, External Tasks, External Milestone, Deadline, Progress, Manual Progress

Page 1

ID	Task Name	Duration	Start	Finish	Predecessors
32	Code Implementation	6 days?	Mon 3/6/17	Mon 3/13/17	
33	Results of Aproximation	3 days?	Mon 3/13/17	Wed 3/15/17	
34	Error Estimation	3 days?	Mon 3/13/17	Wed 3/15/17	
35	Analysis of Algorithms	3 days?	Mon 3/13/17	Wed 3/15/17	
36	Efficiency	3 days?	Mon 3/13/17	Wed 3/15/17	
37	Technique Comparison	8 days	Wed 3/15/17	Fri 3/24/17	
38	Investigative Hypothesis and Questions	4 days?	Wed 3/22/17	Mon 3/27/17	
39	Conclusions	5 days?	Mon 3/27/17	Fri 3/31/17	
40	**Design**	**7 days**	**Fri 3/3/17**	**Mon 3/13/17**	
41	Programming Language	3 days?	Mon 3/6/17	Wed 3/8/17	
42	Input and Output Considerations	4 days?	Wed 3/8/17	Mon 3/13/17	
43	Menu and Windows	4 days?	Wed 3/8/17	Mon 3/13/17	
44	Reports	4 days?	Wed 3/8/17	Mon 3/13/17	
45	Formulas and Format	4 days?	Wed 3/8/17	Mon 3/13/17	
46	First Project Revision	0 days	Fri 3/31/17	Fri 3/31/17	
47	**Development**	**21 days**	**Fri 3/31/17**	**Fri 4/28/17**	
48	**Code**	**6 days?**	**Fri 3/31/17**	**Fri 4/7/17**	
49	Functions and Formulas	6 days?	Fri 3/31/17	Fri 4/7/17	
50	Aproximation Parameters	6 days?	Fri 3/31/17	Fri 4/7/17	
51	Simplicity Factors	6 days?	Fri 3/31/17	Fri 4/7/17	
52	Performance Code	6 days?	Fri 3/31/17	Fri 4/7/17	
53	Reliability	1 day?	Fri 3/31/17	Fri 3/31/17	
54	**Unit Testing**	**6 days**	**Fri 4/7/17**	**Fri 4/14/17**	

Project: Project1
Date: Fri 7/14/17

Task, Split, Milestone, Summary, Project Summary, Inactive Task, Inactive Milestone, Inactive Summary, Manual Task, Duration-only, Manual Summary Rollup, Manual Summary, Start-only, Finish-only, External Tasks, External Milestone, Deadline, Progress, Manual Progress

Page 2

ID	Task Name	Duration	Start	Finish	Predecessors
55	Mathematics Professor	6 days?	Fri 4/7/17	Fri 4/14/17	
56	Computer Science Professor	6 days?	Fri 4/7/17	Fri 4/14/17	
57	Campus Students	6 days?	Fri 4/7/17	Fri 4/14/17	
58	**Documentation**	**5 days**	**Fri 4/14/17**	**Thu 4/20/17**	
59	Organization	5 days?	Fri 4/14/17	Thu 4/20/17	
60	Understandable Explanations	5 days?	Fri 4/14/17	Thu 4/20/17	
61	Formulas and Theory	5 days?	Fri 4/14/17	Thu 4/20/17	
62	**Tests of System Integration**	**6 days**	**Fri 4/7/17**	**Fri 4/14/17**	
63	Software User Friendliness	6 days?	Fri 4/7/17	Fri 4/14/17	
64	User Input Data Easiness	6 days?	Fri 4/7/17	Fri 4/14/17	
65	Understandable Output Data	6 days?	Fri 4/7/17	Fri 4/14/17	
66	**Training**	**4 days**	**Fri 4/14/17**	**Wed 4/19/17**	
67	Menus, Reports, and I/O	4 days?	Fri 4/14/17	Wed 4/19/17	
68	**Implementation**	**8 days**	**Wed 4/19/17**	**Fri 4/28/17**	
69	**Production**	**3 days**	**Wed 4/19/17**	**Fri 4/21/17**	
70	Hardware/Software Requirements	3 days?	Wed 4/19/17	Fri 4/21/17	
71	Software Executable	3 days?	Wed 4/19/17	Fri 4/21/17	
72	Software Installation	3 days?	Wed 4/19/17	Fri 4/21/17	
73	**Configuration**	**2 days**	**Fri 4/21/17**	**Mon 4/24/17**	
74	Available Functions	2 days?	Fri 4/21/17	Mon 4/24/17	
75	Change Aproximation Parameters	2 days?	Fri 4/21/17	Mon 4/24/17	
76	**Security Measures**	**3 days**	**Mon 4/24/17**	**Wed 4/26/17**	
77	Valid Aproximations	3 days?	Mon 4/24/17	Wed 4/26/17	

Project: Project1
Date: Fri 7/14/17

Task, Split, Milestone, Summary, Project Summary, Inactive Task, Inactive Milestone, Inactive Summary, Manual Task, Duration-only, Manual Summary Rollup, Manual Summary, Start-only, Finish-only, External Tasks, External Milestone, Deadline, Progress, Manual Progress

Page 3

ID	Task Name	Duration	Start	Finish	Predecessors
78	Adjustments	5 days	Mon 4/24/17	Fri 4/28/17	
79	Technical Support	5 days	Mon 4/24/17	Fri 4/28/17	
80	Second Project Revision	0 days	Fri 4/28/17	Fri 4/28/17	
81	**End**	**11 days**	**Fri 4/28/17**	**Fri 5/12/17**	
82	Poster	11 days?	Fri 4/28/17	Fri 5/12/17	
83	Oral Presentation	11 days?	Fri 4/28/17	Fri 5/12/17	
84	Finishing Touches	3 days?	Wed 5/10/17	Fri 5/12/17	
85	Presentation	0 days	Fri 5/12/17	Fri 5/12/17	

Project: Project1
Date: Fri 7/14/17

Task, Split, Milestone, Summary, Project Summary, Inactive Task, Inactive Milestone, Inactive Summary, Manual Task, Duration-only, Manual Summary Rollup, Manual Summary, Start-only, Finish-only, External Tasks, External Milestone, Deadline, Progress, Manual Progress

Page 4

Figure 1. Gantt chart detailing the dates and deadlines for the application.

Appendix
Gantt Chart

Figure 2. Gantt chart detailing the dates and deadlines for the application.

