HW3: Ch19.5 Numerical Integration and Differentiation

Bailey Williams

For this assignment I will use Python programs to run the Trapezoid rule, as well as Gaussian quadrature for integration. I will additionally compute derivatives using several different formulas.

Problem 1: The Trapezoid Rule

In the Ch19.5 Part 1 notes, two Python programs were given for the Left Sum Rule, the Midpoint Rule, and Simpsons' Rule. The first program was a traditional program with one or more `for` loops, as well as a percent relative error (PRE) calculation using the `quad` command from the `scipy` package. The second program was a shorter vectorized version. In this problem I will write up both styles of program for the **Trapezoid Rule**. The traditional program for the **Trapezoid Rule** is displayed in *Case 1*, while the vectorized verison is displayed in *Case 2*. This code will be run using the same function $f(x) = e^{-x^2}$ from $[0, 1]$, used in our class notes.

Trapezoid Rule Formulas

Use the area formula of trapezoid for each panel, then add them up.

$$T_n = h/2[f(x_0) + 2f(x_1) + 2f(x_2) + \ldots + 2f(x_{n-1}) + f(x_n)]$$

$$\int_a^b f(x)\,dx = \sum_{k=0}^{n-1} \frac{f(x_k)+f(x_{k+1})}{2} h, x_k = a + hk, h = \frac{b-a}{n}$$

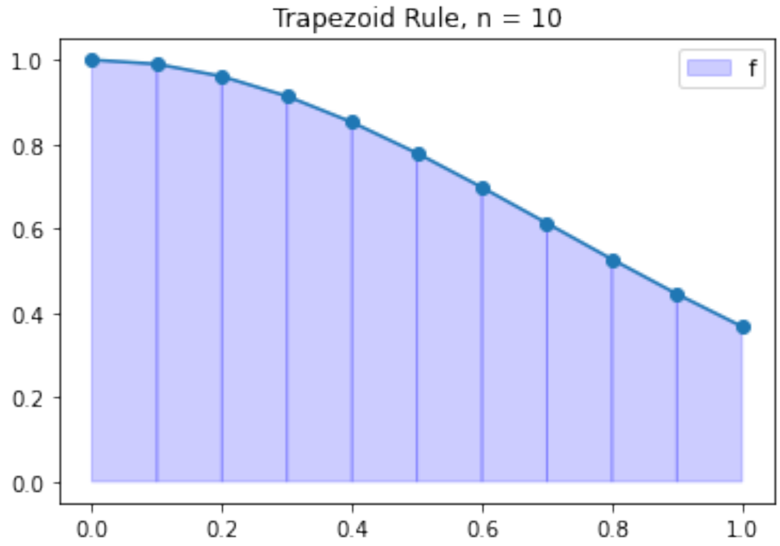
Case 1: Regular Version of Trapezoid Rule

In [1]:

```
#Trapezoid Rule

import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import quad

# Define function to integrate
def f(x):
    return np.exp(-x**2)

# Implementing trapezoidal method
def trapezoidal(x0,xn,n):
    # Step size
    h = (xn - x0) / n
```

```
    # Finding sum
    integration = f(x0) + f(xn)

    for i in range(1,n):
        k = x0 + i*h
        integration = integration + 2 * f(k)

    # Finding final integration value
    integration = integration * h/2

    return integration

# Get result of Trapezoid Rule
result = trapezoidal(0, 1, 10)
print("Integration result by Trapezoidal method is: %0.6f" % (result) )

a = 0
b = 1
n = 10

# Compute Percent Relative Error (PRE)
Q = quad(f,a,b)
Q = Q[0]
PRE = (Q - result)/Q*100 #PRE
print('S = %.6f, Q = %.6f, PRE = %.6f' % (result,Q,PRE))

x = np.linspace(a, b, n+1)
y = f(x)

# Plots individual trapezoids based on n
for i in range(n):
    xs = [x[i],x[i],x[i+1],x[i+1]]
    ys = [0,f(x[i]),f(x[i+1]),0]
    plt.fill(xs,ys,'b',edgecolor='b',alpha=0.2)

# Plot commands
plt.plot(x,y, '-o')
plt.legend(('f'),loc = 0)
plt.title('Trapezoid Rule, n = {}'.format(n))
plt.show()
```

```
Integration result by Trapezoidal method is: 0.746211
S = 0.746211, Q = 0.746824, PRE = 0.082126
```



