The Gaussian Quadrature

In Lecture 5, we explored the three fundamental methods for computing definite integrals numerically: by use of rectangles, the Trapezoid Rule, and Simpson's Rule. We also found that the third method, Simpson's Rule, is superior to the other two methods when it comes to obtaining accurate approximation of any integral. Therefore, if we know how to use Simpson's Rule, there's really no reason for us to even bother using the rectangle method or the Trapezoid Rule as they are obsolete.

However, there is a fourth method that wasn't explicitly mentioned: $The\ Gaussian\ Quadrature\ Rule$. Like with Simpson's Rule, we may be asking "Is this method better than even Simpson's Rule?". The short answer to this question is yes, and we will see later by looking at some examples.

The $Gaussian\ Quadrature\ Rule$ was named after Carl Friedrich Gauss, and it was meant to produce exact results for polynomials of degree $2n-1$ or less by a suitable choice of the nodes x_i and weights w_i for $i = 1, 2, 3, \ldots, n$(Wikipedia). Whereas the rectangle, Trapezoid, and Simpson's rule all require that the width(Δx) be the same, the Gaussian Quadrature Rule dictates that the opposite is more suitable: unequal widths will give better approximation.

So how does the $Gaussian\ Quadrature\ Rule$ work? Suppose we have a function that we want to integrate over the interval $[a, b]$. Let that function be a function of x and let the area under the curve of that function be denoted by A. Then,

$$(1)\ A = \int_a^b f(x)dx$$

As we can see, f is being integrated over the interval $[a, b]$. We need to shift this interval into the following

$$[a, b] \implies [-1, 1]$$

before the Gaussian Quadrature method can work. We will do this in the next section.

Shift(Change) of Interval

The interval of integration for Equation 1 must be changed from $[a, b]$ to $[-1, 1]$. Once we do this, the general form of the Gaussian Quadrature Rule will take the following

$$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b-a}{2}t + \frac{a+b}{2}\right)dt \text{ for all } t \in [-1, 1]$$

However, the right side is still an integral. Our goal is to transform it into an equivalent sum. We can do so by replacing the right side of this equation with something that transforms the whole thing into the following

$$\int_a^b f(x)dx \approx \frac{b-a}{2}\sum_{i=1}^{n} w_i f\left(\frac{b-a}{2}x_i + \frac{a+b}{2}\right)$$

Where w_i is/are the weights, which are specific to the number of points n used, and x_i are the coefficients, which are also specific to the number of points used.

Finding Weights and Coefficients

The following rule, if I didn't already mention,

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

is exact for polynomials of degree $2n-1$, as I mentioned in the beginning. This rule has a special name, $Gauss - Legendre\ Quadrature\ Rule$. This rule will only be an accurate approximation to the integral above if $f(x)$ is well-approximated by a polynomial of degree $2n-1$ or less on $[-1,1]$(Wikipedia). Again, w_i are the weights, which are unique to the number of points that we choose to use. While this goal may seem difficult, it actually is not. This can be approached by either using Linear Algebra without Calculus, or using Calculus without Linear Algebra. We will do the latter.

The general rule for finding the weights w_i is given by

$$w_i = \frac{2}{(1-x_i^2)[P_n'(x_i)]^2}$$

where $P_n(x)$ is the $Legendre - Polynomial$. To make this rule useful and how we can find all w_i's and x_i's, we use Calculus.

Let I be the integral representing the area under the curve of some function $f(x)$. This means

$$I = \int_a^b f(x)dx = w_1 f(x_1) + w_2 f(x_2) + \ldots + w_n f(x_n) = \sum_{i=1}^{n} w_i f(x)$$

We then do the following:

$$f(x) = 1 \implies \int_{-1}^{1} 1dx = w_1 f(x_1) + w_2 f(x_2) = 2$$
$$f(x) = x \implies \int_{-1}^{1} xdx = w_1 f(x_1) + w_2 f(x_2) = 0$$
$$f(x) = x^2 \implies \int_{-1}^{1} x^2dx = w_1 f(x_1) + w_2 f(x_2) = \frac{2}{3}$$
$$f(x) = x^3 \implies \int_{-1}^{1} x^3dx = w_1 f(x_1) + w_2 f(x_2) = 0$$