Appendix

Source Code

```
'     Copyright(c) 2017, Luis A. Flores
'     All rights reserved.

'Redistribution And use In source And binary forms, with Or without modification,
'are permitted provided that the following conditions are met

'Redistributions of source code must retain the above copyright notice,
'this list of conditions And the following disclaimer.

'   Redistributions in binary form must reproduce the above
'copyright notice, this list Of conditions And the following disclaimer
'in the documentation And/Or other materials provided with the distribution.

'THIS SOFTWARE Is PROVIDED BY THE COPYRIGHT HOLDERS And CONTRIBUTORS
'"AS IS" And ANY EXPRESS Or IMPLIED WARRANTIES, INCLUDING, BUT Not LIMITED
'TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY And FITNESS FOR A PARTICULAR
'PURPOSE ARE DISCLAIMED. In NO Event SHALL THE COPYRIGHT OWNER Or CONTRIBUTORS
'BE LIABLE For ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
'Or CONSEQUENTIAL DAMAGES (INCLUDING, BUT Not LIMITED TO, PROCUREMENT OF
'SUBSTITUTE GOODS Or SERVICES; LOSS Of USE, DATA, Or PROFITS;
'Or BUSINESS INTERRUPTION) HOWEVER CAUSED And ON ANY THEORY OF LIABILITY,
'WHETHER IN CONTRACT, STRICT LIABILITY, Or TORT(INCLUDING NEGLIGENCE Or
'   OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
'EVEN If ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

' Numerical Integration App (NIA)
' Form: frmTechnique.vb
' by Luis A. Flores
' SICI4038 7/14/2017

'frmWelcome Class is responsible For displaying the welcome screen to the user
Public Class frmWelcome
    Private Sub frmWelcome_Load(sender As Object, e As EventArgs) Handles MyBase.Load
    End Sub

    'Start button method shows the menu form
    Private Sub btnStart_Click(sender As Object, e As EventArgs) Handles btnStart.Click
        frmMenu.Show()
        Me.Hide()
```

```
    End Sub

    'Exit button method for quitting the application
    Private Sub btnExit_Click(sender As Object, e As EventArgs) Handles btnExit.Click
        Me.Close()
        Application.Exit()
    End Sub
End Class



****************************************************************************************************

'frmMenu.vb
'The class serves as a menu where the user can select the desired technique. Once a technique is selected
the frmTechnique form will show.

Public Class frmMenu

    'Button method for the Left side Riemann Sum selection.
    Private Sub btnRiemannUpper_Click(sender As Object, e As EventArgs) Handles btnRiemannUpper.Click
        'Label displaying the chosen technique in the form.
        frmTechnique.lblTechName.Text = "Left Riemann Sum"
        frmTechnique.txtTerms.Hide()
        frmTechnique.lblTerms.Hide()
        frmTechnique.lblSubinterval.Show()
        frmTechnique.txtSubinterval.Show()
        'Hiding this form
        Me.Hide()
        'Showing the next form
        frmTechnique.Show()
    End Sub

    'Button subrutine for the right side Riemann Sum selection.
    Private Sub btnRiemannLower_Click(sender As Object, e As EventArgs) Handles btnRiemannLower.Click
        'Label displaying the chosen technique in the form.
        frmTechnique.lblTechName.Text = "Right Riemann Sum"
        frmTechnique.txtTerms.Hide()
        frmTechnique.lblTerms.Hide()
```

```
            frmTechnique.lblSubinterval.Show()
            frmTechnique.txtSubinterval.Show()
            Me.Hide()
            frmTechnique.Show()
        End Sub

        'Button subrutine for the Midpoint Rule selection.
        Private Sub btnMidpoint_Click(sender As Object, e As EventArgs) Handles btnMidpoint.Click
            'Label displaying the chosen technique in the form.
            frmTechnique.lblTechName.Text = "Midpoint Rule"
            frmTechnique.txtTerms.Hide()
            frmTechnique.lblTerms.Hide()
            frmTechnique.lblSubinterval.Show()
            frmTechnique.txtSubinterval.Show()
            Me.Hide()
            frmTechnique.Show()

        End Sub

        'Button subrutine for the Trapezoidal Rule selection.
        Private Sub btnTrapezoidal_Click(sender As Object, e As EventArgs) Handles btnTrapezoidal.Click
            'Label displaying the chosen technique in the form.
            frmTechnique.lblTechName.Text = "Trapezoidal Rule"
            frmTechnique.txtTerms.Hide()
            frmTechnique.lblTerms.Hide()
            frmTechnique.lblSubinterval.Show()
            frmTechnique.txtSubinterval.Show()
            Me.Hide()
            frmTechnique.Show()
        End Sub

        'Button subrutine for the Simpson Rule selection.
        Private Sub btnSimpson_Click(sender As Object, e As EventArgs) Handles btnSimpson.Click
            'Label displaying the chosen technique in the form.
            frmTechnique.lblTechName.Text = "Simpson's Rule"
            frmTechnique.txtTerms.Hide()
            frmTechnique.lblTerms.Hide()
            frmTechnique.lblSubinterval.Show()
            frmTechnique.txtSubinterval.Show()
            Me.Hide()
            frmTechnique.Show()
```

```
        End Sub

        'Button subrutine for the Taylor polynomial selection.
        Private Sub btnSeries_Click(sender As Object, e As EventArgs) Handles btnTaylor.Click
            'Label displaying the chosen technique in the form.
            frmTechnique.lblTechName.Text = "Series Approximation"
            frmTechnique.lblSubinterval.Hide()
            frmTechnique.txtSubinterval.Hide()
            frmTechnique.txtTerms.Show()
            frmTechnique.lblTerms.Show()

            Me.Hide()
            frmTechnique.Show()
        End Sub

        'Button subrutine for multiple technique selection.
        Private Sub btnMultiple_Click(sender As Object, e As EventArgs) Handles btnMultiple.Click
            frmTechnique.lblTechName.Text = "All Techniques"
            frmTechnique.txtTerms.Show()
            frmTechnique.lblTerms.Show()
            frmTechnique.txtSubinterval.Show()
            frmTechnique.lblSubinterval.Show()

            Me.Hide()
            frmTechnique.Show()
        End Sub

        'Button subrutine for Boole's rule
        Private Sub btnBoole_Click(sender As Object, e As EventArgs) Handles btnBoole.Click
            frmTechnique.lblTechName.Text = "Boole's Rule"
            frmTechnique.txtTerms.Hide()
            frmTechnique.lblTerms.Hide()
            frmTechnique.txtSubinterval.Show()
            frmTechnique.lblSubinterval.Show()
            Me.Hide()
            frmTechnique.Show()
        End Sub

        'Button subrutine for the Simpson's 3/8 rule
        Private Sub btnThreeEight_Click(sender As Object, e As EventArgs) Handles btnThreeEight.Click
            frmTechnique.lblTechName.Text = "Simpson's 3/8 Rule"
```

```
        frmTechnique.txtTerms.Hide()
        frmTechnique.lblTerms.Hide()
        frmTechnique.txtSubinterval.Show()
        frmTechnique.lblSubinterval.Show()
        Me.Hide()
        frmTechnique.Show()
    End Sub

    'Button subrutine for the back form action
    Private Sub Button10_Click(sender As Object, e As EventArgs) Handles btnBack.Click
        Me.Hide()
        frmWelcome.Show()
    End Sub

    'Button method for the exit form action
    Private Sub btnExit_Click(sender As Object, e As EventArgs) Handles btnExit.Click
        Me.Close()
        Application.Exit()
    End Sub

End Class

************************************************************************************************************




'frmTechnique.vb Class is responsible For calling the numerical integration method selected by the user,
'passing the validated parameters To the respective numerical technique And displaying the results.

Public Class frmTechnique
    'Variable for displaying the fucntion's graph
    Dim graph As System.Drawing.Graphics
    'Variables related to the runtime of each technique
    Dim stopWatch As New Stopwatch()
    Dim ts As TimeSpan
    Dim elapsedTime As String

```

```
'Button subrutine for the calculate action which starts the process of finding the approximation of the
integral's value using the selected technique by the user.
    Private Sub btnCalculate_Click(sender As Object, e As EventArgs) Handles btnCalculate.Click

 'Declaration of several important variables needed for holding the parameters requiered by the Numerical
Integration Techniques (NITs)
        Dim upperBound, lowerBound, subintervalLength, midPointInterval, approximation, approxError,
        exactVal, derivatives(5), errorMax As Double

        Dim numberSubinterval, terms, subdivisions As Integer
        'Clearing the results textBox.
        txtResults.Clear()

        'Try-Catch statement for validating the user's entered parameters.
        Try
            'Converting the user's input data to numerical data
            upperBound = CDbl(txtUpperBound.Text)
            lowerBound = CDbl(txtLowerBound.Text)
            'Technique specific parameters
            If txtSubinterval.Visible = True Then
                numberSubinterval = CInt(txtSubinterval.Text)
                terms = 2
                subdivisions = 2
            ElseIf txtTerms.Visible = True Then
                terms = CInt(txtTerms.Text)
                numberSubinterval = 2
                subdivisions = 2
            End If
        Catch ex As Exception
            'If an error occurs, this message will pop up and alert the user that the default values will
be used.
            MessageBox.Show(ex.Message + vbNewLine + "Default values will be used.", "Input Error!",
MessageBoxButtons.OK, MessageBoxIcon.Warning)
            txtUpperBound.Text = "1"
            upperBound = 1
            txtLowerBound.Text = "-1"
```

```
                lowerBound = -1
                txtSubinterval.Text = "2"
                numberSubinterval = 2
                txtTerms.Text = "2"
                terms = 2
            End Try

            'Validating the integral bounds
            If upperBound <= lowerBound Or lowerBound >= upperBound Then
                MessageBox.Show("Wrong interval." + vbNewLine + "Default values will be used.", "Input Error!",
MessageBoxButtons.OK, MessageBoxIcon.Warning)
                txtUpperBound.Text = "1"
                upperBound = 1
                txtLowerBound.Text = "-1"
                lowerBound = -1
            End If

            'Validating the entered number of subintervals
            If numberSubinterval < 1 Then
                MessageBox.Show("The number of subintervals is too low." + vbNewLine + "Default value of 2 will
be used.", "Input Error!", MessageBoxButtons.OK, MessageBoxIcon.Warning)
                numberSubinterval = 2
                txtSubinterval.Text = "2"
            End If

            'Validating the entered number of terms for the series approximation
            If terms < 0 Then
                MessageBox.Show("The number of terms is too low." + vbNewLine + "Default value of 2 will be
used.", "Input Error!", MessageBoxButtons.OK, MessageBoxIcon.Warning)
                terms = 2
                txtTerms.Text = "2"
            End If


            'Calculating the length of each subinterval
            subintervalLength = ((upperBound - lowerBound) / numberSubinterval)
            'Calculating the midpoint of the interval.
            midPointInterval = (upperBound + lowerBound) / 2
```

```
******************************************************************************************************

'LEFT RIEMANN SUM DISPLAY OF RESULSTS


     If (String.Compare(lblTechName.Text, "Left Riemann Sum")) = 0 Then
            'Variable for holding the result of the technique. Calling the RiemannLeft function and passing
the requiered arguments.
            approximation = RiemannLeft(upperBound, lowerBound, subintervalLength, midPointInterval,
derivatives)
            'Variables to find the runtime of the technique.
            ts = stopWatch.Elapsed
            elapsedTime = String.Format("{0:0hr}:{1:0m}:{2:0s}.{3:0ms}", ts.Hours, ts.Minutes, ts.Seconds,
ts.Milliseconds / 10)


            'Displaying the technique data.
            txtResults.Text = "Fuction:" + vbTab + vbTab + vbTab + CboxFunctions.SelectedItem.ToString +
vbNewLine +
                              "Interval:" + vbTab + vbTab + vbTab + "[" + lowerBound.ToString + "," +
upperBound.ToString + "]" + vbNewLine +
                              "Subintervals:" + vbTab + vbTab + Strings.Format(numberSubinterval, "E") +
vbNewLine +
                              "Length (Δx):" + vbTab + vbTab + Strings.Format(subintervalLength, "E") +
vbNewLine +
                              "Approximation:" + vbTab + vbTab + Strings.Format(approximation, "E") +
vbNewLine +
                              "Runtime:" + vbTab + vbTab + vbTab + elapsedTime + vbNewLine + vbNewLine +
                              "First Derivative:" + vbNewLine +
                              "f '(" + lowerBound.ToString("N3") + ") : " + Strings.Format(derivatives(0),
"E") + vbNewLine +
                              "f '(" + midPointInterval.ToString("N3") + ") : " +
Strings.Format(derivatives(1), "E") + vbNewLine +
                              "f '(" + upperBound.ToString("N3") + ") : " + Strings.Format(derivatives(2),
"E") + vbNewLine + vbNewLine +
                              "Conclusions:" + vbNewLine + "This technique uses the left end point of each
subinterval to create a rectanglea under the curve. " + "If the function is increasing on the interval, " +
                              "then the left Riemann sum is an underestimate of the exact value and the
right Riemann sum is an overestimate." +
```

```
                              "If the function is decreasing on the interval, then the left sum
overestimates and the right sum underestimates the exact value." + vbNewLine + vbNewLine +
                              "Note: A higher quantity of subintervals gives a better approximation but
requieres more time."



            'RIGHT RIEMANN SUM DISPLAY OF RESULTS


        ElseIf (String.Compare(lblTechName.Text, "Right Riemann Sum")) = 0 Then
            'Variable for holding the result of the technique.
            approximation = RiemannRight(upperBound, lowerBound, subintervalLength, midPointInterval,
derivatives)
            'Variables needed to find the runtime of each technique.
            ts = stopWatch.Elapsed
            elapsedTime = String.Format("{0:0hr}:{1:0m}:{2:0s}.{3:0ms}", ts.Hours, ts.Minutes, ts.Seconds,
ts.Milliseconds / 10)

            'Displaying the technique data.
            txtResults.Text = "Fuction:" + vbTab + vbTab + vbTab + CboxFunctions.SelectedItem.ToString +
vbNewLine +
                              "Interval:" + vbTab + vbTab + vbTab + "[" + lowerBound.ToString + "," +
upperBound.ToString + "]" + vbNewLine +
                              "Subintervals:" + vbTab + vbTab + Strings.Format(numberSubinterval, "E") +
vbNewLine +
                              "Length (Δx):" + vbTab + vbTab + Strings.Format(subintervalLength, "E") +
vbNewLine +
                              "Approximation:" + vbTab + vbTab + Strings.Format(approximation, "E") +
vbNewLine +
                              "Runtime:" + vbTab + vbTab + vbTab + elapsedTime + vbNewLine + vbNewLine +
                              "First Derivative:" + vbNewLine +
                              "f '(" + lowerBound.ToString("N3") + ") : " + Strings.Format(derivatives(0),
"E") + vbNewLine +
                              "f '(" + midPointInterval.ToString("N3") + ") : " +
Strings.Format(derivatives(1), "E") + vbNewLine +
                              "f '(" + upperBound.ToString("N3") + ") : " + Strings.Format(derivatives(2),
"E") + vbNewLine + vbNewLine +
                              "Conclusions:" + vbNewLine + "This technique uses the right end point of each
subinterval to create a rectangle under the curve. " + "If the function is increasing on the interval, " +
```

```
                              "then the left Riemann sum is an underestimate of the exact value and the right Riemann sum is an overestimate." +
                              "If the function is decreasing on the interval, then the left sum overestimates and the right sum underestimates the exact value." + vbNewLine + vbNewLine +
                              "Note: A higher quantity of subintervals gives a better approximation but requieres more time."

            'MIDPOINT RULE DISPLAY OF RESULTS


        ElseIf (String.Compare(lblTechName.Text, "Midpoint Rule")) = 0 Then
            approximation = Midpoint(upperBound, lowerBound, subintervalLength, numberSubinterval, approxError, midPointInterval, exactVal)
            ts = stopWatch.Elapsed
            elapsedTime = String.Format("{0:0hr}:{1:0m}:{2:0s}.{3:0ms}", ts.Hours, ts.Minutes, ts.Seconds, ts.Milliseconds / 10)

            'Displaying the technique data.
            txtResults.Text = "Fuction:" + vbTab + vbTab + vbTab + CboxFunctions.SelectedItem.ToString + vbNewLine +
                              "Interval:" + vbTab + vbTab + vbTab + "[" + lowerBound.ToString + "," + upperBound.ToString + "]" + vbNewLine +
                              "Subintervals:" + vbTab + vbTab + Strings.Format(numberSubinterval, "E") + vbNewLine +
                              "Length (Δx):" + vbTab + vbTab + Strings.Format(subintervalLength, "E") + vbNewLine +
                              "Approximation:" + vbTab + vbTab + Strings.Format(approximation, "E") + vbNewLine +
                              "Error Max Bound:" + vbTab + vbTab + Strings.Format(approxError, "E") + vbNewLine +
                              "Possible Exact Value:" + vbTab + Strings.Format(exactVal, "E") + vbNewLine +
                              "Runtime:" + vbTab + vbTab + vbTab + elapsedTime + vbNewLine + vbNewLine +
                              "Conclusions:" + vbNewLine + "This technique uses the midpoint of each subinterval to create a rectangle under the curve. " +
                              "The approximation will be closer to the exact value in comparison to the other Riemann sums given that the midpoints are used. " +
                              "Therefore, the approximation of this technique will be in-between the left and right Riemann sums aproximations." + vbNewLine + vbNewLine +
                              "Note: A higher quantity of subintervals gives a better approximation but requieres more time."