Case 2: Vectorized Version of Trapezoid Rule

In [2]:

```
# Vectorized Trapezoid Rule

a = 0
b = 1
n = 10

# Function to integrate
f = lambda x: np.exp(-x**2)

def trap(f,a,b,n):
    x = np.linspace(a,b,n+1) # Makes n subintervals
    y = f(x)
    right = y[1:] # Right endpoints
    left = y[:-1] # Left endpoints
    dx = (b - a)/n
    T = (dx/2) * np.sum(right + left)
    return T
result = trap(f,a,b,n)
print("Integration result by Trapezoidal method is: %0.6f" % (result) )
```

```
Integration result by Trapezoidal method is: 0.746211
```

Discussion of Results

The **Trapezoidal Rule** evaluates the area under the curve by dividing the total area into small trapezoids rather than rectangles. This rule is typically used in the numerical analysis process, and is more accurate than that of the Left Sum, Right Sum, or Midpoint Rules. The above program demonstates finding the area under the curve using the **Trapezoidal Rule** in both regular and vectorized functions. The vectorized function is more simple and compact in use than the standard. The graph shows 10 trapezoids created in order to calculate $f(x) = e^{-x^2}$ from $[0, 1]$, which resulted in finding the area to be 0.746211. When compared to an actual of 0.746824, the result is a *Percent Relative Error* of 0.082126. Therefore, the **Trapezoidal Rule** seems to be a close compution of the area under the curve of $f(x)$.

Problem 2

For $f(x) = \cos(x)$ on $[a, b]$, I will use the Python programs given in the Ch19.5 Part 2 notes to perform the **Gauss Quadrature** for the following cases.

Case 1:

Two-point quadrature for $[a, b] = [-1, 1]$.

In [3]:
```
from scipy.special.orthogonal import p_roots

[x,w] = p_roots(2)
print('x = ',x, 'w = ',w)
```

```
x =  [-0.57735027  0.57735027] w =  [1. 1.]
```

In [4]:
```
# n-pt GQ on [-1,1]
def GQ1(f,n):
    [x,w] = p_roots(n)
    G = sum(w*f(x))
    return G

f = lambda x: np.cos(x)

r = GQ1(f,2)
print(r)
```

```
1.6758236553899863
```

In [5]:
```
# n-pt GQ on [a,b]
def GQ2(f,n,a,b):
    [x,w] = p_roots(n)
    G = 0.5*(b-a)*sum(w*f(0.5*(b-a)*x+0.5*(b+a)))
    return G

r2 = GQ2(f,2,-1,1)
print(r2)
```

```
1.6758236553899863
```

Case 2:

Three-point quadrature for $[a,b] = [0,2]$.

In [6]:
```
r3 = GQ2(f,3,0,2)
print(r3)
```

```
0.9093306976211126
```

Checking Answer with Desmos

1

$$\int_{-1}^{1} \cos x \, dx$$

$= 1.68294196962$

2

$$\int_{0}^{2} \cos x \, dx$$

$= 0.909297426826$

Discussion of Results

The program and results above show how the **Guass Quadrature** achieves high accuracy for very few nodes (i.e $2-3$). In our notes it was discussed that for N data points, the **Guass Quadrature** is exact for polynomials of degree $2N-1$ or less on $[-1,1]$ which is depicted in the solutions that I found. You can see above, my answers compared to that of those calulated in *Desmos* which shows just how close or exact the **GQ** was.

