APPROXIMATING DEFINITE INTEGRALS [WITH ERRORS] VIA PYTHON

Calculus 2 Final Project

Abstract

This program calculates the value of an integral by using the Trapezoidal Rule, Midpoint Rule, and Simpson's Rule. After this, it calculates the errors of the approximation using numerical analysis.

George Gannon

Gmg3f@mtmail.mtsu.edu

Code at a Glance (Python 3, written in IDLE, Sympy/Numpy Required)

```
1.  import sympy as sym
2.  from sympy import Symbol, sin, cos # for trig
3.  from sympy.solvers import solve
4.  from sympy.utilities.lambdify import lambdify # for shortening lines 41-43
5.  import numpy as np # for arange method
6.  x = Symbol('x')
7.
8.  def f2(x): # Changing this changes function integrated
9.      return sin(x)**0.5
10.
11. def Midpoint(f, step, a, b, n): # calculates by midpoint rule
12.     integral_value = 0
13.     x_points = np.arange(a,b,step)
14.     for array_index in x_points[:]:
15.         integral_value += (f2(0.5*(array_index+array_index+step)))
16.     integral_value *=step
17.
18.     print('Midpoint Sum: The function is approximately equal to',format(integral_value, '.4f'))
19.
20. def calculate_derivatives(rule, f2):
21.     if rule == 'T':
22.         derivative = sym.diff(f2(x),x,x)
23.     elif rule == 'M':
24.         derivative = sym.diff(f2(x),x,x) # second derivative for trap or mid
25.     else:
26.         derivative = sym.diff(f2(x),x,x,x,x) # fourth derivative for simps
27.     return derivative
28.
29. def calculate_error(derivative, a ,b, n, x, method):# FInd critical numbers, plug in, get max, plug into error
30.     new_derivative = sym.diff(derivative,x)
31.     try:
32.         critical_numbers = [sym.nsolve(new_derivative,a,real = True)]
33.     except ValueError:
34.         critical_numbers = [0]
35.     except KeyError:
36.         critical_numbers = [0]
37.     except ZeroDivisionError:
38.         print('The error could not be calculated. I am sorry.''\n') # For when an integral is breaking everything
39.         return
40.     except:
41.         critical_numbers = [sym.nsolve(new_derivative,a, real=True)]
42.     critical_numbers.append(a)
43.     critical_numbers.append(b) # Previous two lines put in endpoints into critical numbers list
44.     d = lambdify(x,derivative, 'sympy')
45.     critical_numbers = [x for x in critical_numbers if not(x<a or x>b)] #Removes critical numbers not in [a,b]
46.     critical_numbers = [abs(d(x)) for x in critical_numbers] # Absolute values all indices and plugs them into respective derivative for calculating maximum
47.     K = max(critical_numbers) # Finds maximum of them
48.     if method == 'M': # letters determine which error calculation formula to use
49.         error = (K*((b-a)**3)) / (24*n**2)
50.         print('The approximate error is:',error, '\n')
```

```python
    elif method == 'T':
        error = (K*((b-a)**3)) / (12*n**2)
        print('The approximate error is:',error, '\n')
    elif method == 'Simpson':
        error = (K*((b-a)**5)) / (180*n**4)
        print('The approximate error is:',error, '\n')


def Trapezoidal(f, step, a, b, n): # Calculates by Trapezoidal rule
    outside = 0.5 * step # (b-a)/n
    x_points = np.arange(a,b,step)
    integral_value = f2(a)
    for array_index in x_points[1:]:
        integral_value += (2*f2(array_index))
    integral_value += f2(b)
    integral_value *= outside

    print('Trapezoidal sum: The function is approximately equal to', format(integral_value, '.4f'))

def Simpson(f2, step, a, b, n): #Extremely accurate
    outside = step/3
    x_points = np.arange(a,b,step)
    integral_value = f2(a)
    ordinate = 1
    for array_index in x_points[1: ]:
        if ordinate%2==0:
           integral_value += (2*f2(array_index))#multiply by 2 if ordinate number is even
           ordinate +=1
        else:
            integral_value += (4*f2(array_index))#multiply by 4 if ordinate number is odd
            ordinate +=1
    integral_value += f2(b)
    integral_value *= outside

    print("Simpson's Rule: The function is approximately equal to", format(integral_value, '.4f'))

def main():
    beginning_interval = float(input('Enter the beginning of the interval: '))
    ending_interval = float(input('Enter the end of the interval: '))
    number_of_subintervals = int(input('Enter the number of subintervals: '))
    print('\n''The function to be integrated is:',str(f2(x)),'\n')
    # Collecting intervals, n, and step size
    a = beginning_interval
    b = ending_interval
    n = number_of_subintervals
    step = ((b-a)/ number_of_subintervals)
    # Reassigning for clarity sake
    f = f2(step)
    Midpoint(f2, step, a, b, n)
    derivative = calculate_derivatives('M', f2)
    error = calculate_error(derivative, a, b, n, x, 'M')

    Trapezoidal(f2, step, a, b, n)
    derivative = calculate_derivatives('T', f2)
    error = calculate_error(derivative, a, b, n, x, 'T')

    Simpson(f2, step, a, b, n)
```

```
108.    derivative = calculate_derivatives('Simpson', f2)
109.    error = calculate_error(derivative, a, b, n, x, 'Simpson')
110.
111.main()
```