```

```


            'TRAPEZOIDAL RULE DISPLAY OF RESULTS


        ElseIf (String.Compare(lblTechName.Text, "Trapezoidal Rule")) = 0 Then
            approximation = Trapezoidal(upperBound, lowerBound, subintervalLength, midPointInterval,
derivatives, approxError, exactVal, errorMax, numberSubinterval)
            ts = stopWatch.Elapsed
            elapsedTime = String.Format("{0:0hr}:{1:0m}:{2:0s}.{3:0ms}", ts.Hours, ts.Minutes, ts.Seconds,
ts.Milliseconds / 10)

            Dim absError As Double = Math.Abs(exactVal - approximation)
            Dim relError As Double = Math.Abs(absError / exactVal)
            Dim percError As Double = relError * 100

            'Displaying the technique data.
            txtResults.Text = "Fuction:" + vbTab + vbTab + vbTab + CboxFunctions.SelectedItem.ToString +
vbNewLine +
                              "Interval:" + vbTab + vbTab + vbTab + "[" + lowerBound.ToString + "," +
upperBound.ToString + "]" + vbNewLine +
                              "Subintervals:" + vbTab + vbTab + Strings.Format(numberSubinterval, "E") +
vbNewLine +
                              "Length (Δx):" + vbTab + vbTab + Strings.Format(subintervalLength, "E") +
vbNewLine +
                              "Approximation:" + vbTab + vbTab + Strings.Format(approximation, "E") +
vbNewLine +
                              "Error:" + vbTab + vbTab + vbTab + Strings.Format(approxError, "E") +
vbNewLine +
                              "Error Max Bound:" + vbTab + vbTab + Strings.Format(errorMax, "E") +
vbNewLine +
                              "Possible Exact Value:" + vbTab + Strings.Format(exactVal, "E") + vbNewLine +
                              "Relative Error:" + vbTab + vbTab + Strings.Format(relError, "E") + vbNewLine
+
                              "Error Percentage:" + vbTab + vbTab + percError.ToString("N3") + "%" +
vbNewLine +
                              "Runtime:" + vbTab + vbTab + vbTab + elapsedTime + vbNewLine + vbNewLine +
                              "Second Derivatives:" + vbNewLine +
                              "f ''(" + lowerBound.ToString("N3") + ") : " + Strings.Format(derivatives(0),
"E") + vbNewLine +
```

```
                              "f ''(" + midPointInterval.ToString("N3") + ") : " +
Strings.Format(derivatives(1), "E") + vbNewLine +
                              "f ''(" + upperBound.ToString("N3") + ") : " + Strings.Format(derivatives(2),
"E") + vbNewLine + vbNewLine +
                              "Conclusions:" + vbNewLine + "This technique uses trapezoids instead of
rectangles to approximate the area under a curve. " +
                              "It follows that if the integrand is concave up (and thus has a positive
second derivative), then the error is negative and the trapezoidal rule overestimates the true value. " +
                              "This can also be seen from the geometric picture: the trapezoids include all
of the area under the curve and extend over it. Similarly, a concave-down function yields an underestimate
" +
                              "because area is unaccounted For under the curve, but none Is counted above.
If the interval of the integral being approximated includes an inflection point, the Error Is harder To
identify." +
                               vbNewLine + vbNewLine + "Note: A higher quantity of subintervals gives a
better approximation but requieres more time. " +
                               "When the function is periodic and one integrates over one full period,
there are about as many sections of the graph that are concave up as concave down, so the errors cancel."

            'SIMPSON'S RULE DISPLAY OF RESULTS


        ElseIf (String.Compare(lblTechName.Text, "Simpson's Rule")) = 0 Then
            approximation = CompositeSimpson(upperBound, lowerBound, subintervalLength, numberSubinterval,
midPointInterval, approxError, exactVal, errorMax)
            ts = stopWatch.Elapsed
            elapsedTime = String.Format("{0:0hr}:{1:0m}:{2:0s}.{3:0ms}", ts.Hours, ts.Minutes, ts.Seconds,
ts.Milliseconds / 10)

            Dim absError As Double = Math.Abs(exactVal - approximation)
            Dim relError = Math.Abs(absError / exactVal)
            Dim percError = relError * 100

            'Displaying the technique data.
            txtResults.Text = "Fuction:" + vbTab + vbTab + vbTab + CboxFunctions.SelectedItem.ToString +
vbNewLine +
                              "Interval:" + vbTab + vbTab + vbTab + "[" + lowerBound.ToString + "," +
upperBound.ToString + "]" + vbNewLine +
                              "Subintervals:" + vbTab + vbTab + Strings.Format(numberSubinterval, "E") +
vbNewLine +
```

```
                              "Length (Δx):" + vbTab + vbTab + Strings.Format(subintervalLength, "E") +
vbNewLine +
                              "Approximation:" + vbTab + vbTab + Strings.Format(approximation, "E") +
vbNewLine +
                              "Error:" + vbTab + vbTab + vbTab + Strings.Format(approxError, "E") +
vbNewLine +
                              "Error Max Bound:" + vbTab + vbTab + Strings.Format(errorMax, "E") +
vbNewLine +
                              "Possible Exact Value:" + vbTab + Strings.Format(exactVal, "E") + vbNewLine +
                              "Relative Error:" + vbTab + vbTab + Strings.Format(relError, "E") + vbNewLine
+
                              "Error Percentage:" + vbTab + vbTab + percError.ToString("N3") + "%" +
vbNewLine +
                              "Runtime:" + vbTab + vbTab + vbTab + elapsedTime + vbNewLine + vbNewLine +
                              "Conclusions:" + vbNewLine + "This technique uses parabolas instead of
straight-line segments but the number of subintervals must be even. " +
                              "Simpson's rule provides exact results for any polynomial of degree three or
less, since the fourth derivative of such a polynomial is zero at all points." +
                              vbNewLine + vbNewLine + "Note: A higher quantity of subintervals gives a
better approximation but requieres more time."


            'SERIES APPROXIMATION TECHNIQUE DISPLAY OF RESULTS


        ElseIf (String.Compare(lblTechName.Text, "Series Approximation")) = 0 Then
            approximation = Series(upperBound, lowerBound, midPointInterval, terms, approxError, exactVal)
            ts = stopWatch.Elapsed
            elapsedTime = String.Format("{0:00hr}:{1:00m}:{2:00s}.{3:00ms}", ts.Hours, ts.Minutes,
ts.Seconds, ts.Milliseconds / 10)

            'Display the technique data.
            txtResults.Text = "Fuction:" + vbTab + vbTab + vbTab + CboxFunctions.SelectedItem.ToString +
vbNewLine +
                              "Interval:" + vbTab + vbTab + vbTab + "[" + lowerBound.ToString + "," +
upperBound.ToString + "]" + vbNewLine +
                              "Approximation:" + vbTab + vbTab + Strings.Format(approximation, "E") +
vbNewLine +
                              "Max Error Bound:" + vbTab + vbTab + Strings.Format(approxError, "E") +
vbNewLine +
                              "Possible Exact Value:" + vbTab + Strings.Format(exactVal, "E") + vbNewLine +
```

```
                              "Runtime:" + vbTab + vbTab + vbTab + elapsedTime + vbNewLine + vbNewLine +
                              "Conclusions:" + vbNewLine + vbNewLine + "Note: The taylor seires
approximation with " + terms.ToString + " terms."


            'BOOLE'S RULE DISPLAY OF RESULTS


        ElseIf (String.Compare(lblTechName.Text, "Boole's Rule")) = 0 Then
            approximation = Boole(lowerBound, upperBound, approxError, exactVal, midPointInterval,
numberSubinterval)
            ts = stopWatch.Elapsed
            elapsedTime = String.Format("{0:00hr}:{1:00m}:{2:00s}.{3:00ms}", ts.Hours, ts.Minutes,
ts.Seconds, ts.Milliseconds / 10)

            Dim absError As Double = Math.Abs(exactVal - approximation)
            Dim relError = Math.Abs(absError / exactVal)
            Dim percError = relError * 100

            'Display the technique data.
            txtResults.Text = "Fuction:" + vbTab + vbTab + vbTab + CboxFunctions.SelectedItem.ToString +
vbNewLine +
                              "Interval:" + vbTab + vbTab + vbTab + "[" + lowerBound.ToString + "," +
upperBound.ToString + "]" + vbNewLine +
                              "Approximation:" + vbTab + vbTab + Strings.Format(approximation, "E") +
vbNewLine +
                              "Approximation Error:" + vbTab + vbTab + Strings.Format(approxError, "E") +
vbNewLine +
                              "Possible Exact Value:" + vbTab + Strings.Format(exactVal, "E") + vbNewLine +
                              "Relative Error:" + vbTab + vbTab + Strings.Format(relError, "E") + vbNewLine
+
                              "Error Percentage:" + vbTab + vbTab + percError.ToString("N3") + "%" +
vbNewLine +
                              "Runtime:" + vbTab + vbTab + vbTab + elapsedTime + vbNewLine + vbNewLine +
                              "Conclusions:" + vbNewLine + vbNewLine + ""
```

```
        'SIMPSON'S COMPOSITE 3/8 RULE

    ElseIf (String.Compare(lblTechName.Text, "Simpson's 3/8 Rule")) = 0 Then
        approximation = CompositeThreeEight(lowerBound, upperBound, approxError, exactVal,
midPointInterval, numberSubinterval)