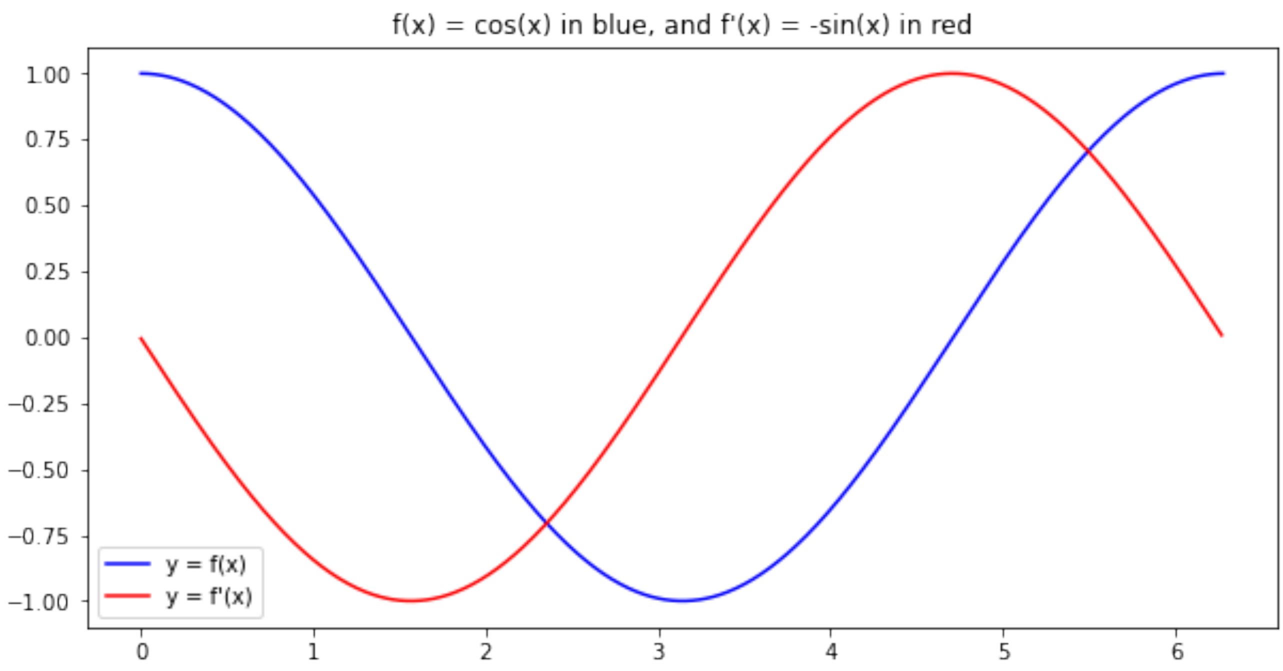
Problem 3

For $f(x) = \cos(x)$ on $[0, 2\pi]$, I will adapt the program in the Ch19.5 Part 3 notes that incorporates the `np.diff` command to perform **forward differences**.

Forward Difference Code:

In [7]:

```
import numpy as np
import matplotlib.pyplot as plt
h = 0.01
x = np.arange(0,2*np.pi,h)
f = np.cos(x)
fp = np.diff(f)/h
plt.figure(figsize=(10,5))
plt.plot(x,f,'b',label="y = f(x)")
plt.plot(x[:-1],fp,'r',label="y = f'(x)")
plt.title("f(x) = cos(x) in blue, and f'(x) = -sin(x) in red")
plt.legend(loc='best')
plt.show()
```