If we clean things up by rearranging, we get

$$w_1 f(x_1) + w_2 f(x_2) = 2$$
$$w_1 f(x_1) + w_2 f(x_2) = 0$$
$$w_1 f(x_1) + w_2 f(x_2) = \frac{2}{3}$$
$$w_1 f(x_1) + w_2 f(x_2) = 0$$

We have four equations and four unknowns. The four unknowns are readily obtainable by performing typical row-operations with Linear Algebra. Upon performing the row-operations, we get that

$$w_0 = w_1 = 1 \text{ and } x_0 = -\frac{1}{\sqrt{3}} \text{ and } x_1 = \frac{1}{\sqrt{3}}$$

What we have actually obtained here is only applicable to two points of evaluation. Usually, we will need more than two points to get great results whenever we are applying the $Gaussian\ Quadrature\ Rule$. We can repeat what we did and generalize things for up to four points. Below are the suitable w_i's and x_i's for a given number of points n.

n = 2

$$w_1 = 1 \text{ and } x_1 = -1/\sqrt{3}$$
$$w_2 = 1 \text{ and } x_2 = 1/\sqrt{3}$$

n = 3

$$w_1 = 5/9 \text{ and } x_1 = -\sqrt{3/5}$$
$$w_2 = 8/9 \text{ and } x_2 = 0$$
$$w_3 = 5/9 \text{ and } x_3 = \sqrt{3/5}$$

n = 4

$$w_1 = (18 - \sqrt{30})/36 \text{ and } x_1 = -\sqrt{525 + 70\sqrt{30}}/35$$
$$w_2 = (18 + \sqrt{30})/36 \text{ and } x_2 = -\sqrt{525 - 70\sqrt{30}}/35$$
$$w_3 = (18 + \sqrt{30})/36 \text{ and } x_3 = \sqrt{525 - 70\sqrt{30}}/35$$
$$w_4 = (18 - \sqrt{30})/36 \text{ and } x_4 = \sqrt{525 + 70\sqrt{30}}/35$$

The list goes on. The bigger the value of n, the better the approximation becomes.

In [65]:

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from math import exp
%matplotlib inline

def p(x, order): # Legendre Polynomials up to p8
    if order == 0:
        return 1
    elif order == 1:
        return x
    elif order == 2:
        return (1/2)*(3*x**2 - 1)
    elif order == 3:
        return (1/2)*(5*x**3 - 3*x)
    elif order == 4:
        return (1/8)*(35*x**4 - 30*x**2 + 3)
    elif order == 5:
        return (1/8)*(63*x**5 - 70*x**3 + 15*x)
    elif order == 6:
        return (1/16)*(231*x**6 - 315*x**4 + 105*x**2 - 5)
    elif order == 7:
        return (1/16)*(429*x**7 - 693*x**5 + 315*x**3 - 35*x)
    elif order == 8:
        return (1/128)*(6435*x**8 - 12012*x**6 + 6930*x**4 - 1260*x**2