        ts = stopWatch.Elapsed
        elapsedTime = String.Format("{0:00hr}:{1:00m}:{2:00s}.{3:00ms}", ts.Hours, ts.Minutes,
ts.Seconds, ts.Milliseconds / 10)

        Dim absError As Double = Math.Abs(exactVal - approximation)
        Dim relError = Math.Abs(absError / exactVal)
        Dim percError = relError * 100

        'Display the technique data.
        txtResults.Text = "Fuction:" + vbTab + vbTab + vbTab + CboxFunctions.SelectedItem.ToString +
vbNewLine +
                          "Interval:" + vbTab + vbTab + vbTab + "[" + lowerBound.ToString + "," +
upperBound.ToString + "]" + vbNewLine +
                          "Approximation:" + vbTab + vbTab + Strings.Format(approximation, "E") +
vbNewLine +
                          "Approximation Error:" + vbTab + vbTab + Strings.Format(approxError, "E") +
vbNewLine +
                          "Possible Exact Value:" + vbTab + Strings.Format(exactVal, "E") + vbNewLine +
                          "Relative Error:" + vbTab + vbTab + Strings.Format(relError, "E") + vbNewLine
+
                          "Error Percentage:" + vbTab + vbTab + percError.ToString("N3") + "%" +
vbNewLine +
                          "Runtime:" + vbTab + vbTab + vbTab + elapsedTime + vbNewLine + vbNewLine +
                          "Conclusions:" + vbNewLine + vbNewLine + ""


        'RESULTS OF ALL THE AVAILABLE TECHNIQUES
    ElseIf (String.Compare(lblTechName.Text, "All Techniques")) = 0 Then

        'Display the technique data.
        txtResults.Text = "Fuction:" + vbTab + vbTab + vbTab + CboxFunctions.SelectedItem.ToString +
vbNewLine +
                          "Interval:" + vbTab + vbTab + vbTab + "[" + lowerBound.ToString + "," +
upperBound.ToString + "]" + vbNewLine + vbNewLine +
                          "Technique:" + vbTab + vbTab + "Approximation:" + vbNewLine + vbNewLine +
```

```
                              "Right Riemann Sum:" + vbTab + vbTab +
Strings.Format(RiemannRight(upperBound, lowerBound, subintervalLength, midPointInterval, derivatives), "E")
+ vbNewLine +
                              "Runtime:" + vbTab + vbTab + vbTab +
String.Format("{0:00hr}:{1:00m}:{2:00s}.{3:00ms}", stopWatch.Elapsed.Hours, stopWatch.Elapsed.Minutes,
stopWatch.Elapsed.Seconds, stopWatch.Elapsed.Milliseconds / 10) + vbNewLine + vbNewLine +
                              "Left Riemann Sum:" + vbTab + vbTab + Strings.Format(RiemannLeft(upperBound,
lowerBound, subintervalLength, midPointInterval, derivatives), "E") + vbNewLine +
                              "Runtime:" + vbTab + vbTab + vbTab +
String.Format("{0:00hr}:{1:00m}:{2:00s}.{3:00ms}", stopWatch.Elapsed.Hours, stopWatch.Elapsed.Minutes,
stopWatch.Elapsed.Seconds, stopWatch.Elapsed.Milliseconds / 10) + vbNewLine + vbNewLine +
                              "Midpoint Rule:" + vbTab + vbTab + Strings.Format(Midpoint(upperBound,
lowerBound, subintervalLength, numberSubinterval, approxError, midPointInterval, exactVal), "E") +
vbNewLine +
                              "Runtime:" + vbTab + vbTab + vbTab +
String.Format("{0:00hr}:{1:00m}:{2:00s}.{3:00ms}", stopWatch.Elapsed.Hours, stopWatch.Elapsed.Minutes,
stopWatch.Elapsed.Seconds, stopWatch.Elapsed.Milliseconds / 10) + vbNewLine + vbNewLine +
                              "Trapeziodal Rule:" + vbTab + vbTab + Strings.Format(Trapezoidal(upperBound,
lowerBound, subintervalLength, midPointInterval, derivatives, approxError, exactVal, errorMax,
numberSubinterval), "E") + vbNewLine +
                              "Runtime:" + vbTab + vbTab + vbTab +
String.Format("{0:00hr}:{1:00m}:{2:00s}.{3:00ms}", stopWatch.Elapsed.Hours, stopWatch.Elapsed.Minutes,
stopWatch.Elapsed.Seconds, stopWatch.Elapsed.Milliseconds / 10) + vbNewLine + vbNewLine +
                              "Simpson's Rule:" + vbTab + vbTab +
Strings.Format(CompositeSimpson(upperBound, lowerBound, subintervalLength, numberSubinterval,
midPointInterval, approxError, exactVal, errorMax), "E") + vbNewLine +
                              "Runtime:" + vbTab + vbTab + vbTab +
String.Format("{0:00hr}:{1:00m}:{2:00s}.{3:00ms}", stopWatch.Elapsed.Hours, stopWatch.Elapsed.Minutes,
stopWatch.Elapsed.Seconds, stopWatch.Elapsed.Milliseconds / 10) + vbNewLine + vbNewLine +
                              "Simpson's 3/8 Rule:" + vbTab + vbTab +
Strings.Format(CompositeThreeEight(lowerBound, upperBound, approxError, exactVal, midPointInterval,
numberSubinterval), "E") + vbNewLine +
                              "Runtime:" + vbTab + vbTab + vbTab +
String.Format("{0:00hr}:{1:00m}:{2:00s}.{3:00ms}", stopWatch.Elapsed.Hours, stopWatch.Elapsed.Minutes,
stopWatch.Elapsed.Seconds, stopWatch.Elapsed.Milliseconds / 10) + vbNewLine + vbNewLine +
                              "Series Approximation:" + vbTab + Strings.Format(Series(upperBound,
lowerBound, midPointInterval, terms, approxError, exactVal), "E") + vbNewLine +
                              "Runtime:" + vbTab + vbTab + vbTab +
String.Format("{0:00hr}:{1:00m}:{2:00s}.{3:00ms}", stopWatch.Elapsed.Hours, stopWatch.Elapsed.Minutes,
stopWatch.Elapsed.Seconds, stopWatch.Elapsed.Milliseconds / 10) + vbNewLine + vbNewLine +
```

```
                              "Boole's Rule:" + vbTab + vbTab + Strings.Format(Boole(lowerBound,
upperBound, approxError, exactVal, midPointInterval, numberSubinterval), "E") + vbNewLine +
                              "Runtime:" + vbTab + vbTab + vbTab +
String.Format("{0:00hr}:{1:00m}:{2:00s}.{3:00ms}", stopWatch.Elapsed.Hours, stopWatch.Elapsed.Minutes,
stopWatch.Elapsed.Seconds, stopWatch.Elapsed.Milliseconds / 10) + vbNewLine + vbNewLine

        End If
    End Sub

**********************************************************************************************************

'Right Riemann Sum technique
    'This function calculates the approximation of the integral's value using the right Riemann sum
technique.
    Public Function RiemannRight(UpperBound As Double, LowerBound As Double, SubintervalLength As Double,
midPointInterval As Double, derivatives() As Double) As Double
        'Variable for holding the approximation.
        Dim Sum As Double = 0
        'Variable for holding the function
        Dim func As Func(Of Double, Double)
        'Resseting the stopwatch everytime the technique is used.
        stopWatch.Reset()

        'If - ElseIf statements that determine the appropriate formula to use depending on the function
selected by the user.
        If CboxFunctions.SelectedIndex = 0 Then
            'Start the stopwatch to determine runtime.
            stopWatch.Start()
            'For cycle from the integral's lower bound limit to the upper bound limit with a step of the
subinterval length value.
            For x As Double = LowerBound + Math.Abs(SubintervalLength) To UpperBound Step
Math.Abs(SubintervalLength)
                'Accumulate the right Riemann sum term values to the variable sum in order to determine the
approximation.
                Sum += Math.Pow(x, 2) * SubintervalLength
            Next
            'Stop the stopwatch and save the elapsed time.
            stopWatch.Stop()
            'Store the user's selected fuction to calculate the derivatives
            func = Function(x) Math.Pow(x, 2)
```

```
            'Calculating the first derivatives and storing them in the corresponding variables.
            ' Using Math.Net Numerics Library
            derivatives(0) = MathNet.Numerics.Differentiate.Derivative(func, UpperBound, 1)
            derivatives(1) = MathNet.Numerics.Differentiate.Derivative(func, LowerBound, 1)
            derivatives(2) = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 1)

            'User's selected function
        ElseIf CboxFunctions.SelectedIndex = 1 Then
            stopWatch.Start()
            For x As Double = LowerBound + Math.Abs(SubintervalLength) To UpperBound Step
Math.Abs(SubintervalLength)
                Sum += (Math.Pow(x, 2) + 1) * SubintervalLength
            Next
            stopWatch.Stop()

            func = Function(x) Math.Pow(x, 2) + 1
            derivatives(0) = MathNet.Numerics.Differentiate.Derivative(func, UpperBound, 1)
            derivatives(1) = MathNet.Numerics.Differentiate.Derivative(func, LowerBound, 1)
            derivatives(2) = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 1)

        ElseIf CboxFunctions.SelectedIndex = 2 Then
            stopWatch.Start()
            For x As Double = LowerBound + Math.Abs(SubintervalLength) To UpperBound Step
Math.Abs(SubintervalLength)
                Sum += Math.Sin(Math.Pow(x, 2)) * SubintervalLength
            Next
            stopWatch.Stop()

            func = Function(x) Math.Sin(Math.Pow(x, 2))
            derivatives(0) = MathNet.Numerics.Differentiate.Derivative(func, UpperBound, 1)
            derivatives(1) = MathNet.Numerics.Differentiate.Derivative(func, LowerBound, 1)
            derivatives(2) = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 1)

        ElseIf CboxFunctions.SelectedIndex = 3 Then
            stopWatch.Start()
            For x As Double = LowerBound + Math.Abs(SubintervalLength) To UpperBound Step
Math.Abs(SubintervalLength)
                Sum += Math.Exp(Math.Pow(x, 2)) * SubintervalLength
            Next
            stopWatch.Stop()

