Project 1 Writeup

Cricket Bergner

February 18, 2026

1 Introduction

Project 1 shows how different approximations for integration such as the trapezoid rule, Simpson's rule, and Gaussian quadrature can vary in accuracy and runtime. Each method will work more efficiently for certain integrals, intervals, or number of slices values. These differences will be explored through this writeup.

1.1 Languages and Libraries

The code for this project was written in Python. The libraries used were numpy, math, prettytable, scipy, matplotlib, and scipy.special. While all the libraries were well documented and easy to follow, a few stand out due to their particularly interesting contributions to the code.

PrettyTable is a fairly simple Python library in terms of the expanse of its function; it is only used to format tables. However, the tables are printed in an attractive ASCII form, which makes for visually appealing output. PrettyTable also formats the tables on its own. It centers input and adds division lines between the columns. While the visual appeal of formatted tables was not a necessity for the assignment, the organization this library naturally applies to datasets made it easy to track output.

Scipy was another interesting library, as from its subsidiary, scipy.special, it would calculate the roots of any N-th degree Legendre polynomial, along with the coefficients assigned to each root, called the weight. When summed, the weighted sum of the function evaluations could approximate an integral. In general, scipy is a great library for mathematical tasks a bit more complicated than the typical multiplication, square rooting, and exponentiation done by numpy, as it includes functions for integration and linear algebra operators. Scipy.special is a specialized library within scipy which gives access to special functions in mathematics, such as Bessel functions or the Legendre polynomials. Overall, it was a helpful tool in the coding for this assignment and simplified a calculation that could have been twenty lines of code into four or five.

2 Undergirding Theory

2.1 The Definite Integral

The definite integral is described by Equation 1, where a and b are the endpoints of the interval, f(x) is the function, and Δx and x_i^* are the width and height of the sub intervals (n) that the interval is broken into.

$$\int_a^b f(x)\,dx = \lim_{N\to\infty} \sum_{i=1}^{N} f(x_i^*)\,\Delta x, \qquad \Delta x = \frac{b-a}{N}, \quad x_i^* = a + i\Delta x \tag{1}$$

As the number of sub intervals increases, the approximation of the integral will become more accurate. However, this equation is for the area under f(x) between a and b to be broken into rectangles. This is where the left and right endpoint rules emerge. While this method also lends itself to the midpoint approximation–where $x_i^* = a + \Delta x/2 + i\Delta x$–there are other ways to approximate the a-b interval.

2.2 Trapezoid Rule

The trapezoid rule is obtained by averaging the left and right endpoint methods of approximation. This gives forth an approximation as shown by Equation 2.

$$I_T = \frac{1}{2}(I_L + I_R) = \sum_{i=0}^{N-1} \frac{1}{2}\left[f(a+ih) + f(a+ih+h)\right] h \tag{2}$$

The trapezoid rule is more effective than using left or right endpoints. Since it averages left and right endpoints, the error reduces by a factor $O(h^2)$, a second order reduction, while left and right endpoints error reduces by a factor of $O(h)$, a first order reduction. Thus, the trapezoid rule will converge to the correct answer much faster.

2.3 Simpson's Rule

Simpson's rule uses both the midpoint and trapezoidal rules to approximate integrals using quadratic polynomials. As a result, it converges faster and is significantly more efficient than the left, right, midpoint, and trapezoidal rules when the integral's endpoints are smooth. The error is reduced by $O(h^4)$, a fourth order reduction, making it converge exponentially faster than the other approximation methods.

2.4 Legendre Polynomials and Gaussian Quadrature

The Legendre polynomials are a series of orthogonal polynomials, first derived from the multipole expansion in electrostatics. Since they are orthogonal, $P_i \cdot P_j = \delta_{i,j}$, meaning $P_i \cdot P_j = 1$ if $i = j$ and $P_i \cdot P_j = 0$ if $i \neq j$. The Legendre polynomials and Gaussian quadrature are intimately connected.

For an n-point Gaussian quadrature rule, there are 2n unknowns, the nodes and weights. However, it can achieve an exactness of $2n-1$. Given a general polynomial of degree $2n-1$, by long division it becomes Equation 3.

$$p(x) = q(x)L_n(x) + r(x), \tag{3}$$

Then, integrate over the interval [-1,1].

$$\int_{-1}^{1} p(x)\,dx = \int_{-1}^{1} q(x)L_n(x)\,dx + \int_{-1}^{1} r(x)\,dx \tag{4}$$

$$\int_{-1}^{1} q(x)L_n(x)\,dx = 0 \tag{5}$$

$$\int_{-1}^{1} p(x)\,dx = \int_{-1}^{1} r(x)\,dx \tag{6}$$

The high-degree portion disappears. If the integral ignores the higher order term, then the Gaussian quadrature must as well.

$$L_n(x_i) = 0, \qquad i = 1, \ldots, n \tag{7}$$

There is a leftover troublesome term, but if the quadrature points are chosen to be the zeroes of the Legendre polynomials, then the term vanishes. Thus, any $2n - 1$ degree polynomial can be split apart, and the orthogonality of the Legendre polynomials cancels off the troublesome part. The remaining piece can be matched easily, allowing for Gaussian quadrature to make an approximation.