def Gauss(f, a, b, n): # for  2 <= n <= 4
    if n == 2:
        wi = np.array([1, 1])
        xi = np.array([-1/3**(.5), 1/3**(.5)])
    elif n == 3:
        wi = np.array([5/9, 8/9, 5/9])
        xi = np.array([-(3/5)**.5, 0, (3/5)**.5])
    elif n == 4:
        c0 = (18 - np.sqrt(30))/36
        c1 = (18 + np.sqrt(30))/36
        c2 = c1
        c3 = c0
        x2 = np.sqrt(525 - 70*np.sqrt(30))/35
        x3 = np.sqrt(525 + 70*np.sqrt(30))/35
        x0 = -x3
        x1 = -x2
        wi = np.array([c0, c1, c2, c3])
        xi = np.array([x0, x1, x2, x3])
    elif n == 5:
        wi = w5
        xi = x5
    elif n == 6:
        wi = w6
        xi = x6
    elif n == 7:
        wi = w7
        xi = x7
    elif n == 8:
        wi = w8
        xi = x8
```

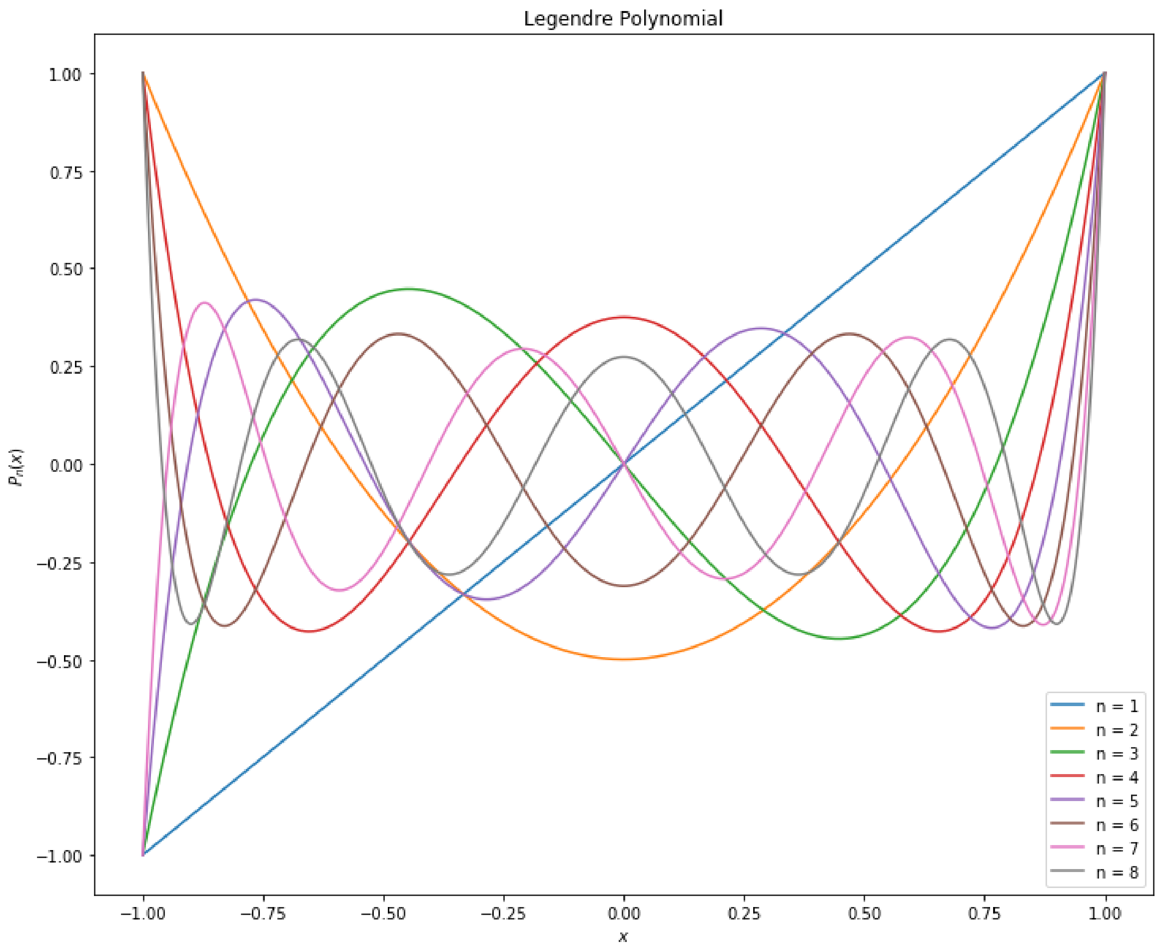
```
    return ((b - a)/2)*sum(wi*f((b - a)/2*xi + (a + b)/2))

def integral(function, lower, upper, n, method = 'Rectangle'):
    Sum = 0
    deltaX = (upper - lower)/n
    if method == 'Rectangle':
        x = np.linspace(lower, upper, n)
        for i in x:
            Sum = Sum + function(i)
        area = deltaX*Sum
        return area
    elif method == 'Trapezoid':
        x1 = lower
        x2 = x1 + deltaX
        while x2 <= upper:
            Sum = Sum + (deltaX/2)*(function(x1) + function(x2))
            x1 = x2
            x2 = x1 + deltaX
        return Sum
    elif method == 'Simpson':
        x1 = lower
        x2 = x1 + deltaX
        x3 = x2 + deltaX
        while x3 <= upper:
            Sum = Sum + (deltaX/3)*(function(x1) + 4*function(x2) + fu
            x1 = x3
            x2 = x1 + deltaX
            x3 = x2 + deltaX
        return Sum
    else:
        print('Choose a method from any of the following: Rectangle, T
        return None

def root(f, initial):
    x0 = initial
    if f(x0) > 0:
        while f(x0) > 0:
            x0 = x0 - 0.00001
    elif f(x0) < 0:
        while f(x0) < 0:
            x0 = x0 - 0.00001
    return x0