Key Locations

Function for Integration: Line 9

Midpoint Rule: Line 11-18

Simpson’s Rule: Line 70-85

Trapezoidal Rule: Line 59-68

Main Routine: Line 87-109

Derivative Calculation: 20-27

Error Calculation: 29-56

Goals and Methodology of Project

Before I began coding, I wanted this, in addition to being an exercise in Riemann sums, I wanted the program to be a practice in using packages I haven't used before in Python. I had heard of the power of Sympy and Numpy, so I imported those in and got to work. We had been working with lists and arrays in my CSCI 1170 class, so I also wanted to have some practice using those. Therefore, the code itself is most likely *not the optimal solution* to the problem at hand, and it might be very CPU intensive. Also, I have no experience whatsoever coding in an environment besides IDLE, so I did not use say, a Jupyter Notebook.

I first researched all the methods to make sure I had a good grasp on what mathematics had to done. I had knowledge on how Midpoint and Trapezoidal sums worked, but I will admit I had never utilized Simpson's Rule. Calculating errors would also prove to be a learning experience as I had never done it.

Coding

After doing some brief 'sketching' and experimenting using a placeholder IDLE file, I concluded that the best approach would be to explode the interval of integration into tiny pieces separated by a 'step' size. The step size would be calculated as a variable in the main() and then get passed as an argument into the three main functions (see line 83). After finding the step size, I would put it into the .arange() method with arguments (a,b, step) so that it made an array similar to the following:

For a =1, b=2 ,n=10 and step = (2-1)/10=0.1

[1, 1.1, 1.2, 1.3, 1.4…2.0]

I began coding, and put in a user prompt for specifying a, b, and the number of subintervals. I first coded the midpoint rule (line 12). It proved to be the shortest in terms of code. Next, I coded the Trapezoidal rule and first used a variable called 'outside' which is essentially like the outside numbers of the calculation. In Trapezoidal rule's case, it is delta-x (step)/2. This type of code would appear again in Simpson's Rule. After researching how Simpson's Rule worked, I typed a code out that gave me an extremely wrong integral. After some debugging, I found that it was not properly multiplying values and had to code in an 'ordinate' system. This solved the bug by ensuring only the odd valued x-sub values would be multiplied by 4, and the even valued ones would be multiplied by 2.

To ensure it was properly calculating the sums, I used a very simple function X^2 as I could easily calculate the integral. I found that there was a point where a subinterval around 1000 could severely skew the calculations, but after increasing it or *decreasing it*, it would fix itself. I imagine this was due to the arrange function truncating decimals in an odd fashion. I enhanced the complexity of the functions until it was working for polynomials of the seventh degree.