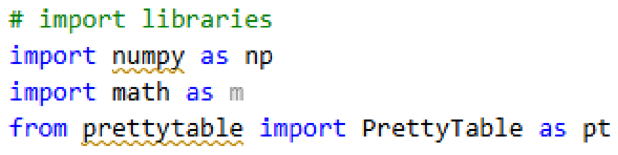
While this makes Gaussian quadrature effective, it is only for polynomial-like functions on well defined intervals. When the endpoints approach a singularity, this breaks the assumptions needed for a polynomial approximation, which will cause the error for Gaussian quadrature to scale like $O(N^{-1})$. For high N values, the approximation will take longer and longer to run.

3 Walking Through the Code

Each section below follows a similar pattern: import libraries, set up function definitions, then run a loop that will produce the desired output. Each subsection will cover the different areas of the code in a more specific detail.

3.1 Adaptive Integration

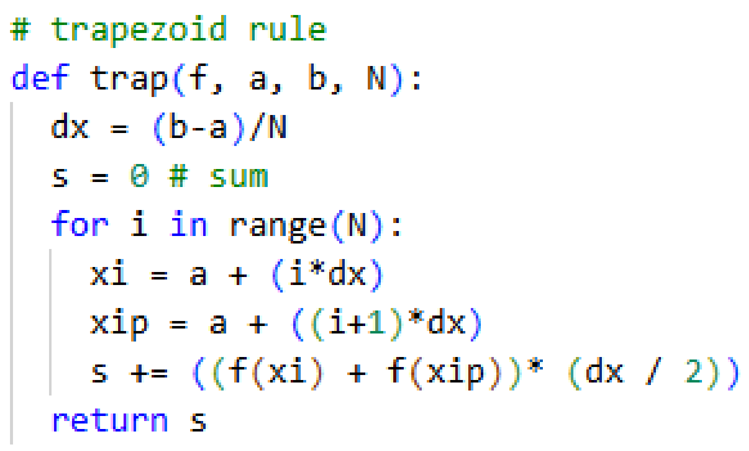
This section starts by importing math, numpy, and prettytable libraries for math functions and table formatting, respectively.



```
# import libraries
import numpy as np
import math as m
from prettytable import PrettyTable as pt
```

Figure 1: Section of code showing the numpy, math, and prettytable libraries being imported.

The trapezoid rule function is defined after the initial variables are set, defining the Δx and x_i^* terms, titled dx and xi. A for loop is used to sum the terms over N intervals.

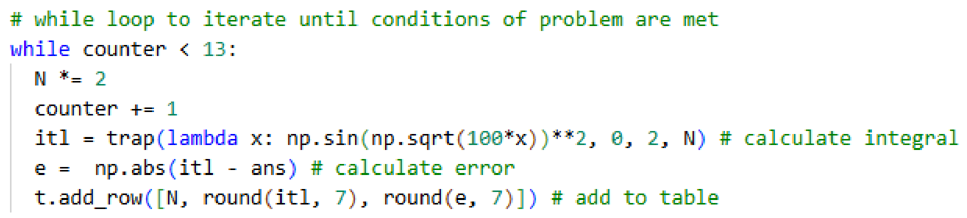


```
# trapezoid rule
def trap(f, a, b, N):
  dx = (b-a)/N
  s = 0 # sum
  for i in range(N):
    xi = a + (i*dx)
    xip = a + ((i+1)*dx)
    s += ((f(xi) + f(xip))* (dx / 2))
  return s
```

Figure 2: The Python function defined for the trapezoid rule, with initial variables defined for Δx and x_i^*, along with a for loop to sum over N number of intervals.

After this, a while loop runs over a set of conditions where the N value is increased (increasing the number of intervals), and for each N, the trapezoid rule is approximated for the integral listed in Equation 8, and the error between the trapezoid function's estimate and correct value is calculated. All these values are put into a table. From the table, the number of intervals needed to correctly approximate the integral to 10^{-6} error can be extracted.

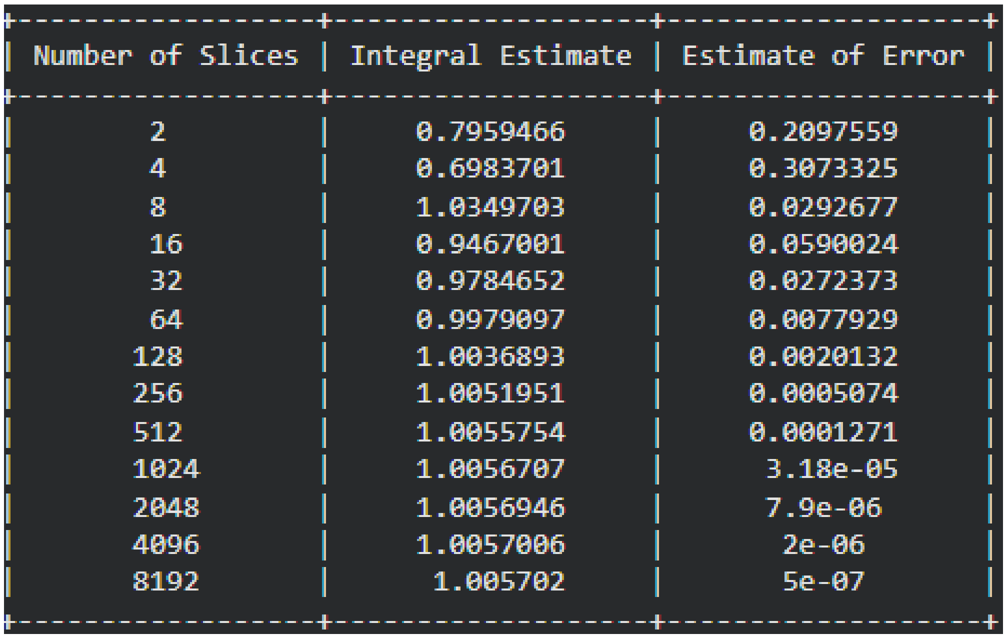
$$\int_0^2 \sin^2\left(\sqrt{100x}\right)\, dx \tag{8}$$



```
# while loop to iterate until conditions of problem are met
while counter < 13:
  N *= 2
  counter += 1
  itl = trap(lambda x: np.sin(np.sqrt(100*x))**2, 0, 2, N) # calculate integral
  e =  np.abs(itl - ans) # calculate error
  t.add_row([N, round(itl, 7), round(e, 7)]) # add to table
```

