Mario Jerez

Nick Rauh

Calculus II

21 December 2018

An Exploration of Numerical Integration with Scheme

In this project, I will explore differentiation techniques which can be utilized even when conventional means of manipulating equations are insufficient for solving more complex equations. Particularly, I will investigate different methods of numerical integration and compare their effectiveness in measuring the area underneath different curves. In this paper, I will outline the process by which I coded these methods of numerical integration into Racket, compare the approximations of the areas computed to the true area, and finally rank the different methods of integration by their ability to derive accurate results for different types of curves.

The most simple way to approximate the area between the curve and the x-axis is to draw multiple rectangles as shown in figures 1 and 2. It is worth recalling that an integral can be

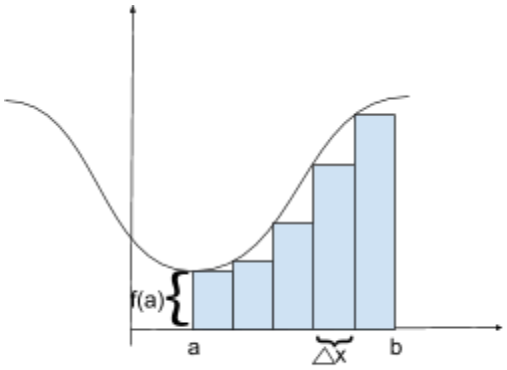


Figure 1: left-bound rectangle numerical integration technique (my own image).

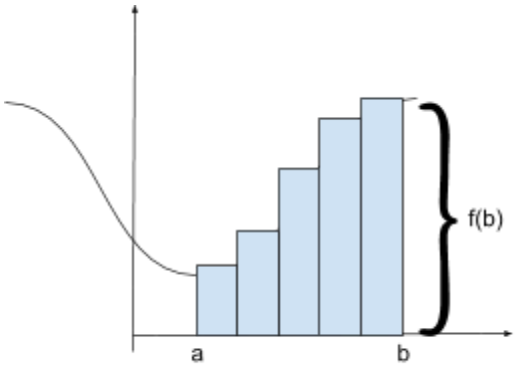


Figure 2: right-bound rectangle numerical integration technique (my own image).

defined as $\lim_{\Delta x \to 0} \sum_{a}^{b} f(x^*)\Delta x$, where Δx can be thought of as the width of each rectangle and $f(x)$ as the height.

There are three methods to take an antiderivative using rectangles:

- Let the left corner determine the height of the rectangle (figure 1).
- Let the right corner determine the height of the rectangle (figure 2).
- Let the midpoint determine the height of the rectangle.

A method of computing the area of a left-bound rectangle can be expressed in the formula $A = (x_{i+1} - x_i)f(x_i)$ where $x_{i+1} - x_i$ is the width of a rectangle which will be expressed as dx. The area of all of the rectangles can then be summed, as shown in the equation $A_T = dx(f(x_0) + f(x_1) + f(x_2) + ... + f(x_{n-1}))$. Note that $dx = \frac{x_n - x_0}{n} = \frac{\Delta x}{n}$ where x_0 and x_n are the left-most and right-most values of the domain respectively and n represents the number of rectangles. In my code, I use the notation x1 instead of x_0 and x2 instead of x_n.

Coding Numerical Integration by Left-bound Rectangles

For instructions on downloading the same version of scheme as I used in this project, please see the following link which will redirect you to the webpage of my FYS teacher, Jim Marshall: http://science.slc.edu/jmarshall/geb/concabs-library/instrux.html

Step 1: Set up the Recursion

In preparing my program, I chose the variables, n, x1, x2 as my inputs. Because I wanted my program to behave recursively, I specified that when n=0, it can stop computing. The first step to setting up the recursion was to make my problem smaller; that is, instead of inputting n, x1, x2

into the recursion, I inputted n-1, x1, x2-dx (remember that dx = $\frac{x_n - x_0}{n}$). This program only computes the rectangles highlighted in red in figure 3.