```

```
            func = Function(x) Math.Exp(Math.Pow(x, 2))
            derivatives(0) = MathNet.Numerics.Differentiate.Derivative(func, UpperBound, 1)
            derivatives(1) = MathNet.Numerics.Differentiate.Derivative(func, LowerBound, 1)
            derivatives(2) = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 1)

        End If

        Return Sum
    End Function
```

```

'Left Riemann Sum technique
    'This function calculates the approximation of the integral's value using the left Riemann sum
technique.
    Public Function RiemannLeft(UpperBound As Double, LowerBound As Double, SubintervalLength As Double,
midPointInterval As Double, derivatives() As Double) As Double
        Dim Sum As Double = 0
        Dim func As Func(Of Double, Double)
        stopWatch.Reset()

        'The only diffeerence between this technique and the right Riemann sum is that this one takes the
left subinterval value
        If CboxFunctions.SelectedIndex = 0 Then
            stopWatch.Start()
            'For cycle from the lowerBound to the last subinterval left side point.
            'Increments are the subinterval length magnitude.
            For x As Double = LowerBound To UpperBound - Math.Abs(SubintervalLength) Step
Math.Abs(SubintervalLength)
                Sum += Math.Pow(x, 2) * SubintervalLength
            Next
            stopWatch.Stop()
            'Calculating derivatives
            func = Function(x) Math.Pow(x, 2)
            derivatives(0) = MathNet.Numerics.Differentiate.Derivative(func, UpperBound, 1)
            derivatives(1) = MathNet.Numerics.Differentiate.Derivative(func, LowerBound, 1)
            derivatives(2) = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 1)

        ElseIf CboxFunctions.SelectedIndex = 1 Then
```

```
            stopWatch.Start()
            For x As Double = LowerBound To UpperBound - Math.Abs(SubintervalLength) Step
Math.Abs(SubintervalLength)
                Sum += (Math.Pow(x, 2) + 1) * SubintervalLength
            Next
            stopWatch.Stop()

            func = Function(x) Math.Pow(x, 2) + 1
            derivatives(0) = MathNet.Numerics.Differentiate.Derivative(func, UpperBound, 1)
            derivatives(1) = MathNet.Numerics.Differentiate.Derivative(func, LowerBound, 1)
            derivatives(2) = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 1)

        ElseIf CboxFunctions.SelectedIndex = 2 Then
            stopWatch.Start()
            For x As Double = LowerBound To UpperBound - Math.Abs(SubintervalLength) Step
Math.Abs(SubintervalLength)
                Sum += Math.Sin(Math.Pow(x, 2)) * SubintervalLength
            Next
            stopWatch.Stop()

            func = Function(x) Math.Sin(Math.Pow(x, 2))
            derivatives(0) = MathNet.Numerics.Differentiate.Derivative(func, UpperBound, 1)
            derivatives(1) = MathNet.Numerics.Differentiate.Derivative(func, LowerBound, 1)
            derivatives(2) = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 1)

        ElseIf CboxFunctions.SelectedIndex = 3 Then
            stopWatch.Start()
            For x As Double = LowerBound To UpperBound - Math.Abs(SubintervalLength) Step
Math.Abs(SubintervalLength)
                Sum += Math.Exp(Math.Pow(x, 2)) * SubintervalLength
            Next
            stopWatch.Stop()

            func = Function(x) Math.Exp(Math.Pow(x, 2))
            derivatives(0) = MathNet.Numerics.Differentiate.Derivative(func, UpperBound, 1)
            derivatives(1) = MathNet.Numerics.Differentiate.Derivative(func, LowerBound, 1)
            derivatives(2) = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 1)

        End If
        Return Sum
    End Function
```

```
'Midpoint rule technique
    'This function calculates the approximation of the integral's value using the midpoint ruel technique.
    Public Function Midpoint(UpperBound As Double, LowerBound As Double, SubintervalLength As Double, numberSubinterval As Double, ByRef approxError As Double, midPointInterval As Double, ByRef exactVal As Double) As Double
        Dim Sum As Double = 0
        Dim func As Func(Of Double, Double)
        'Variables for holding the midpoint of each subinterval and the second derivative evaluated at the midPoint of the interval
        Dim subIntervalMid, secondDerivative As Double
        stopWatch.Reset()

        stopWatch.Start()
        If CboxFunctions.SelectedIndex = 0 Then
            'This technique requieres to find the subinterval midpoint of each subinterval and evaluate the function at that point.
            For x As Double = LowerBound To UpperBound - Math.Abs(SubintervalLength) Step Math.Abs(SubintervalLength)
                subIntervalMid = (x + (x + SubintervalLength)) / 2
                Sum += Math.Pow(subIntervalMid, 2) * SubintervalLength
            Next

            'Calculating derivatives and error utilizing mathematical formula.
            func = Function(x) Math.Pow(x, 2)
            secondDerivative = Math.Abs(MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 2))
            approxError = Math.Abs(secondDerivative * (Math.Pow(UpperBound - LowerBound, 3)) / (24 * Math.Pow(numberSubinterval, 2)))
            exactVal = Sum - approxError

        ElseIf CboxFunctions.SelectedIndex = 1 Then
            For x As Double = LowerBound To UpperBound - Math.Abs(SubintervalLength) Step Math.Abs(SubintervalLength)
                subIntervalMid = (x + (x + SubintervalLength)) / 2
                Sum += (Math.Pow(subIntervalMid, 2) + 1) * SubintervalLength
            Next
```

```


                func = Function(x) Math.Pow(x, 2) + 1
                secondDerivative = Math.Abs(MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 2))
                approxError = Math.Abs(secondDerivative * (Math.Pow(UpperBound - LowerBound, 3)) / (24 * Math.Pow(numberSubinterval, 2)))
                exactVal = Sum - approxError

            ElseIf CboxFunctions.SelectedIndex = 2 Then
                For x As Double = LowerBound To UpperBound - Math.Abs(SubintervalLength) Step Math.Abs(SubintervalLength)
                    subIntervalMid = (x + (x + SubintervalLength)) / 2
                    Sum += Math.Sin(Math.Pow(subIntervalMid, 2)) * SubintervalLength
                Next

                func = Function(x) Math.Sin(Math.Pow(x, 2))
                secondDerivative = Math.Abs(MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 2))
                approxError = Math.Abs(secondDerivative * (Math.Pow(UpperBound - LowerBound, 3)) / (24 * Math.Pow(numberSubinterval, 2)))
                exactVal = Sum - approxError

            ElseIf CboxFunctions.SelectedIndex = 3 Then
                For x As Double = LowerBound To UpperBound - Math.Abs(SubintervalLength) Step Math.Abs(SubintervalLength)
                    subIntervalMid = (x + (x + SubintervalLength)) / 2
                    Sum += Math.Exp(Math.Pow(x, 2)) * SubintervalLength
                Next

                func = Function(x) Math.Exp(Math.Pow(x, 2))
                secondDerivative = Math.Abs(MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 2))
                approxError = Math.Abs(secondDerivative * (Math.Pow(UpperBound - LowerBound, 3)) / (24 * Math.Pow(numberSubinterval, 2)))
                exactVal = Sum - approxError

            End If
            stopWatch.Stop()
            Return Sum
        End Function
```

```
'Multiple Segment / Composite Trapezoidal rule technique (n=1 for elementary formula having one subinterval)
    'This function calculates the approximation of the integral's value using the trapezium rule technique.
    Public Function Trapezoidal(UpperBound As Double, LowerBound As Double, SubintervalLength As Double, midPointInterval As Double, ByRef derivatives() As Double, ByRef approxError As Double, ByRef exactVal As Double,
                                ByRef errorMax As Double, numberSubinterval As Integer) As Double

        Dim TApprox As Double = 0
        Dim func As Func(Of Double, Double)
        stopWatch.Reset()

        stopWatch.Start()
        'If selected function has index 0 then do this.
        If CboxFunctions.SelectedIndex = 0 Then
            'From lower bound to upperbound with step given by the length of each subinterval
            For x As Double = LowerBound To UpperBound Step Math.Abs(SubintervalLength)
                'The mathematical formula states that the first and last evaluations are not multiplied by two
                If x = LowerBound Or x = UpperBound Then
                    TApprox += Math.Pow(x, 2) * SubintervalLength / 2
                Else
                    TApprox += (2 * Math.Pow(x, 2) * (SubintervalLength / 2))
                End If
            Next

            'Calculating derivatives and error, estimating an exact value using mathematical formula.
            func = Function(x) Math.Pow(x, 2)
            derivatives(0) = MathNet.Numerics.Differentiate.Derivative(func, UpperBound, 2)
            derivatives(1) = MathNet.Numerics.Differentiate.Derivative(func, LowerBound, 2)
            derivatives(2) = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 2)
            'Error computation given by mathematical formula.
            'The exact value is not exact, it is the possible exact value and it is used to calculate the relative error.
            'Errormax holds the max bound of the error.
```

```
                approxError = Math.Abs(((UpperBound - LowerBound) / 12) * derivatives(2) *
Math.Pow(SubintervalLength, 2))
                errorMax = derivatives(2) * (Math.Pow(UpperBound - LowerBound, 3) / 12 *
Math.Pow(numberSubinterval, 2))
                exactVal = TApprox - approxError


            ElseIf CboxFunctions.SelectedIndex = 1 Then
                For x As Double = LowerBound To UpperBound Step Math.Abs(SubintervalLength)
                    If x = LowerBound Or x = UpperBound Then
                        TApprox += (Math.Pow(x, 2) + 1) * SubintervalLength / 2
                    Else
                        TApprox += (2 * (Math.Pow(x, 2) + 1) * (SubintervalLength / 2))
                    End If
                Next

                func = Function(x) Math.Pow(x, 2) + 1
                derivatives(0) = MathNet.Numerics.Differentiate.Derivative(func, UpperBound, 2)
                derivatives(1) = MathNet.Numerics.Differentiate.Derivative(func, LowerBound, 2)
                derivatives(2) = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 2)
                approxError = Math.Abs(((UpperBound - LowerBound) / 12) * derivatives(2) *
Math.Pow(SubintervalLength, 2))
                errorMax = derivatives(2) * (Math.Pow(UpperBound - LowerBound, 3) / 12 *
Math.Pow(numberSubinterval, 2))
                exactVal = TApprox - approxError

            ElseIf CboxFunctions.SelectedIndex = 2 Then

                For x As Double = LowerBound To UpperBound Step Math.Abs(SubintervalLength)
                    If x = LowerBound Or x = UpperBound Then
                        TApprox += Math.Sin(Math.Pow(x, 2)) * SubintervalLength / 2
                    Else
                        TApprox += ((2 * Math.Sin(Math.Pow(x, 2))) * (SubintervalLength / 2))