Figure 3: A while loop running over a counter to approximate Equation 8 using the trapezoid rule under an increased number of sub intervals and the error between the estimate and the correct answer.

The loop gave Figure 4 as a result.

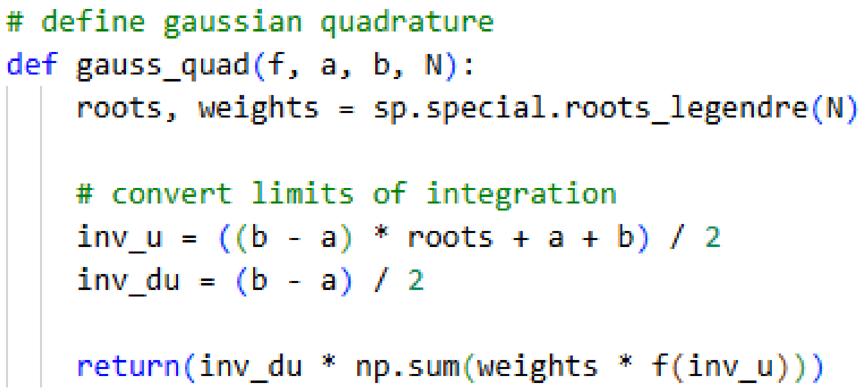


```
+------------------+-------------------+-------------------+
| Number of Slices | Integral Estimate | Estimate of Error |
+------------------+-------------------+-------------------+
|        2         |     0.7959466     |     0.2097559     |
|        4         |     0.6983701     |     0.3073325     |
|        8         |     1.0349703     |     0.0292677     |
|        16        |     0.9467001     |     0.0590024     |
|        32        |     0.9784652     |     0.0272373     |
|        64        |     0.9979097     |     0.0077929     |
|       128        |     1.0036893     |     0.0020132     |
|       256        |     1.0051951     |     0.0005074     |
|       512        |     1.0055754     |     0.0001271     |
|       1024       |     1.0056707     |      3.18e-05     |
|       2048       |     1.0056946     |      7.9e-06      |
|       4096       |     1.0057006     |       2e-06       |
|       8192       |      1.005702     |       5e-07       |
+------------------+-------------------+-------------------+
```

Figure 4: A figure of the code output, showing a table generated by the while loop using the trapezoid rule to approximate the integral.

3.2 Gaussian Quadrature

For this section, only the scipy library needed to be imported for scipy.special, which will be used to get the roots and weights for the Nth order Legendre polynomial. After this, the Gaussian quadrature function was defined. The function goes through the process of converting the given limits of integration from [a,b] to [-1, 1], then performs a u-substitution over the integral to make it easier to approximate.