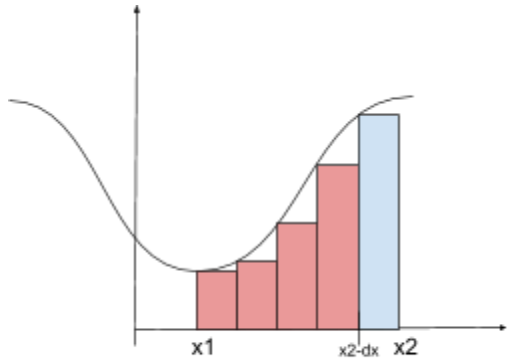


Figure 3 (my own image)

Now, the program needs to compute the area of the right-most rectangle:

$A_{rightmost} = dx \cdot f(x2 - dx)$. When written in the proper syntax, the program called area-reclft-f1-help (reclft stands for "rectangle left(bounded)") now looks like this:

```
(define area-reclft-f1-help
  (lambda (dx n x1 x2)
    (cond
      ((= n 0) '())
      (else (add-to-end (* dx (f1 (- x2 dx)))
                        (area-reclft-f1-help dx (- n 1) x1 (- x2
dx)))))))
```

Note: *add-to-end* and *add-all* are both programs that I defined earlier in the semester. They are expressed below:

```
(define add-to-end
  (lambda (n nums)
    (cond
      ((null? nums) (cons n '()))
      (else (cons (car nums) (add-to-end n (cdr nums)))))))

(define add-all
  (lambda (nums)
    (cond
      ((null? nums) 0)
      (else (+ (car nums) (add-all (cdr nums)))))))
```

This helper program provides the user with a list of areas for each rectangle it was asked for. For example, one could run the program by inputting the following:

Input: `>(area-reclft-f1-help 1 4 0 4)`
output: `(4.0 3.5416666666666665 2.666666666666667 2.875)`

To input a well-formed command, one has to provide it with the dx, n, x1, and x2 values. Then, all the values have to be summed. To make the program user friendly, the next step is to write a program that calculates dx and adds up the list for the user.

```
(define area-reclft-f1
  (lambda (n x1 x2)
    (add-all (area-reclft-f1-help (/ (- x2 x1) n) n x1 x2))))
```

Now, the user can input the command

```
> (area-reclft-f1 4 0 4)
```

And receive

```
13.083333333333334
```

f1 is another program that I have defined which represents the function $f(x) = \frac{x^4}{4!} - \frac{x^2}{2} + 4$ and is expressed in Scheme as:

```
(define f1
  (lambda (x)
    (+ (- (/ (power x 4) 24) (/ (power x 2) 2)) 4.0)))
```

Another program f2 is referenced in the report which can be expressed as the function $g(x) = x + 4$ or as

```
(define f2
  (lambda (x)
    (+ x 4)))
```

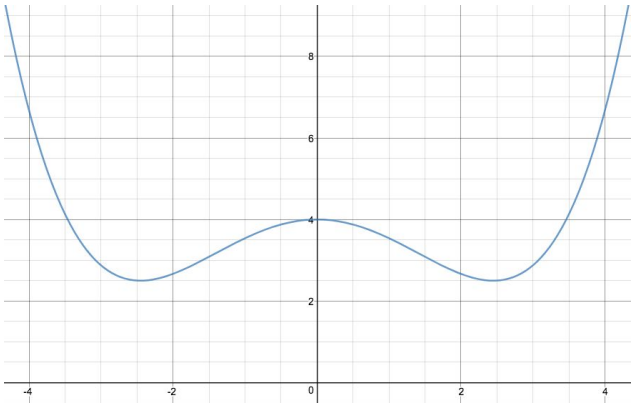


Figure 4: f1 graphed by desmos

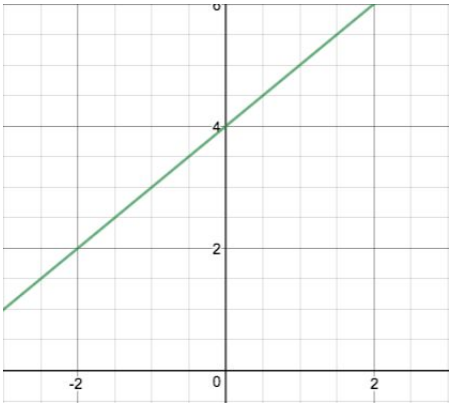


Figure 5: f2 graphed by Desmos

Note that while the examples I use in this paper only call on the program f1, one can simply replace f1 with f2 to calculate the area under g(x).