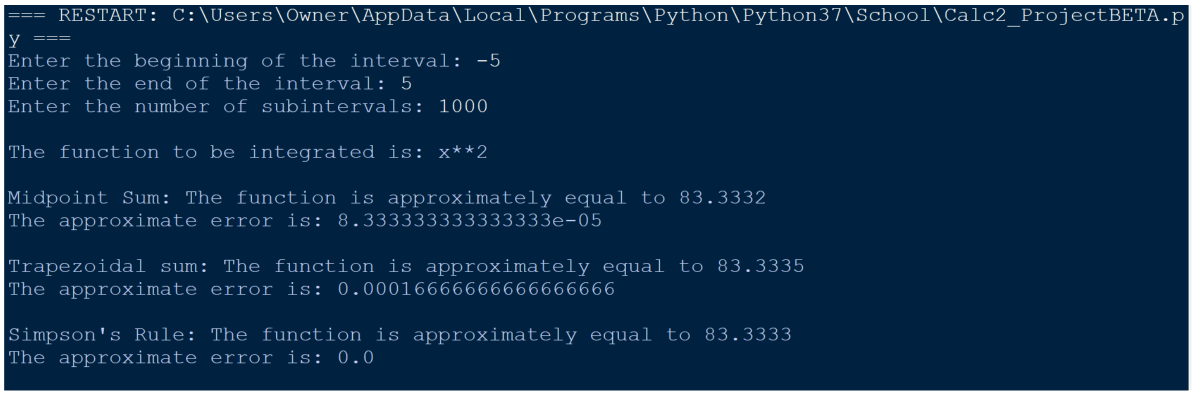
I took a week-long break, and then came back to code the error calculation. This proved to be a very bad move. I started off by coding a calculate derivative function as both error calculation formulas required them. This was not bad as I got to practice using sympy methods (sym.diff). However, once I began to actually code the error calculations, roadblocks began to happen. Exception after exception occurred, new bugs arose from fixing old ones, certain methods didn't work because of conflicting whitespace, etc. I had to use many try and exception blocks to get it working. However, after fixing those issues, I can now say that it works with a few limitations. Getting trigonometry to work with the code proved to be a learning experience, but now Sin and Cos work. I imagine that converting a few functions in the program to use SciPy methods would perhaps simplify the code, and maybe even make it even more accurate.

Limitations/Bugs

1. Interval must be such that **a<b**
2. Trigonometry only works for two of the three basic functions.
    a. Sin and Cos are verified to be working, but strange errors occur that keep Tan from having precise calculations.
    b. Not been tested with inverse trig, or reciprocals.
3. Strange error occurs with intervals
    a. Can replicate by setting f2 (line 9) to x^2 and then n to 1000.
4. At larger intervals of a and b, the error calculation can be near zero, but still be off by margin, or be near the integral value and show a large error. If I had to make an educated guess with my limited Analysis knowledge, this might be due to non-real answers.