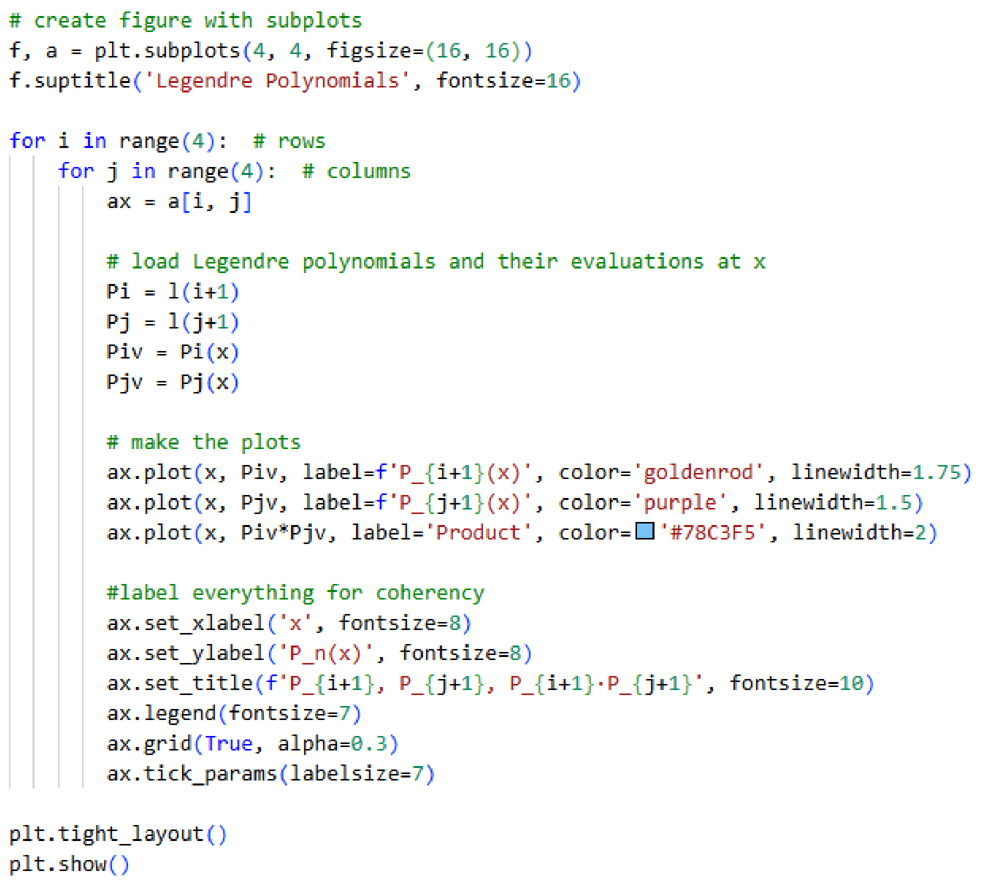
```python
# define gaussian quadrature
def gauss_quad(f, a, b, N):
    roots, weights = sp.special.roots_legendre(N)

    # convert limits of integration
    inv_u = ((b - a) * roots + a + b) / 2
    inv_du = (b - a) / 2

    return(inv_du * np.sum(weights * f(inv_u)))
```

Figure 5: The function definition for Gaussian quadrature. It uses scipy.special to calculate the roots and weights for an Nth order Legendre polynomial, converts the limits of integration to [-1,1], then performs a u-substitution before making the approximation by summing over the function values at each of the weights.

After this, Equation 8 is passed through the function, giving the output 1.0057025428257227.

3.3 Subplots

For this section, the Legendre polynomial functions were going to be plotted on a four by four grid, with P_i, P_j, and $P_i \times P_j$ plotted with each other for $1 \leq i, j \leq 4$. This was done in a nested for loop, where the outer loop cycled through the rows, and the inner loop cycled through the columns. Within the loops, the given Legendre polynomials at i and j were evaluated, then plotted with each other. Different colors and labels were assigned to each line to keep the graphs coherent and visually pleasing. An important factor in getting this to work was the Python subplot function, which allows multiple plots to be shown together without having space in between.



```python
# create figure with subplots
f, a = plt.subplots(4, 4, figsize=(16, 16))
f.suptitle('Legendre Polynomials', fontsize=16)

for i in range(4):  # rows
    for j in range(4):  # columns
        ax = a[i, j]

        # load Legendre polynomials and their evaluations at x
        Pi = l(i+1)
        Pj = l(j+1)
        Piv = Pi(x)
        Pjv = Pj(x)

        # make the plots
        ax.plot(x, Piv, label=f'P_{i+1}(x)', color='goldenrod', linewidth=1.75)
        ax.plot(x, Pjv, label=f'P_{j+1}(x)', color='purple', linewidth=1.5)
        ax.plot(x, Piv*Pjv, label='Product', color='#78C3F5', linewidth=2)

        #label everything for coherency
        ax.set_xlabel('x', fontsize=8)
        ax.set_ylabel('P_n(x)', fontsize=8)
        ax.set_title(f'P_{i+1}, P_{j+1}, P_{i+1}·P_{j+1}', fontsize=10)
        ax.legend(fontsize=7)
        ax.grid(True, alpha=0.3)
        ax.tick_params(labelsize=7)

plt.tight_layout()
plt.show()
```

Figure 6: The subplot function and nested for loops which generated the Legendre polynomials, P_i, P_j, and $P_i \times P_j$ plotted with each other for $1 \leq i, j \leq 4$. The loops assign each line a different color and label the lines for visual clarity.

The end result of this loop is shown below in Figure 7.