Coding Numerical Integration by Right-bound Rectangles

Making a code for right-bound rectangles is similar to the process described above. The only difference is that now dx is multiplied by f(x2) rather than f(x2-dx) when calculating the right-most rectangle.

```
(define area-recrt-f1-help
  (lambda (dx n x1 x2)
    (cond
      ((= n 0) '())
      (else (add-to-end (* dx (f1 x2))
                        (area-recrt-f1-help dx (- n 1) x1 (- x2
dx)))))))


(define area-recrt-f1
  (lambda (n x1 x2)
    (add-all (area-recrt-f1-help (/ (- x2 x1) n) n x1 x2))))
```

Coding Numerical Integration for Mid-bound rectangles

Mid-bound rectangles have their height determined by their midpoint. The midpoint of the rectangle furthest to the right is $xn - \frac{dx}{2}$.

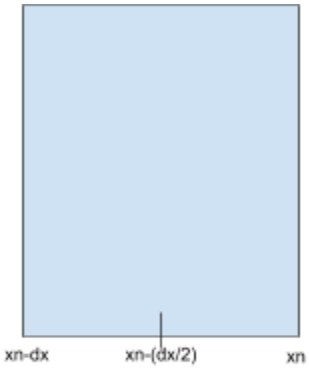


Figure 6: Midpoint of Rectangle

```
(define area-recmid-f1-help
  (lambda (dx n x1 x2)
    (cond
      ((= n 0) '())
      (else (add-to-end (* dx (f1 (- x2 (* (/ 1 2) dx))))
                        (area-recmid-f1-help dx (- n 1) x1 (- x2
dx)))))))


(define area-recmid-f1
   (lambda (n x1 x2)
    (add-all (area-recmid-f1-help (/ (- x2 x1) n) n x1 x2))))
```

Coding Numerical Integration for Trapezoids

The geometry of trapezoids allows the shape to stay with the curve for longer, so one might expect that this method will offer more accurate approximations than rectangles.

$$Area\ of\ a\ trapezoid = \frac{f(x_i)+f(x_{i+1})}{2}dx$$

$$Sum\ of\ Areas\ of\ trapezoids = \frac{f(x_0)+f(x_1)}{2}dx + \frac{f(x_1)+f(x_2)}{2}dx + \frac{f(x_2)+f(x_3)}{2}dx + ... + \frac{f(x_{n-1})+f(x_n)}{2}dx$$

Where $f(x_0) + f(x_1)$ can also be written as f(x1) + f(x1+dx) in my program's notation.

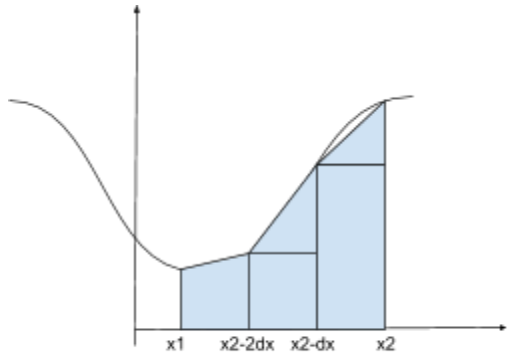


Figure 7: Numerical Integration with Trapezoids
(my own image)

This time, the area of the right-most trapezoid can be calculated by $\frac{f(x2)+f(x2-dx)}{2}$.
Everything else works the same way as the last two programs.

```
(define area-trap-f1-help
  (lambda (dx n x1 x2)
    (cond
      ((= n 0) '())
      (else (add-to-end (* dx (/ (+ (f1 x2) (f1 (- x2 dx))) 2))
                        (area-trap-f1-help dx (- n 1) x1 (- x2
dx)))))))

(define area-trap-f1
  (lambda (n x1 x2)
    (add-all (area-trap-f1-help (/ (- x2 x1) n) n x1 x2))))
```

Coding the Simpson's Rule

The Simpson's rule utilizes parabolas' ability to trace a curve. Calculating one of the segments shown in figure 8 is significantly more complicated than techniques used previously.