                    End If
                Next

                func = Function(x) Math.Sin(Math.Pow(x, 2))
                derivatives(0) = MathNet.Numerics.Differentiate.Derivative(func, UpperBound, 2)
                derivatives(1) = MathNet.Numerics.Differentiate.Derivative(func, LowerBound, 2)
                derivatives(2) = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 2)
```

```
            approxError = Math.Abs(((UpperBound - LowerBound) / 12) * derivatives(2) *
Math.Pow(SubintervalLength, 2))
            errorMax = derivatives(2) * (Math.Pow(UpperBound - LowerBound, 3) / 12 *
Math.Pow(numberSubinterval, 2))
            exactVal = TApprox - approxError

        ElseIf CboxFunctions.SelectedIndex = 3 Then

            For x As Double = LowerBound To UpperBound Step Math.Abs(SubintervalLength)
                If x = LowerBound Or x = UpperBound Then
                    TApprox += Math.Exp(Math.Pow(x, 2)) * SubintervalLength / 2
                Else
                    TApprox += ((2 * Math.Exp(Math.Pow(x, 2))) * (SubintervalLength / 2))
                End If
            Next

            func = Function(x) Math.Exp(Math.Pow(x, 2))
            derivatives(0) = MathNet.Numerics.Differentiate.Derivative(func, UpperBound, 2)
            derivatives(1) = MathNet.Numerics.Differentiate.Derivative(func, LowerBound, 2)
            derivatives(2) = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 2)
            approxError = Math.Abs(((UpperBound - LowerBound) / 12) * derivatives(2) *
Math.Pow(SubintervalLength, 2))
            errorMax = derivatives(2) * (Math.Pow(UpperBound - LowerBound, 3) / 12 *
Math.Pow(numberSubinterval, 2))
            exactVal = TApprox - approxError

        End If

        stopWatch.Stop()
        Return TApprox
    End Function
```

```
▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪

'Multiple Segment / Composite Simpson's rule technnique (n=2 for 1/3 rule)
    'This function calculates the approximation of the integral's value using the Simpson's rule technique.
    Public Function CompositeSimpson(UpperBound As Double, LowerBound As Double, SubintervalLength As
Double, numberSubinterval As Integer, midPointInterval As Double, ByRef approxError As Double,
                            ByRef exactVal As Double, ByRef errorMax As Double) As Double

        'The counter variable is used for knowing when we are at the lowerbound and upperbound
        Dim counter As Integer = 0
        Dim func As Func(Of Double, Double)
        Dim fourthDerivative As Double
        Dim SApprox As Double = 0
        stopWatch.Reset()

        stopWatch.Start()
        'Selection of function
        If CboxFunctions.SelectedIndex = 0 Then
            'Calculating the approximation using the mathematical formula
            For x As Double = LowerBound To UpperBound Step Math.Abs(SubintervalLength)
                If counter = 0 Or counter = numberSubinterval Then
                    SApprox += Math.Pow(x, 2) * (SubintervalLength / 3)
                ElseIf counter Mod 2 = 0 Then
                    SApprox += 2 * Math.Pow(x, 2) * (SubintervalLength / 3)
                Else
                    SApprox += 4 * Math.Pow(x, 2) * (SubintervalLength / 3)
                End If
                counter += 1
            Next

            'Calculating derivatives needed for error analysis
            func = Function(x) Math.Pow(x, 2)
            fourthDerivative = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 4)
            'Error computation
            approxError = Math.Abs(((UpperBound - LowerBound) / 180) * fourthDerivative *
Math.Pow(SubintervalLength, 4))
            errorMax = fourthDerivative * (Math.Pow(UpperBound - LowerBound, 5) / 180 *
Math.Pow(numberSubinterval, 4))
```

```
                exactVal = SApprox - approxError

            ElseIf CboxFunctions.SelectedIndex = 1 Then
                For x As Double = LowerBound To UpperBound Step Math.Abs(SubintervalLength)
                    If counter = 0 Or counter = numberSubinterval Then
                        SApprox += (Math.Pow(x, 2) + 1) * (SubintervalLength / 3)
                    ElseIf counter Mod 2 = 0 Then
                        SApprox += 2 * (Math.Pow(x, 2) + 1) * (SubintervalLength / 3)
                    Else
                        SApprox += 4 * (Math.Pow(x, 2) + 1) * (SubintervalLength / 3)
                    End If
                    counter += 1
                Next

                func = Function(x) Math.Pow(x, 2) + 1
                fourthDerivative = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 4)
                approxError = Math.Abs(((UpperBound - LowerBound) / 180) * fourthDerivative *
Math.Pow(SubintervalLength, 4))
                errorMax = fourthDerivative * (Math.Pow(UpperBound - LowerBound, 5) / 180 *
Math.Pow(numberSubinterval, 4))
                exactVal = SApprox - approxError

            ElseIf CboxFunctions.SelectedIndex = 2 Then
                For x As Double = LowerBound To UpperBound Step Math.Abs(SubintervalLength)
                    If counter = 0 Or counter = numberSubinterval Then
                        SApprox += Math.Sin(Math.Pow(x, 2)) * (SubintervalLength / 3)
                    ElseIf counter Mod 2 = 0 Then
                        SApprox += 2 * Math.Sin(Math.Pow(x, 2)) * (SubintervalLength / 3)
                    Else
                        SApprox += 4 * Math.Sin(Math.Pow(x, 2)) * (SubintervalLength / 3)
                    End If
                    counter += 1
                Next

                func = Function(x) Math.Sin(Math.Pow(x, 2))
                fourthDerivative = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 4)
                approxError = Math.Abs(((UpperBound - LowerBound) / 180) * fourthDerivative *
Math.Pow(SubintervalLength, 4))
                errorMax = fourthDerivative * (Math.Pow(UpperBound - LowerBound, 5) / 180 *
Math.Pow(numberSubinterval, 4))
                exactVal = SApprox - approxError
```

```
        ElseIf CboxFunctions.SelectedIndex = 3 Then
            For x As Double = LowerBound To UpperBound Step Math.Abs(SubintervalLength)
                If counter = 0 Or counter = numberSubinterval Then
                    SApprox += Math.Exp(Math.Pow(x, 2)) * (SubintervalLength / 3)
                ElseIf counter Mod 2 = 0 Then
                    SApprox += 2 * Math.Exp(Math.Pow(x, 2)) * (SubintervalLength / 3)
                Else
                    SApprox += 4 * Math.Exp(Math.Pow(x, 2)) * (SubintervalLength / 3)
                End If
                counter += 1
            Next

            func = Function(x) Math.Exp(Math.Pow(x, 2))
            fourthDerivative = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 4)
            approxError = Math.Abs(((UpperBound - LowerBound) / 180) * fourthDerivative *
Math.Pow(SubintervalLength, 4))
            errorMax = fourthDerivative * (Math.Pow(UpperBound - LowerBound, 5) / 180 *
Math.Pow(numberSubinterval, 4))
            exactVal = SApprox - approxError

        End If

        stopWatch.Stop()
        Return SApprox
    End Function

****************************************************************************************************
```

```
'Taylor series approximation using term by term integration
    'The terms are specified by the user and the approximation is found using term by term integration.
    Public Function Series(UpperBound As Double, LowerBound As Double, midPointInterval As Double, terms As
Integer, ByRef approxError As Double, ByRef exactVal As Double) As Double
        'Array for holding the values of the terms once they have been integrated
        Dim sum(terms) As Double
        Dim approximation As Double = 0
        Dim func As Func(Of Double, Double)
        stopWatch.Reset()

        stopWatch.Start()
        If CboxFunctions.SelectedIndex = 0 Then
            'func holds the function
            func = Function(x) Math.Pow(x, 2)
            'This line cannot be inside the for cycle becuase the Math.Net Numerics Derivative method
starts calculating derivatives of order 1
            approximation = MathNet.Numerics.Integrate.OnClosedInterval(Function(x)
(Math.Pow(midPointInterval, 2) * Math.Pow(x - midPointInterval, 0)) / factorial(0), LowerBound, UpperBound)

            For n As Integer = 1 To terms - 1 Step 1
                'Calculate the derivative, evaluate the polynomial term in the midpoint of the interval,
and then integrate the term.
                sum(n) = MathNet.Numerics.Integrate.OnClosedInterval(Function(x)
(MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, n) * Math.Pow(x - midPointInterval, n))
/ factorial(n), LowerBound, UpperBound)
                'Holds the sum of integrated terms. Yields the approximation of the antiderivative value
                approximation += sum(n)
            Next

            'Reminder theorem error
            approxError = Math.Abs((Math.Pow(UpperBound, 2) * Math.Pow(midPointInterval - LowerBound, terms
+ 1)) / factorial(terms + 1))
            exactVal = approximation - approxError


        ElseIf CboxFunctions.SelectedIndex = 1 Then
            func = Function(x) Math.Pow(x, 2) + 1
```

```
            approximation = MathNet.Numerics.Integrate.OnClosedInterval(Function(x)
((Math.Pow(midPointInterval, 2) + 1) * Math.Pow(x - midPointInterval, 0)) / factorial(0), LowerBound,
UpperBound)

            For n As Integer = 1 To terms - 1 Step 1
                sum(n) = MathNet.Numerics.Integrate.OnClosedInterval(Function(x)
(MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, n) * Math.Pow(x - midPointInterval, n))
/ factorial(n), LowerBound, UpperBound)
                approximation += sum(n)
            Next

            approxError = Math.Abs((Math.Pow(UpperBound, 2) + 1 * Math.Pow(midPointInterval - LowerBound,
terms + 1)) / factorial(terms + 1))
            exactVal = approximation - approxError

        ElseIf CboxFunctions.SelectedIndex = 2 Then
            For n As Integer = 0 To terms Step 1
                'Using the series expansion for sin(x) and term by term integration
                sum(n) = MathNet.Numerics.Integrate.OnClosedInterval(Function(x) (Math.Pow(-1, n) *
Math.Pow(x, 4 * n + 2)) / factorial(2 * n + 1), LowerBound, UpperBound)
                approximation += sum(n)
            Next

            approxError = Math.Abs((Math.Sin(Math.Pow(UpperBound, 2)) * Math.Pow(midPointInterval -
LowerBound, terms + 1)) / factorial(terms + 1))
            exactVal = approximation - approxError

        ElseIf CboxFunctions.SelectedIndex = 3 Then
            For n As Integer = 0 To terms Step 1
                'Using the series expansion for e^x and term by term integration
                sum(n) = MathNet.Numerics.Integrate.OnClosedInterval(Function(x) (Math.Pow(x, 2 * n)) /
factorial(n), LowerBound, UpperBound)
                approximation += sum(n)
            Next

            approxError = Math.Abs((Math.Exp(Math.Pow(UpperBound, 2)) * Math.Pow(midPointInterval -
LowerBound, terms + 1)) / factorial(terms + 1))
            exactVal = approximation - approxError

        End If