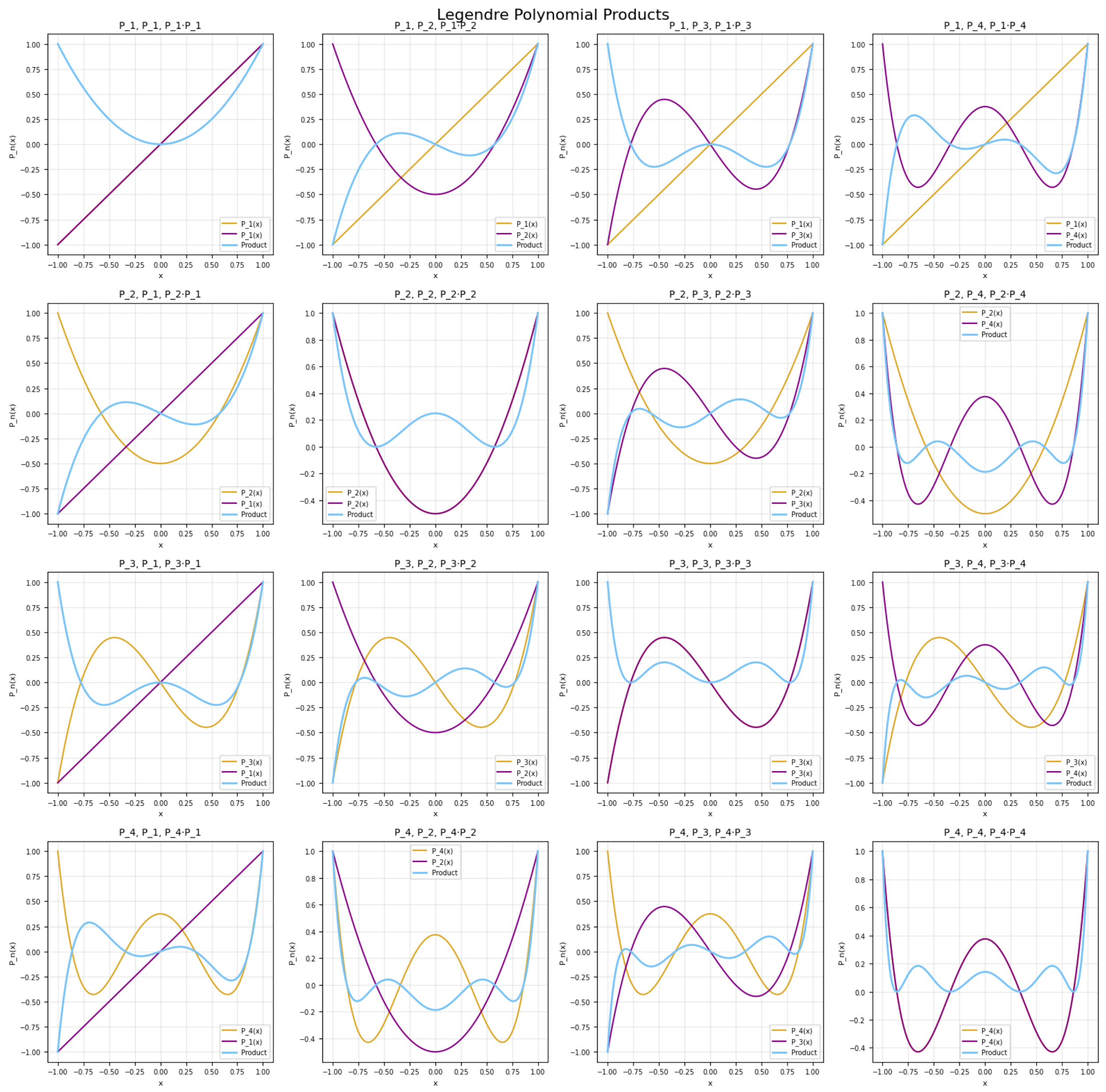


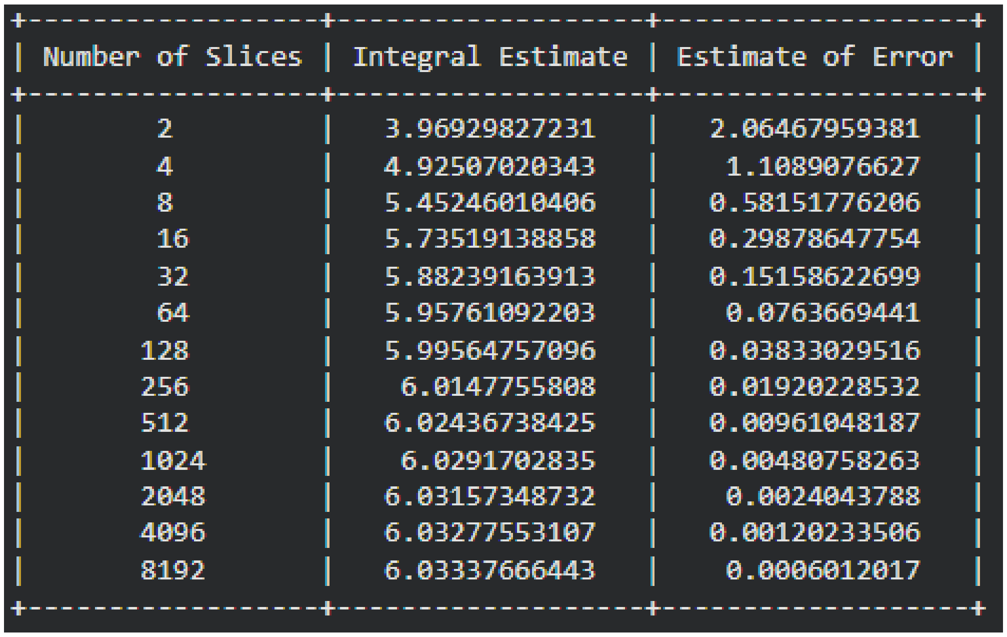
Figure 7: The output generated by the nested for loops that shows the Legendre polynomials, P_i, P_j, and $P_i \times P_j$ plotted with each other for $1 \leq i, j \leq 4$.