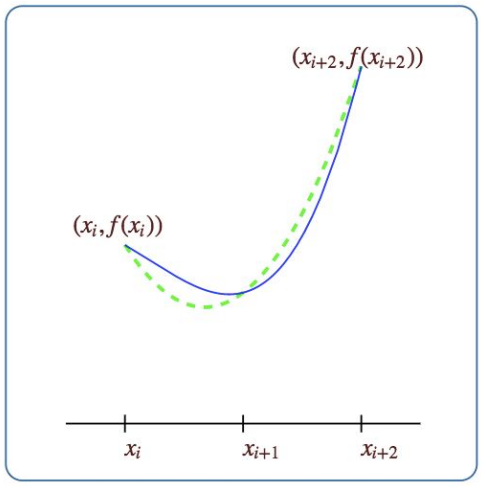


Figure 9: Best fit parabola from Simpson's Rule (Whitman)

Simpson's Rule takes advantage of the fact that any three points in a 2-dimensional space correspond to only one quadratic equation associated to a particular parabola (as long as they have different x coordinates). A parabola will need three points, such as (x2-2dx, f(x2-2dx)), (x2-dx, f(x2-dx), (x2, f(x2). Consequently the parabola can only be divided into an even number of segments to have enough points to work with.

To acquire the quadratic equation for each parabola, one has to solve for a, b, and c for these three equations (Guichard):

$$f(x_n - 2 \cdot dx) = a(x_n - 2 \cdot dx)^2 + b(x_n - 2 \cdot dx) + c$$

$$f(x_n - dx) = a(x_n - dx)^2 + b(x_n - dx) + c$$

$$f(x_n) = a(x_n)^2 + b(x_n) + c$$

However, because Simpson's rule requires three points, translating the equations written above into code would be misleading because inputted n would only be treated as half of simpson's segment in the algorithm. So instead, I adapted the formulas to better suit my purpose:

$$f(x_n - dx) = a(x_n - dx)^2 + b(x_n - dx) + c$$

$f(x_n - \frac{dx}{2}) = a(x_n - \frac{dx}{2})^2 + b(x_n - \frac{dx}{2}) + c$

$f(x_n) = a(x_n)^2 + b(x_n) + c$

Now, n will sincerely be equal to the number of segments and will no longer be restricted to even numbers. After doing the algebra, one can find the area of each segment (Guichard):

$A = \frac{dx}{6}(f(x_n - dx) + 4f(x_n - \frac{dx}{2}) + f(x_n)$

The same computation was used in creating the code:

```
(define area-sim-f1-help
  (lambda (dx n x1 x2)
    (cond
      ((= n 0) '())
      (else (add-to-end (* (/ dx 6) (+ (f1 (- x2 dx)) (+ (* 4 (f1 (-
x2 (/ dx 2)))) (f1 x2))))
                        (area-sim-f1-help dx (- n 1) x1 (- x2
dx)))))))


(define area-sim-f1
  (lambda (n x1 x2)
    (add-all (area-sim-f1-help (/ (- x2 x1) n) n x1 x2))))
```

Testing the Effectiveness of Numerical Integration Techniques

Rectangles

I found that when testing rectangular methods on a first degree polynomial (f2), there was no difference in accuracy between using left-bound rectangles and a right-bound rectangles. The important difference between the two methods when approximating a positive linear slope was that left-bound rectangles underestimated the integral while the right-bound rectangles

overestimated the integral. Nevertheless, the error of the left-bounded rectangles canceled out the error of the right-bounded rectangles, giving them identical percentage errors.