```

```
            Return approximation
        End Function


    'Composite Boole's Rule
        Public Function Boole(lowerBound As Double, upperBound As Double, ByRef approxError As Double, ByRef exactVal As Double, midPointInterval As Double, numberSubintervals As Integer) As Double
            'Calculating the length of the subintervals
            Dim h As Double = (upperBound - lowerBound) / (4 * numberSubintervals)
            'array for holding the values of x sub k. The array is of size 4*number of subintervals because the mathematical formula
            Dim x(4 * numberSubintervals) As Double
            Dim func As Func(Of Double, Double)
            Dim sixthDerivative As Double
            Dim bApprox As Double = 0
            stopWatch.Reset()

            stopWatch.Start()
            'Calculating the subinterval points where the function will be evaluated
            For j As Integer = 0 To 4 * numberSubintervals Step 1
                x(j) = lowerBound + j * h
            Next

            If CboxFunctions.SelectedIndex = 0 Then
                For j As Integer = 1 To numberSubintervals Step 1
                    'Formula is created from the mathematical theory, selecting 5 points and evaluating them.
                    bApprox += (2 * h / 45) * (7 * Math.Pow(x(4 * j - 4), 2) + 32 * Math.Pow(x(4 * j - 3), 2) + 12 * Math.Pow(x(4 * j - 2), 2) + 32 * Math.Pow(x(4 * j - 1), 2) + 7 * Math.Pow(x(4 * j), 2))
                Next
                'storing the function to calculate its derivative
                func = Function(y) Math.Pow(y, 2)
                sixthDerivative = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 6)
                'Error computation
                approxError = Math.Abs(((2 * (upperBound - lowerBound) * sixthDerivative) / 945) * Math.Pow(h, 6))
                exactVal = bApprox - approxError

            ElseIf CboxFunctions.SelectedIndex = 1 Then
```

```
            For j As Integer = 1 To numberSubintervals Step 1
                bApprox += (2 * h / 45) * (7 * (Math.Pow(x(4 * j - 4), 2) + 1) + 32 * (Math.Pow(x(4 * j - 3), 2) + 1) + 12 * (Math.Pow(x(4 * j - 2), 2) + 1) + 32 * (Math.Pow(x(4 * j - 1), 2) + 1) + 7 * (Math.Pow(x(4 * j), 2)) + 1)
            Next
            func = Function(y) Math.Pow(y, 2) + 1
            sixthDerivative = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 6)
            approxError = Math.Abs(((2 * (upperBound - lowerBound) * sixthDerivative) / 945) * Math.Pow(h, 6))
            exactVal = bApprox - approxError

        ElseIf CboxFunctions.SelectedIndex = 2 Then
            For j As Integer = 1 To numberSubintervals Step 1
                bApprox += (2 * h / 45) * (7 * Math.Sin(Math.Pow(x(4 * j - 4), 2)) + 32 * Math.Sin(Math.Pow(x(4 * j - 3), 2)) + 12 * Math.Sin(Math.Pow(x(4 * j - 2), 2)) + 32 * Math.Sin(Math.Pow(x(4 * j - 1), 2)) + 7 * Math.Sin(Math.Pow(x(4 * j), 2)))
            Next
            func = Function(y) Math.Sin(Math.Pow(y, 2))
            sixthDerivative = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 6)
            approxError = Math.Abs(((2 * (upperBound - lowerBound) * sixthDerivative) / 945) * Math.Pow(h, 6))
            exactVal = bApprox - approxError

        ElseIf CboxFunctions.SelectedIndex = 3 Then
            For j As Integer = 1 To numberSubintervals Step 1
                bApprox += (2 * h / 45) * (7 * Math.Exp(Math.Pow(x(4 * j - 4), 2)) + 32 * Math.Exp(Math.Pow(x(4 * j - 3), 2)) + 12 * Math.Exp(Math.Pow(x(4 * j - 2), 2)) + 32 * Math.Exp(Math.Pow(x(4 * j - 1), 2)) + 7 * Math.Exp(Math.Pow(x(4 * j), 2)))
            Next
            func = Function(y) Math.Exp(Math.Pow(y, 2))
            sixthDerivative = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 6)
            approxError = Math.Abs(((2 * (upperBound - lowerBound) * sixthDerivative) / 945) * Math.Pow(h, 6))
            exactVal = bApprox - approxError

        End If

        stopWatch.Stop()
        Return bApprox
    End Function
```

```
*************************************************************************************************

'Composite Simpson's 3/8 Rule
    Public Function CompositeThreeEight(lowerBound As Double, upperBound As Double, ByRef approxError As
Double, ByRef exactVal As Double, midPointInterval As Double, numberSubintervals As Integer) As Double
        'Similar to Boole's rule using four points instead of five.
        'h holds the subinterval length
        Dim h As Double = (upperBound - lowerBound) / (3 * numberSubintervals)
        'Array for holding the values of x
        Dim x(3 * numberSubintervals) As Double
        Dim func As Func(Of Double, Double)
        Dim fourthDerivative As Double
        Dim Approx As Double = 0
        stopWatch.Reset()

        stopWatch.Start()
        'Calculating the x values on the subintervals that the function will be evaluated on
        For j As Integer = 0 To 3 * numberSubintervals Step 1
            x(j) = lowerBound + j * h
        Next

        If CboxFunctions.SelectedIndex = 0 Then
            For j As Integer = 1 To numberSubintervals Step 1
                'Translated from the mathematical sumation formula
                Approx += (3 * h / 8) * (Math.Pow(x(3 * j - 3), 2) + 3 * Math.Pow(x(3 * j - 2), 2) + 3 *
Math.Pow(x(3 * j - 1), 2) + Math.Pow(x(3 * j), 2))
            Next
            func = Function(y) Math.Pow(y, 2)
            fourthDerivative = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 4)
            'Calculating error
            approxError = Math.Abs((((upperBound - lowerBound) * fourthDerivative) / 80) * Math.Pow(h, 4))
            exactVal = Approx - approxError

        ElseIf CboxFunctions.SelectedIndex = 1 Then
            For j As Integer = 1 To numberSubintervals Step 1
                Approx += (3 * h / 8) * ((Math.Pow(x(3 * j - 3), 2) + 1) + 3 * (Math.Pow(x(3 * j - 2), 2) +
1) + 3 * (Math.Pow(x(3 * j - 1), 2) + 1) + (Math.Pow(x(3 * j), 2) + 1))
            Next
            func = Function(y) Math.Pow(y, 2) + 1
```

```
            fourthDerivative = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 4)
            approxError = Math.Abs((((upperBound - lowerBound) * fourthDerivative) / 80) * Math.Pow(h, 4))
            exactVal = Approx - approxError

        ElseIf CboxFunctions.SelectedIndex = 2 Then
            For j As Integer = 1 To numberSubintervals Step 1
                Approx += (3 * h / 8) * (Math.Sin(Math.Pow(x(3 * j - 3), 2)) + 3 * Math.Sin(Math.Pow(x(3 *
j - 2), 2)) + 3 * Math.Sin(Math.Pow(x(3 * j - 1), 2)) + Math.Sin(Math.Pow(x(3 * j), 2)))
            Next
            func = Function(y) Math.Sin(Math.Pow(y, 2))
            fourthDerivative = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 4)
            approxError = Math.Abs((((upperBound - lowerBound) * fourthDerivative) / 80) * Math.Pow(h, 4))
            exactVal = Approx - approxError

        ElseIf CboxFunctions.SelectedIndex = 3 Then
            For j As Integer = 1 To numberSubintervals Step 1
                Approx += (3 * h / 8) * (Math.Exp(Math.Pow(x(3 * j - 3), 2)) + 3 * Math.Exp(Math.Pow(x(3 *
j - 2), 2)) + 3 * Math.Exp(Math.Pow(x(3 * j - 1), 2)) + Math.Exp(Math.Pow(x(3 * j), 2)))
            Next
            func = Function(y) Math.Exp(Math.Pow(y, 2))
            fourthDerivative = MathNet.Numerics.Differentiate.Derivative(func, midPointInterval, 4)
            approxError = Math.Abs((((upperBound - lowerBound) * fourthDerivative) / 80) * Math.Pow(h, 4))
            exactVal = Approx - approxError

        End If

        stopWatch.Stop()
        Return Approx
    End Function
****************************************************************************************************
```

```
'Function for calculating the factorial of a given number
    Public Function factorial(n As Integer) As Double
        If n = 0 Then
            Return 1
        Else
            Return n * factorial(n - 1)
        End If
    End Function


'Button subrutine for the back action
    Private Sub btnBack_Click(sender As Object, e As EventArgs) Handles btnBack.Click
        txtUpperBound.Clear()
        txtLowerBound.Clear()
        txtSubinterval.Clear()
        txtResults.Clear()
        CboxFunctions.SelectedIndex = -1
        Me.Close()
        frmMenu.Show()
    End Sub

    'Clear button subrutine for clearing the form's contents
    Private Sub btnClear_Click(sender As Object, e As EventArgs) Handles btnClear.Click
        txtUpperBound.Clear()
        txtLowerBound.Clear()
        txtSubinterval.Clear()
        txtTerms.Clear()
        txtResults.Clear()
        CboxFunctions.SelectedIndex = -1
        frmGraph.Hide()

    End Sub

    'Graph button subrutinte for graphing the function
    Public Sub btnGraph_Click(sender As Object, e As EventArgs) Handles btnGraph.Click
        frmGraph.Show()
    End Sub

End Class
****************************************************************************************************
```

```

'Button subrutine for the back action
    Private Sub btnBack_Click(sender As Object, e As EventArgs) Handles btnBack.Click
        txtUpperBound.Clear()
        txtLowerBound.Clear()
        txtSubinterval.Clear()
        txtResults.Clear()
        CboxFunctions.SelectedIndex = -1
        Me.Close()
        frmMenu.Show()
    End Sub

    'Clear button subrutine for clearing the form's contents
    Private Sub btnClear_Click(sender As Object, e As EventArgs) Handles btnClear.Click
        txtUpperBound.Clear()
        txtLowerBound.Clear()
        txtSubinterval.Clear()
        txtTerms.Clear()
        txtResults.Clear()
        CboxFunctions.SelectedIndex = -1
        frmGraph.Hide()

    End Sub

    'Graph button subrutinte for graphing the function
    Public Sub btnGraph_Click(sender As Object, e As EventArgs) Handles btnGraph.Click
        frmGraph.Show()
    End Sub

End Class