3.4 Extensions

This section introduces a new integral meant to challenge the Gaussian quadrature method of approximating integrals. This integral is not smooth at its endpoint $y = 2$, which increases the error scaling significantly, causing the Gaussian quadrature function to take longer and longer times to evaluate N sub intervals. A while loop runs Equation 9 through the Gaussian quadrature function to show that the approximation can still be done for smaller N values. This while loop is similar to the ones above.

$$\int_0^2 \frac{y^2}{\sqrt{2-y}}\,dy \tag{9}$$

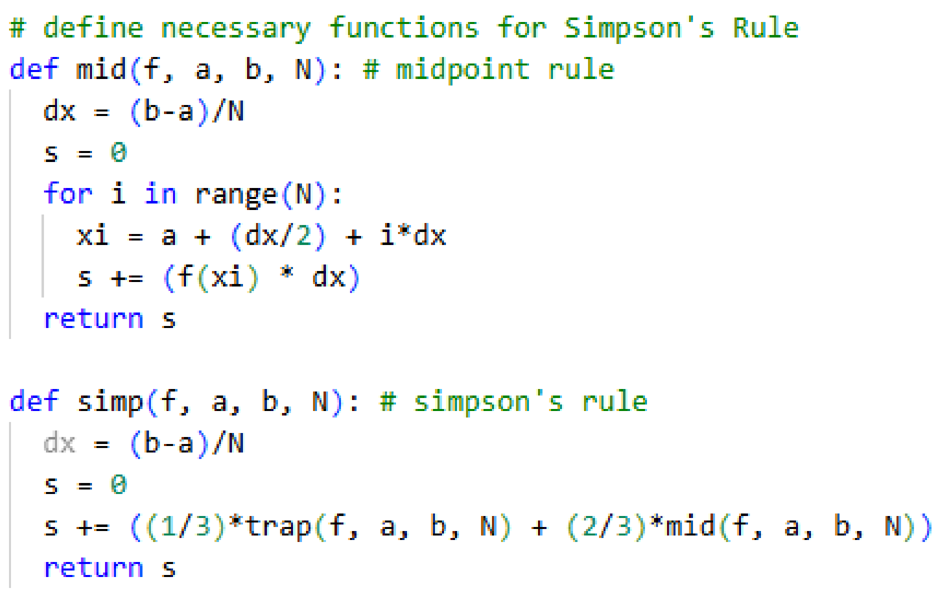
The while loop generates Figure 8.



```
+------------------+--------------------+-------------------+
| Number of Slices | Integral Estimate  | Estimate of Error |
+------------------+--------------------+-------------------+
|        2         |   3.96929827231    |   2.06467959381   |
|        4         |   4.92507020343    |    1.1089076627   |
|        8         |   5.45246010406    |   0.58151776206   |
|        16        |   5.73519138858    |   0.29878647754   |
|        32        |   5.88239163913    |   0.15158622699   |
|        64        |   5.95761092203    |    0.0763669441   |
|       128        |   5.99564757096    |   0.03833029516   |
|       256        |    6.0147755808    |   0.01920228532   |
|       512        |   6.02436738425    |   0.00961048187   |
|       1024       |    6.0291702835    |   0.00480758263   |
|       2048       |   6.03157348732    |    0.0024043788   |
|       4096       |   6.03277553107    |   0.00120233506   |
|       8192       |   6.03337666443    |    0.0006012017   |
+------------------+--------------------+-------------------+
```

Figure 8: A figure of the code output, showing a table generated by the while loop using the Gaussian quadrature rule to approximate the integral. The number of intervals does not go very high due to the increase in runtime.

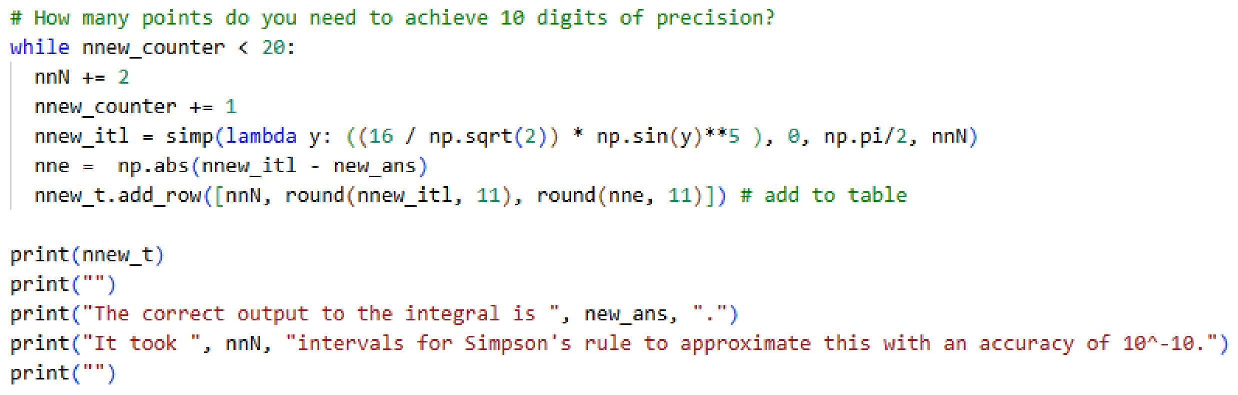
Simpson's rule requires the midpoint rule to be defined, which works similar to the left and right endpoint rules learned in calculus 1. Simpson's rule itself approximates an integral using both the midpoint and trapezoid rule.



```python
# define necessary functions for Simpson's Rule
def mid(f, a, b, N): # midpoint rule
    dx = (b-a)/N
    s = 0
    for i in range(N):
        xi = a + (dx/2) + i*dx
        s += (f(xi) * dx)
    return s

def simp(f, a, b, N): # simpson's rule
    dx = (b-a)/N
    s = 0
    s += ((1/3)*trap(f, a, b, N) + (2/3)*mid(f, a, b, N))
    return s
```

Figure 9: Function definitions for the midpoint rule, which approximates integrals using Δx and the midpoints of the N sub intervals, and Simpson's rule, which approximates integrals by summing (1/3) times the trapezoid rule and (2/3) times the midpoint rule.