Note: $\%\ error = \frac{(derived\ value)-(true\ value)}{true\ value} \bullet 100$

	Left-bounded Rectangles		Right-bounded Rectangles	
Number of Rectangles (n)	Approximated Area	% Error	Approximated Area	% Error
1	16	33.3333333	32	33.3333333
10	23.2	3.33333333	24.8	3.33333333
100	23.91999999999	0.33333333	24.07999999999	0.33333333
1000	23.99199999999	0.03333333	24.00799999999	0.03333333
10000	23.99920000000	0.00333333	24.00080000000	0.00333333
Actual Value	24			

Table 1: Approximate area under curve of the function g(x)=x+4 (f2) on [0, 4].

Because the over approximations and the under approximations canceled each other out, it wasn't so surprising to see that the method where rectangles which were bounded at the midpoint had 0% error for any natural number n. The results in this portion of the investigation suggested that for any first degree polynomial, left and right bounded rectangles share the same % error value when they also share the same n value while the areas of mid-bounded rectangles equal the area under the curve no matter the number of rectangles.

However, the differences between the methods which incorporate rectangles become magnified when analyzing >2 degree polynomials, especially with quickly accelerating positive slopes and quickly decelerations negative slopes. For example, table 2 shows data from

approximations of area under the curve of f1 on $[\sqrt{6},\ 10]$. These points were chosen because, as depicted in figure 4, this domain begins with slope zero and quickly accelerates to a steep, positive slope.

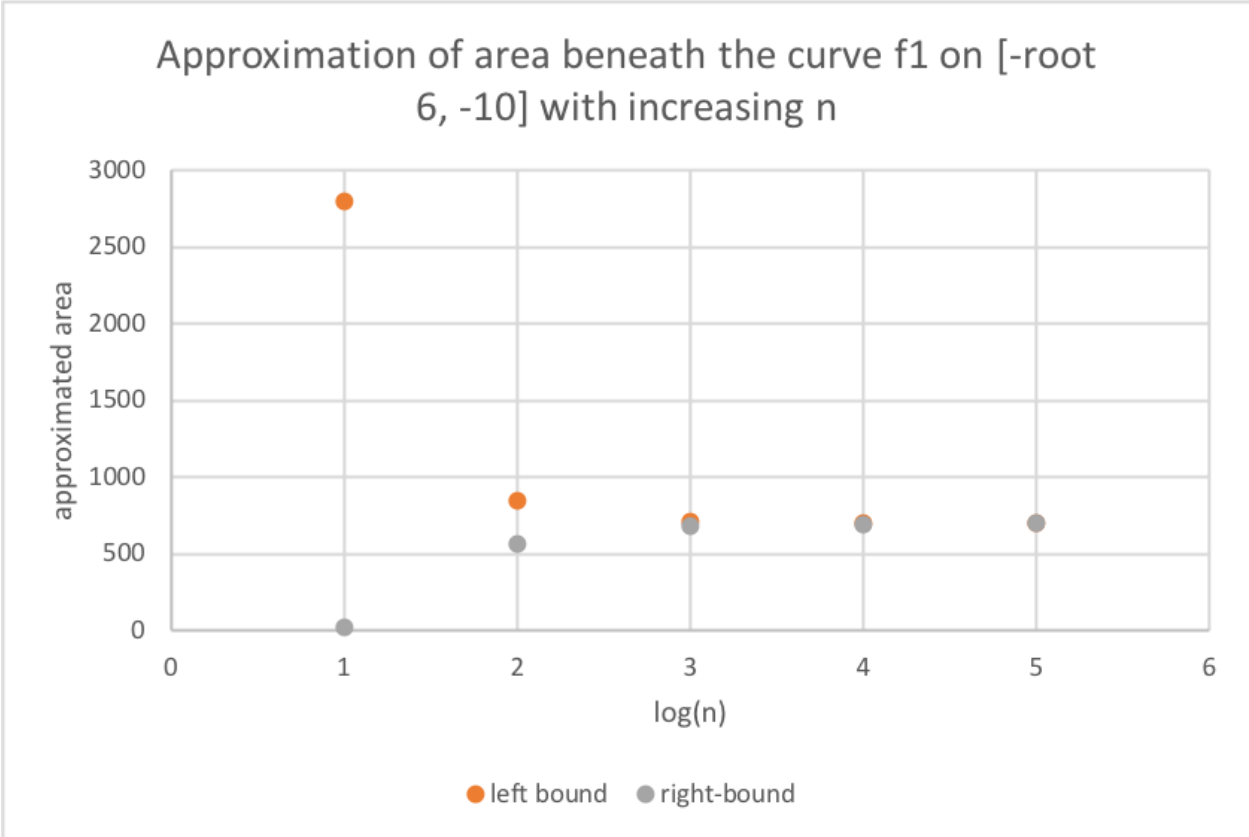
	Left-bounded Rectangles		Right-bounded Rectangles		Mid-bounded Rectangles	
Number of Rectangles (n)	Approximated Area	% Error	Approximated Area	% Error	Approximated Area	% Error
1	18.876276	97.297921	2798.7225	300.62828	356.25679	49.002965
10	567.03063	18.831356	845.01525	20.961264	694.86483	0.5322940
100	684.75855	1.9789766	712.55701	2.0002854	698.54614	0.0053272
1000	697.19417	0.1988566	699.97402	0.1990697	698.58298	5.327E-05
10000	698.44437	0.0198952	698.72235	0.0198974	698.58335	5.328E-07
Actual Value	698.583350515482					