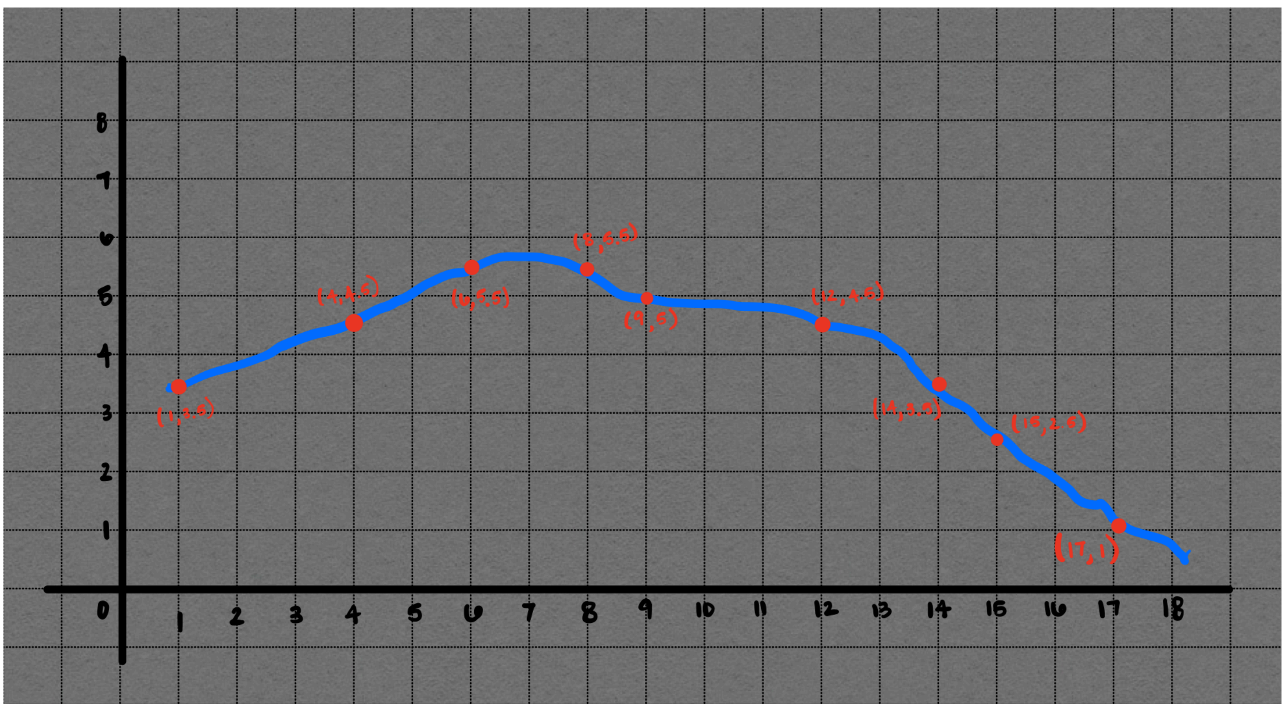
Discussion of Results

Forward differences are useful in solving ordinary differential equations by single-step predictor-corrector methods. In the program above, $f(x) = \cos(x)$ and the command *np.diff* is used to help find $f'(x)$. The graph provides assurance that the $f'(x)$ found is correct due to when the slope of $f(x)$ is increasing the graph of $f'(x)$ is positive and when $f(x)$ is decreasing $f'(x)$ is negative. The derivative of $f(x) = \cos(x)$ is commonly known to be $f'(x) = -\sin(x)$ which appears to be the curve shown for $f'(x)$.

Problem 4

In the Ch19.5 Part 3 notes, a **cubic spline** was fitted to the hand profile data, and the **derivative** of the spline was computed. I will adapt and run this program for my hand profile data, producing the graphs of f and f' on the same axes (with grid lines) as in our class notes.