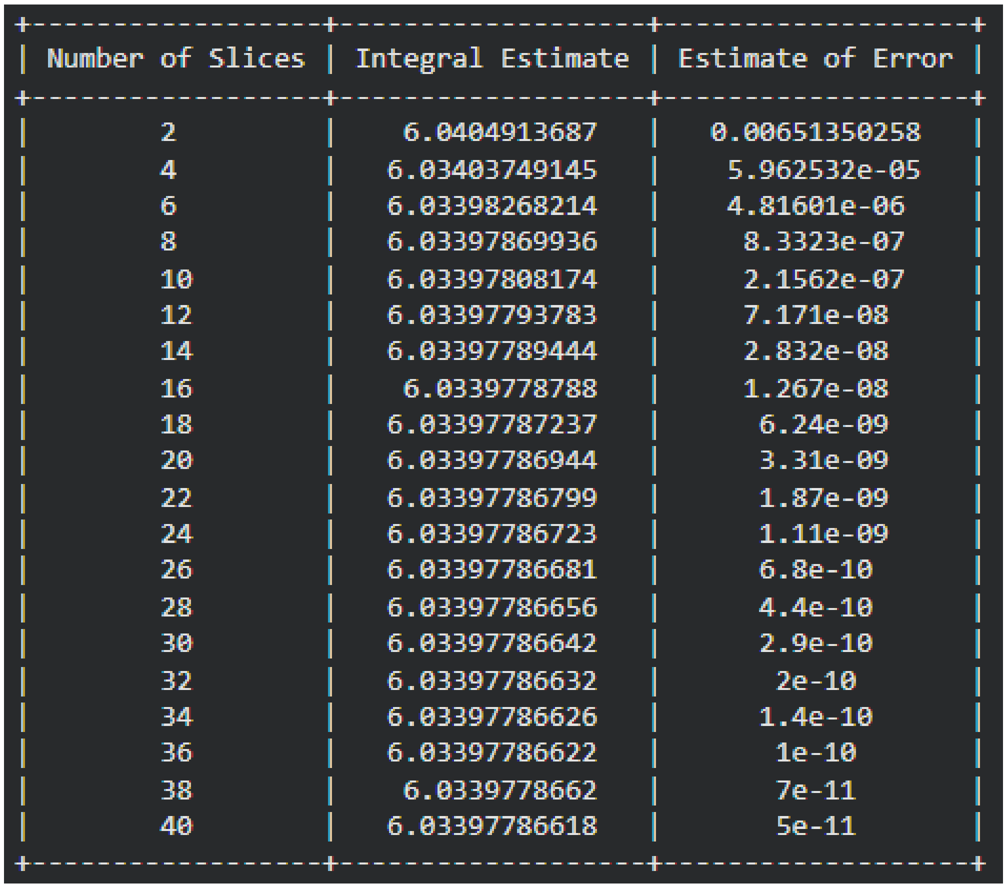
Using the u-substitution $y = 2\sin^2\theta$, Simpson's rule iterates over increasing N values to approximate the integral to an accuracy of 10^{-10}.



```python
# How many points do you need to achieve 10 digits of precision?
while nnew_counter < 20:
    nnN += 2
    nnew_counter += 1
    nnew_itl = simp(lambda y: ((16 / np.sqrt(2)) * np.sin(y)**5 ), 0, np.pi/2, nnN)
    nne =  np.abs(nnew_itl - new_ans)
    nnew_t.add_row([nnN, round(nnew_itl, 11), round(nne, 11)]) # add to table

print(nnew_t)
print("")
print("The correct output to the integral is ", new_ans, ".")
print("It took ", nnN, "intervals for Simpson's rule to approximate this with an accuracy of 10^-10.")
print("")
```

Figure 10: The while loop that iterates over an increasing counter, using Simpson's rule to evaluate the integral over an increasing N value, recording N, the approximation, and error.