Table 2: Approximate area under an accelerating positive curve of f1 on $[\sqrt{6},\ 10]$.

The table shows that particularly when using a smaller number of rectangles, right-bounded rectangles are the most unreliable when analyzing an accelerating positive slope. Mid-bound rectangles, on the other hand, have shown to be the most effective slope-approximating rectangular technique for both quadratics and first degree polynomials.

A similar analysis to table 2 was taken with the only difference being that the range was changed to $[-10,\ -\sqrt{6}]$. Due to the symmetry of f1, this range provides a comparable decelerating negative slope. The results turned out to be exactly the same except that the approximated area and % error values that were derived for left-bounded rectangles in table 2

were instead derived for right-bounded rectangles, and vice versa. Graph 2 shows how in this situation, the over-approximation of the left-bounded rectangles overwhelmed the under-approximation of the right-bounded rectangles.

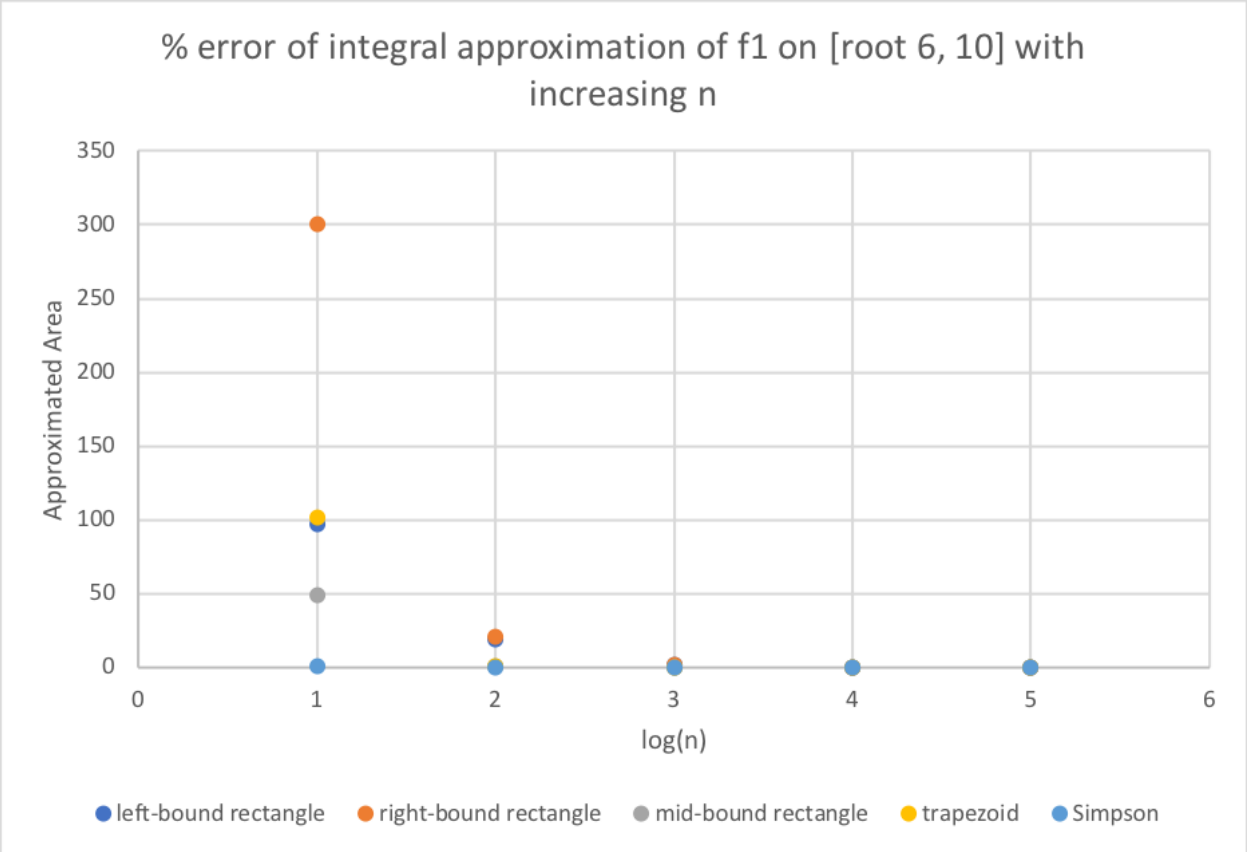


Graph 1: Approximation of area beneath a negative, decelerating slope.

These findings suggest that regardless of what graph is being analyzed, mid-bounded rectangles are more accurate than any other rectangular method. Otherwise, left-bounded rectangles are the next best approximators for accelerating positive slopes and right-bound rectangles are the next best approximators for decelerating negative slopes.

Overall Numerical Integration Results

As I expected, the trapezoidal approximation method consistently gave more accurate measurements then the left-bounded and right-bounded rectangle methods, and, like the midpoint and simpson technique, approximated the area under a linear line perfectly for any $n \geq 1$. However, I was surprised to see that the midpoint technique out-performed the trapezoidal technique consistently.



Graph 2