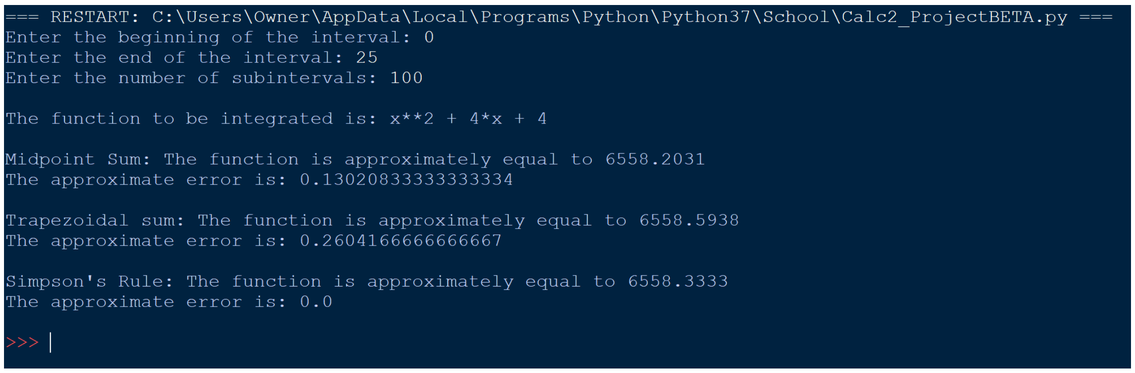
Sample Outputs

$$\int_{-5}^{5} x^2\,dx$$



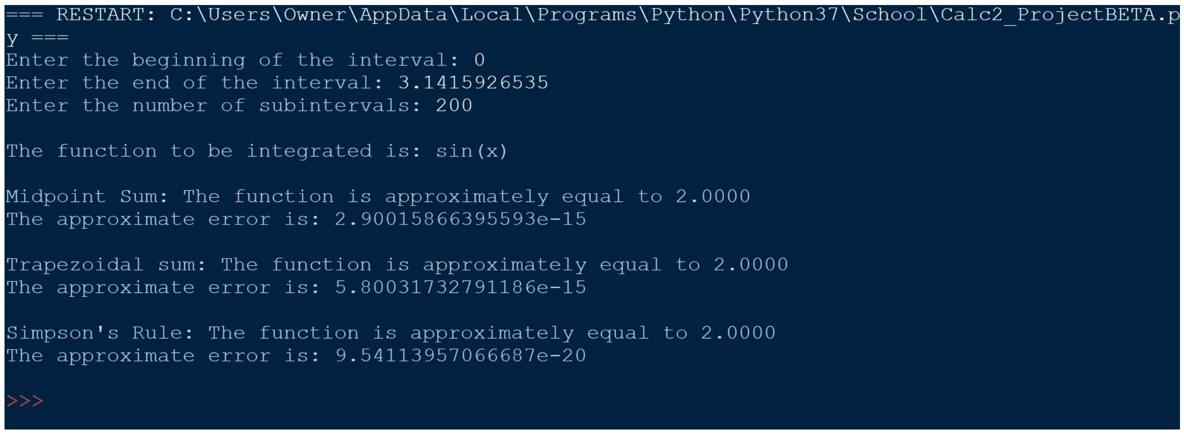
```
=== RESTART: C:\Users\Owner\AppData\Local\Programs\Python\Python37\School\Calc2_ProjectBETA.p
y ===
Enter the beginning of the interval: -5
Enter the end of the interval: 5
Enter the number of subintervals: 1000

The function to be integrated is: x**2

Midpoint Sum: The function is approximately equal to 83.3332
The approximate error is: 8.333333333333333e-05

Trapezoidal sum: The function is approximately equal to 83.3335
The approximate error is: 0.00016666666666666666

Simpson's Rule: The function is approximately equal to 83.3333
The approximate error is: 0.0
```

$$\int_{0}^{25} x^2 + 4x + 4\,dx$$



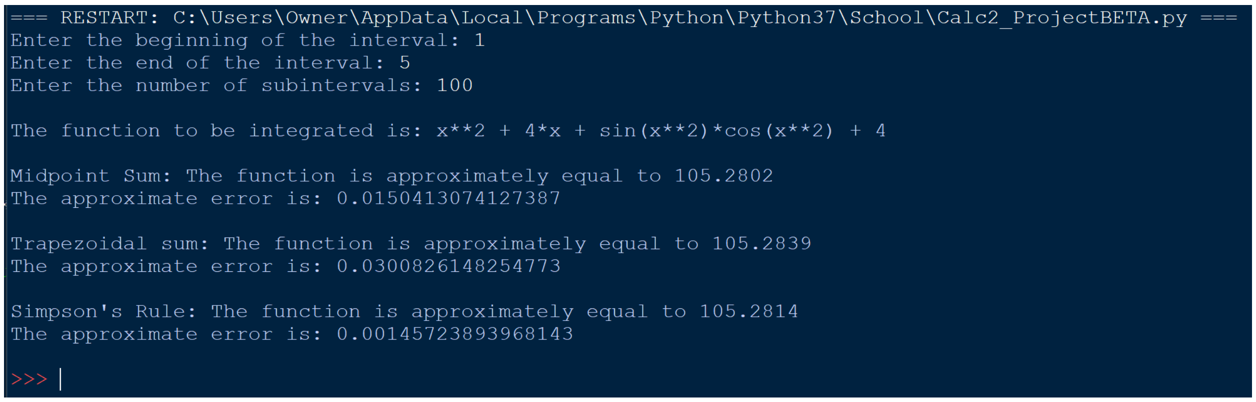
```
=== RESTART: C:\Users\Owner\AppData\Local\Programs\Python\Python37\School\Calc2_ProjectBETA.py ===
Enter the beginning of the interval: 0
Enter the end of the interval: 25
Enter the number of subintervals: 100

The function to be integrated is: x**2 + 4*x + 4

Midpoint Sum: The function is approximately equal to 6558.2031
The approximate error is: 0.13020833333333334

Trapezoidal sum: The function is approximately equal to 6558.5938
The approximate error is: 0.2604166666666667

Simpson's Rule: The function is approximately equal to 6558.3333
The approximate error is: 0.0

>>> |
```