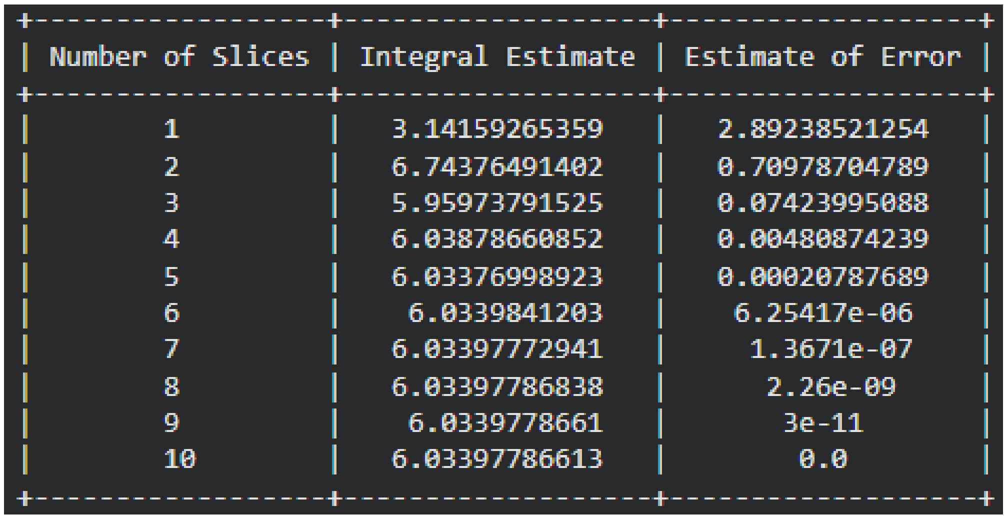
The loop generated Figure 11.



```
+-------------------+-------------------+-------------------+
| Number of Slices  | Integral Estimate | Estimate of Error |
+-------------------+-------------------+-------------------+
|         2         |    6.0404913687   |   0.00651350258   |
|         4         |   6.03403749145   |    5.962532e-05   |
|         6         |   6.03398268214   |    4.81601e-06    |
|         8         |   6.03397869936   |     8.3323e-07    |
|        10         |   6.03397808174   |     2.1562e-07    |
|        12         |   6.03397793783   |     7.171e-08     |
|        14         |   6.03397789444   |     2.832e-08     |
|        16         |    6.0339778788   |     1.267e-08     |
|        18         |   6.03397787237   |      6.24e-09     |
|        20         |   6.03397786944   |      3.31e-09     |
|        22         |   6.03397786799   |      1.87e-09     |
|        24         |   6.03397786723   |      1.11e-09     |
|        26         |   6.03397786681   |      6.8e-10      |
|        28         |   6.03397786656   |      4.4e-10      |
|        30         |   6.03397786642   |      2.9e-10      |
|        32         |   6.03397786632   |       2e-10       |
|        34         |   6.03397786626   |      1.4e-10      |
|        36         |   6.03397786622   |       1e-10       |
|        38         |    6.0339778662   |       7e-11       |
|        40         |   6.03397786618   |       5e-11       |
+-------------------+-------------------+-------------------+
```

Figure 11: A figure of the code output, showing a table generated by the while loop using Simpson's rule to approximate the integral.

After this, the new integral is fed back to the Gaussian quadrature function, which approximates it to an accuracy of 10^{-10} within 9 intervals thanks to the u-substitution fixing the smoothness at its endpoints. This is shown in Figure 12.



```
+------------------+-------------------+-------------------+
| Number of Slices | Integral Estimate | Estimate of Error |
+------------------+-------------------+-------------------+
|        1         |   3.14159265359   |   2.89238521254   |
|        2         |   6.74376491402   |   0.70978704789   |
|        3         |   5.95973791525   |   0.07423995088   |
|        4         |   6.03878660852   |   0.00480874239   |
|        5         |   6.03376998923   |   0.00020787689   |
|        6         |    6.0339841203   |    6.25417e-06    |
|        7         |   6.03397772941   |     1.3671e-07    |
|        8         |   6.03397786838   |      2.26e-09     |
|        9         |    6.0339778661   |       3e-11       |
|        10        |   6.03397786613   |        0.0        |
+------------------+-------------------+-------------------+
```

Figure 12: A figure of the code output, showing a table generated by the while loop using Gaussian quadrature rule to approximate the new integral. Now that the bounds have been adjusted, it converges much faster.

4 Lessons Learned

It was good to review different loops (for, while, etc). Much of my research work with Python has been editing existing code or rooting out errors, so writing it from scratch was a nice reset into the basics. I was also good to review the different Python libraries and their functions; the math will be very helpful if Desmos fails with future homework or I need to plot different relationships. Making the subplots using a for loop inside of a for loop was also challenging, but good fun, as I liked to think of it as making the elements in a matrix. Another, and possibly the most important lesson, was relearning how to organize code, comment clearly what is going on, and name variables such as it is clear what their function is, while keeping the code compact.

4.1 Time Keeping

The code took around seven hours to write, edit, format, and proofread. The writeup took five to six hours. However, that is not counting the several hours of undocumented time either in class, discussing the assignment among peers, thinking about how to solve problems within the assignment, or asking questions in person. With all that in mind I would say a good estimate of time would be around twenty hours.

4.2 Attributions

- `https://www.geeksforgeeks.org/python/python-while-loop/` This was to make sure I was using the correct notation with a while loop.
- `https://www.integral-calculator.com` To find the correct answer to the integral

so I could calculate error.

- https://python4physics.in/program/python/program.php?menu_id=14&submenu_id=2#gsc.tab=0 This helped me figure out how to use existing Python libraries to generate the Legendre polynomials.
- https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.grid.html To learn a bit more about adding grids in plots.
- https://matplotlib.org/stable/users/explain/colors/colors.html I wanted to see if I could add custom colors to the lines in plots. Turns out you can!
- https://www.w3schools.com/python/matplotlib_subplot.asp How to use the subplot function.
- https://en.wikipedia.org/wiki/Legendre_polynomials A great overview of the Legendre polynomials and what math helped derive them.
- https://math.stackexchange.com/questions/1275344/why-are-quadrature-points-given-by A useful stack exchange forum to explain the derivation of Gaussian quadrature using the zeros of the Legendre polynomials.