```

```
'   Copyright(c) 2017, Luis A. Flores
'   All rights reserved.

'Redistribution And use In source And binary forms, with Or without modification,
'   are permitted provided that the following conditions are met

'   Redistributions of source code must retain the above copyright notice,
'   this list of conditions And the following disclaimer.
```

```
'   Redistributions in binary form must reproduce the above
'copyright notice, this list Of conditions And the following disclaimer
'in the documentation And/Or other materials provided with the distribution.

'THIS SOFTWARE Is PROVIDED BY THE COPYRIGHT HOLDERS And CONTRIBUTORS
'"AS IS" And ANY EXPRESS Or IMPLIED WARRANTIES, INCLUDING, BUT Not LIMITED
'TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY And FITNESS FOR A PARTICULAR
'PURPOSE ARE DISCLAIMED. In NO Event SHALL THE COPYRIGHT OWNER Or CONTRIBUTORS
'BE LIABLE For ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
'Or CONSEQUENTIAL DAMAGES (INCLUDING, BUT Not LIMITED TO, PROCUREMENT OF
'SUBSTITUTE GOODS Or SERVICES; LOSS Of USE, DATA, Or PROFITS;
'Or BUSINESS INTERRUPTION) HOWEVER CAUSED And ON ANY THEORY OF LIABILITY,
'WHETHER IN CONTRACT, STRICT LIABILITY, Or TORT(INCLUDING NEGLIGENCE Or
'   OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
'EVEN If ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


' Numerical Integration App (NIA)
' Form: frmTechnique.vb
' by Luis A. Flores
' SICI4038 7/14/2017

'frmGraph.vb
'Class responsible for graphing the selected function.

Public Class frmGraph
    Private pBoxGraph As New PictureBox()
    Private Sub frmGraph_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        pBoxGraph.Dock = DockStyle.Fill
        pBoxGraph.BackColor = Color.White
        pBoxGraph.SizeMode = PictureBoxSizeMode.StretchImage

        ' Connect the Paint event of the PictureBox to the event handler method.
        AddHandler pBoxGraph.Paint, AddressOf Me.pBoxGraph_Paint
        ' Add the PictureBox control to the Form.
        Me.Controls.Add(pBoxGraph)
    End Sub


    Private Sub pBoxGraph_Paint(ByVal sender As Object, ByVal e As System.Windows.Forms.PaintEventArgs)
        ' Create a local version of the graphics object for the PictureBox.
```

```
        'Parameters needed to paint in the picturebox
        Dim graph As Graphics = e.Graphics
        Dim upperBound As Integer = 20
        Dim lowerBound As Integer = -20
        Dim subintervalLength As Double = 0.5
        Dim count As Integer = 0
        'Points are single data type (possible overflow)
        Dim pnts(80) As PointF

        'Offsets to start graphing in the center
        Dim Xoffset As Single = CSng(pBoxGraph.ClientSize.Width / 2)
        Dim Yoffset As Single = CSng(pBoxGraph.ClientSize.Height / 2)

        'Pens for painting in the picturebox
        Dim blackPen As New Pen(Color.Black, 0.0F)
        Dim bluePen As New Pen(Color.SkyBlue, 0.3F)

        'If-Elseif statements that get the points needed to paint the graph
        If frmTechnique.CboxFunctions.SelectedIndex = 0 Then
            For x As Double = lowerBound To upperBound Step subintervalLength
                pnts(count) = New PointF(CSng(x), CSng(-1 * Math.Pow(x, 2)))
                count += 1
            Next

        ElseIf frmTechnique.CboxFunctions.SelectedIndex = 1 Then
            For x As Double = lowerBound To upperBound Step subintervalLength
                pnts(count) = New PointF(CSng(x), CSng(-1 * (Math.Pow(x, 2) + 1)))
                count += 1
            Next

        ElseIf frmTechnique.CboxFunctions.SelectedIndex = 2 Then
            For x As Double = lowerBound To upperBound Step subintervalLength
                pnts(count) = New PointF(CSng(x), CSng(-1 * Math.Sin(Math.Pow(x, 2))))
                count += 1
            Next

        ElseIf frmTechnique.CboxFunctions.SelectedIndex = 3 Then

            For x As Double = lowerBound To upperBound / 2 Step subintervalLength
                pnts(count) = New PointF(CSng(x), CSng(-1 * Math.Exp(Math.Pow(x, 2))))
                count += 1
            Next
```

```vb



        End If

        Try
            'Showing the graphs name
            graph.SmoothingMode = Drawing2D.SmoothingMode.HighQuality
            If frmTechnique.CboxFunctions.SelectedIndex = 0 Then
                graph.DrawString("Graph:" + vbNewLine + "f(x)=x^2", New Font("Arial", 10), Brushes.Red, New PointF(30.0F, 30.0F))
            ElseIf frmTechnique.CboxFunctions.SelectedIndex = 1 Then
                graph.DrawString("Graph:" + vbNewLine + "f(x)=x^2 + 1", New Font("Arial", 10), Brushes.Red, New PointF(30.0F, 30.0F))
            ElseIf frmTechnique.CboxFunctions.SelectedIndex = 2 Then
                graph.DrawString("Graph:" + vbNewLine + "f(x)=Sin(x^2)", New Font("Arial", 10), Brushes.Red, New PointF(30.0F, 30.0F))
            ElseIf frmTechnique.CboxFunctions.SelectedIndex = 3 Then
                graph.DrawString("Graph:" + vbNewLine + "f(x)=e^(x^2)", New Font("Arial", 10), Brushes.Red, New PointF(30.0F, 30.0F))
            End If

            'Transform the scale (bigger or smaller)
            graph.TranslateTransform(Xoffset, Yoffset)
            graph.ScaleTransform(15, 20)
            'drawing cartesian plane
            graph.DrawLine(blackPen, -300, 0, 300, 0)
            graph.DrawLine(blackPen, 0, -200, 0, 200)
            'Drawing the curve
            graph.DrawCurve(bluePen, pnts)
            graph.ResetTransform()

            'Error message (possible overflow)
        Catch ex As Exception
            Me.Hide()
            MessageBox.Show("Application Error: Values of the function are too big." + vbNewLine + ex.Message, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error)
        End Try

    End Sub 'pictureBox1_Paint

End Class
```