$$\int_{0}^{\pi} \sin(x)\,dx$$



```
=== RESTART: C:\Users\Owner\AppData\Local\Programs\Python\Python37\School\Calc2_ProjectBETA.p
y ===
Enter the beginning of the interval: 0
Enter the end of the interval: 3.1415926535
Enter the number of subintervals: 200

The function to be integrated is: sin(x)

Midpoint Sum: The function is approximately equal to 2.0000
The approximate error is: 2.900158663955593e-15

Trapezoidal sum: The function is approximately equal to 2.0000
The approximate error is: 5.800317327911186e-15

Simpson's Rule: The function is approximately equal to 2.0000
The approximate error is: 9.54113957066687e-20

>>>
```

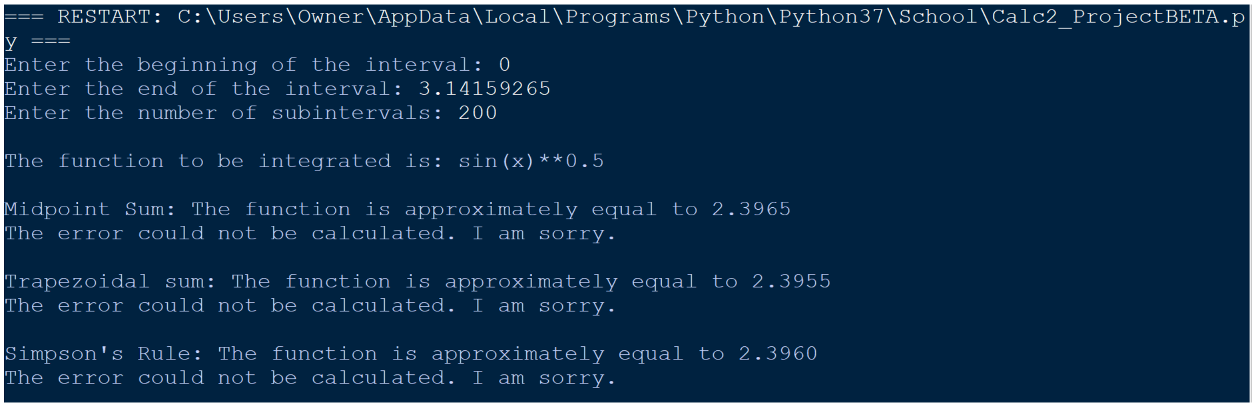
$$\int_1^5 x^2 + 4x + 4 + \sin(x^2) * \cos(x^2)\, dx$$



```
=== RESTART: C:\Users\Owner\AppData\Local\Programs\Python\Python37\School\Calc2_ProjectBETA.py ===
Enter the beginning of the interval: 1
Enter the end of the interval: 5
Enter the number of subintervals: 100

The function to be integrated is: x**2 + 4*x + sin(x**2)*cos(x**2) + 4

Midpoint Sum: The function is approximately equal to 105.2802
The approximate error is: 0.0150413074127387

Trapezoidal sum: The function is approximately equal to 105.2839
The approximate error is: 0.0300826148254773

Simpson's Rule: The function is approximately equal to 105.2814
The approximate error is: 0.00145723893968143

>>> |
```

*This output is interesting because the output is the real part of a complex number.

$$\int_0^\pi \sqrt{\sin(x)}$$



```
=== RESTART: C:\Users\Owner\AppData\Local\Programs\Python\Python37\School\Calc2_ProjectBETA.p
y ===
Enter the beginning of the interval: 0
Enter the end of the interval: 3.14159265
Enter the number of subintervals: 200

The function to be integrated is: sin(x)**0.5

Midpoint Sum: The function is approximately equal to 2.3965
The error could not be calculated. I am sorry.

Trapezoidal sum: The function is approximately equal to 2.3955
The error could not be calculated. I am sorry.

Simpson's Rule: The function is approximately equal to 2.3960
The error could not be calculated. I am sorry.
```

In order to prevent a crash when a zero-division error occurs during error calculation, I coded in an exception block that breaks the error calculation.