Graph 1 provides an example where the mid-point technique shows to be more accurate than the trapezoid technique. For a detailed comparison, see table 3 below.

	Mid-bounded Rectangles		Trapezoids	
Number of Rectangles (n)	Approximated Area	% Error	Approximated Area	% Error
1	356.256794	49.00296522	1408.79937	101.6651801
10	694.864833	0.532293986	706.022941	1.064953897
100	698.546136	0.005327166	698.65778	0.010654369
1000	698.582978	5.32721E-05	698.584095	0.000106544
10000	698.583347	5.32748E-07	698.583358	1.06541E-06

Table 3: % error of integral approximation on $[\sqrt{6},\ 10]$

Graph 2 also illustrates, as does the rest of the data collected, that the Simpson Rule offers by far the most accurate approximation of area beneath a graph.

Conclusion

In conclusion, the project gave me a more conceptual understanding of what it meant to differentiate an equation. The process of translating the equations into code required me to have a fully visual understanding of what was going on. While the formulas themselves were not very difficult to work with, I hit several roadblocks along the way when writing the code and consulted with my FYS teacher Jim Marshall for guidance a couple of times.

It was an exciting experience preparing and conducting an empirical experiment on the nature of shapes and numbers. I was pleasantly surprised to have come across results that I did not expect, namely that mid-bound rectangles made better approximators than trapezoids. While my current computer programming skills are limited, I think that this project has room for growth. Particularly, I was interested in writing a program that could automatically switch between left and right bounded rectangles whenever the slope experienced a change in sign. I am

excited to see what else can be learned by combing computer programing and mathematics in my future studies.

Works Cited

Guichard, David. “8.6 Numerical Integration.” Whitman, https://www.whitman.edu/mathematics/calculus_online/section08.06.html.