def derivative(f, x):
    h = 0.000000000000001
    return (f(x + h) - f(x))/h

def wi(f, xi):
    den = (1 - xi**2)*(derivative(f, xi))**2
    return 2/den
wi = np.vectorize(wi)
root = np.vectorize(root)
```

Legendre Polynomial

Below is a graph of the Legendre Polynomial in the interval $-1 \leq x \leq 1$.

In [4]:

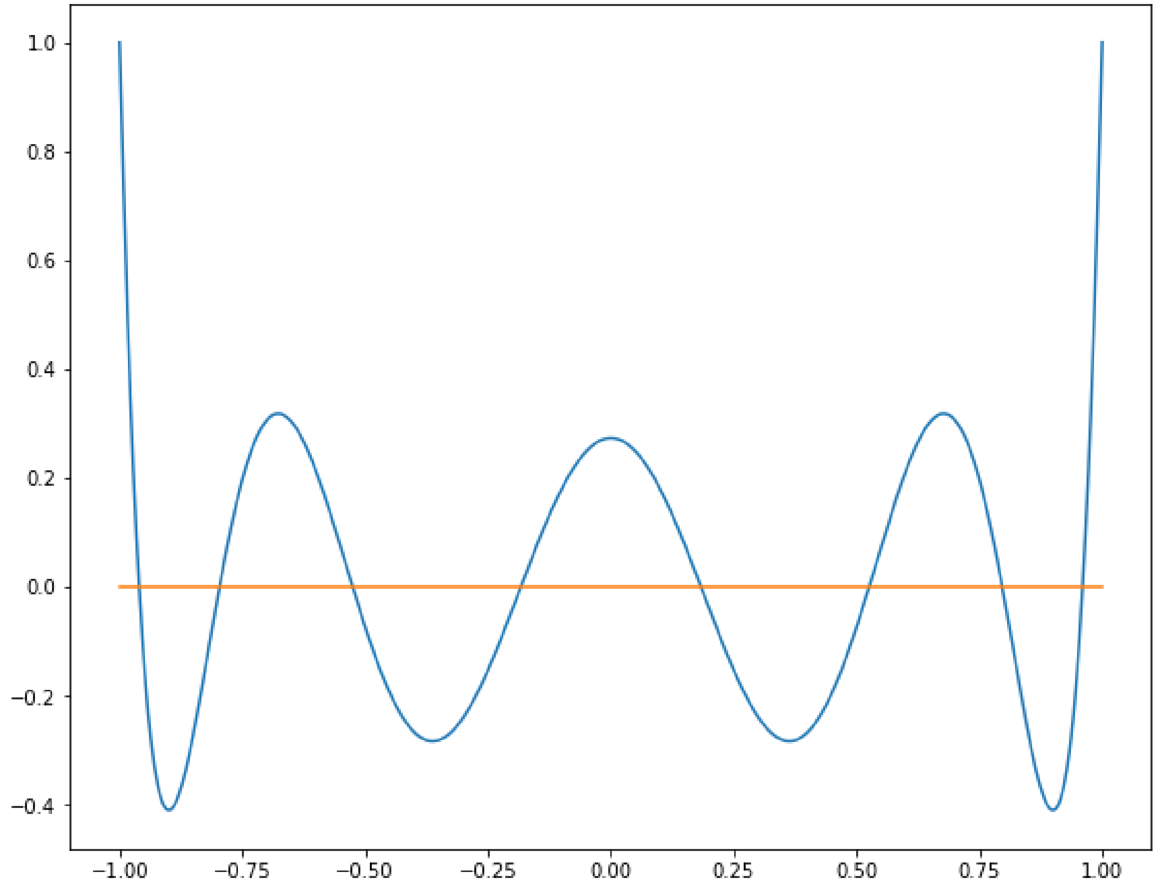
```
x = np.linspace(-1, 1, 1000)
Labels = ['n = 0', 'n = 1', 'n = 2', 'n = 3', 'n = 4', 'n = 5', 'n = 6'
plt.figure(figsize = (12, 10))
for i in range(1, 9, 1):
    plt.plot(x, p(x, i), label = Labels[i])
plt.xlabel('$ x $')
plt.ylabel('$ P_n(x) $')
plt.title('Legendre Polynomial')
plt.legend()
plt.show()
```



Obtaining More Weights for $5 \leq n \leq 8$.

In [13]:

```
plt.figure(figsize = (10, 8))
plt.plot(x, p(x, 8))
plt.plot([-1, 1], [0, 0])
plt.show()
```



In [29]:

```
# I will pass these values back to my p(x, order) function
g5 = lambda x: (1/8)*(63*x**5 - 70*x**3 + 15*x)
g6 = lambda x: (1/16)*(231*x**6 - 313*x**4 + 105*x**2 - 5)
g7 = lambda x: (1/16)*(429*x**7 - 693*x**5 + 315*x**3 - 35*x)
g8 = lambda x: (1/128)*(6435*x**8 - 12012*x**6 + 6930*x**4 - 1260*x**2
x5 = root(g5, [1, .6, .2, -.5, -.8])
x6 = root(g6, [1, .75, .3, -.1, -.6, -.8])
x7 = root(g7, [1, .8, .5, .1, -.26, -.70, -.8])
x8 = root(g8, [1, .85, .6, .25, -.1, -.4, -.75, -.88])
```

In [30]:
```
print('x5 =', x5)
print('x6 =', x6)
print('x7 =', x7)
print('x8 =', x8)
```

```
x5 = [ 9.06170000e-01  5.38460000e-01 -5.92188821e-14 -5.38470000e-01
 -9.06180000e-01]
x6 = [ 0.92217  0.66912  0.23842 -0.23843 -0.66913 -0.92218]
x7 = [ 9.49100000e-01  7.41530000e-01  4.05840000e-01 -9.99999999e-06
 -4.05850000e-01 -7.41540000e-01 -9.49110000e-01]
x8 = [ 0.96028  0.79666  0.52553  0.18343 -0.18344 -0.52554 -0.79667 -0.960
29]
```

In [64]:
```
w5 = [2/((1 - i**2)*(derivative(g5, i))**2) for i in x5]
w6 = [2/((1 - i**2)*(derivative(g6, i))**2) for i in x6]
w7 = [2/((1 - i**2)*(derivative(g7, i))**2) for i in x7]
w8 = [2/((1 - i**2)*(derivative(g8, i))**2) for i in x8]
```

Gaussian Quadrature, scipy, and Simpson's Rule(Comparison)

Let us test the three methods on the integral

$$\int_1^2 \left(2x + \frac{3}{x}\right)^2 dx$$

In [72]:
```
f = lambda x: (2*x + 3/x)**2
from scipy.integrate import quad
print('scipy.integrate.quad:', quad(f, 1, 2)[0])
print('Gaussian Quadrature with 4 points:', Gauss(f, 1, 2, 4))
print("Simpson's Rule:", integral(f, 1, 2, 1000, method = 'Simpson'))
```

```
scipy.integrate.quad: 25.833333333333332
Gaussian Quadrature with 4 points: 25.833289661396922
Simpson's Rule: 25.833333333333993
```

Speed Comparison

In [74]:
```
%timeit quad(f, 1, 2)[0]
%timeit Gauss(f, 1, 2, 4)
%timeit integral(f, 1, 2, 1000, method = 'Simpson')
```

```
9.61 µs ± 330 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
18.3 µs ± 611 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
653 µs ± 46.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

The Gaussian Quadrature method is a little less than half(1.9 times) as fast as the built-in function $quad$, but it is still much faster than $Simpson's\ Rule$ at almost 36 times the speed of Simpson's Rule, and considering that we didn't have to use as many points as we did using Simpson's Rule and still get an accurate answer, it goes to show just how good the Gaussian Quadrature method is. Let us try another example.

$$\int_{0.1}^{1.3} 5xe^{-2x}\,dx$$

In [78]:

```
f = lambda x: 5*x*np.exp(-2*x)
print('scipy.integrate.quad:', quad(f, 0.1, 1.3)[0])
print('Gaussian Quadrature with 4 points:', Gauss(f, 0.1, 1.3, 4))
print("Simpson's Rule:", integral(f, 0.1, 1.3, 1000, method = 'Simpson'
```

```
scipy.integrate.quad: 0.8938650276524702
Gaussian Quadrature with 4 points: 0.8938681930382849
Simpson's Rule: 0.8927046475426311
```

In [80]:

```
%timeit quad(f, .1, 1.3)[0]
%timeit Gauss(f, .1, 1.3, 4)
%timeit integral(f, 0.1, 1.3, 1000, method = 'Simpson')
```

```
23.9 µs ± 924 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
17.7 µs ± 122 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
1.82 ms ± 43.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```