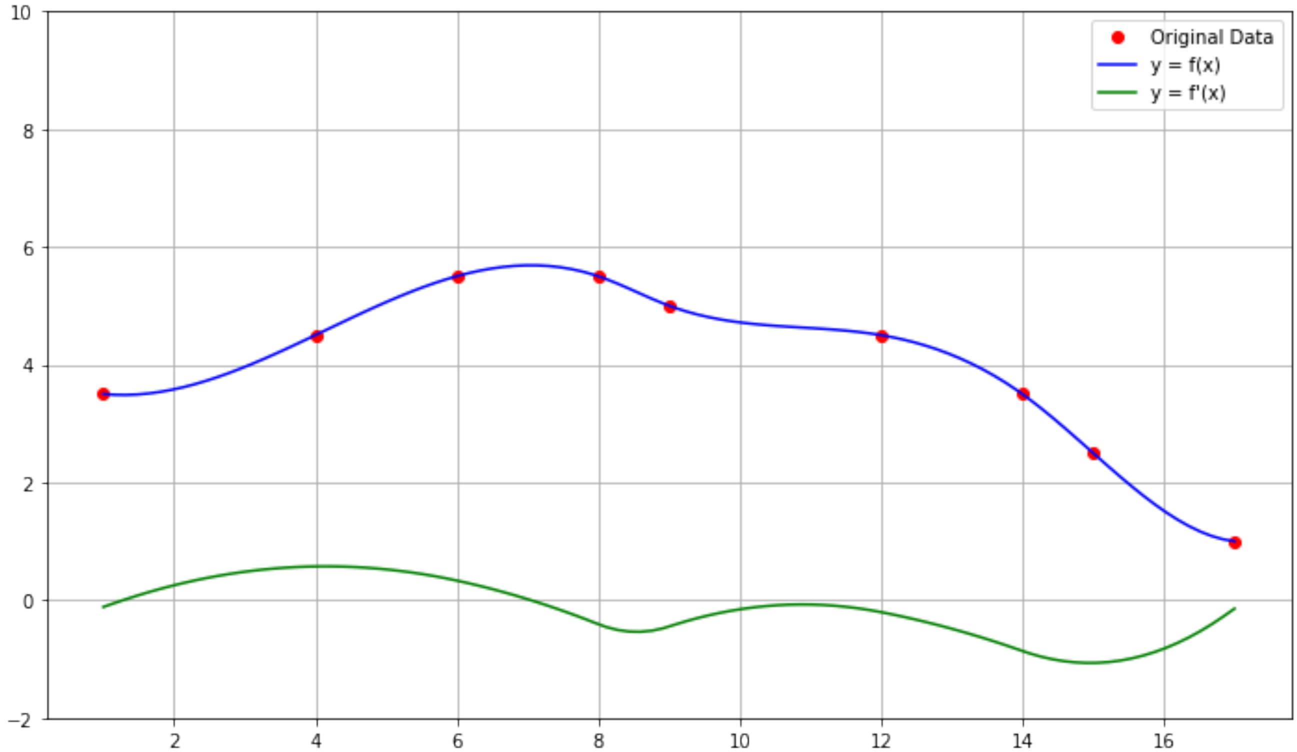
Original Hand Profile Graph



Python Program

In [8]:

```
import numpy as np
from scipy.interpolate import interp1d
import matplotlib.pyplot as plt

# fwd difference function
def fwd(f,a,h=0.01):
    fp = (f(a+h)-f(a))/h
    return fp

# ctr difference function
def sym(f,a,h=0.0001):
    fp = (f(a+h)-f(a-h))/(2*h)
    return fp

# Enter Data
xdata = [1,4,6,8,9,12,14,15,17]
ydata = [3.5,4.5,5.5,5.5,5,4.5,3.5,2.5,1]
n = len(xdata)

# Command for cubic spline polynomial p(x)
f = interp1d(xdata,ydata,kind='cubic')
N = 100
x = np.linspace(xdata[0],xdata[n-1],N)
y = f(x)

h = 0.01
fp = np.zeros(N)
# fwd difference at 0
fp[0] = fwd(f,x[0])
# bwd difference at 99
fp[N-1] = fwd(f,x[N-1],-h)
# ctr difference at all other nodes
for k in range(1,N-1):
    fp[k] = sym(f,x[k],h)


# Plot commands
plt.figure(figsize=(12,7))
plt.plot(xdata,ydata,'ro',label='Original Data')
plt.plot(x,y,'b',label='y = f(x)')
plt.plot(x,fp,'g',label="y = f'(x)")
plt.legend()
plt.grid(True)
plt.ylim(-2,10)
plt.show()
```



Discussion of Results

As seen in the graph above, the python calculated **cubic spline** interpolates the data points which provides a smooth trend in the data without large ocillations between data points. This spline provides a reasonably accurate representation of between the data points, but not outside the range of the data points (extrapolation). The beauty of the cubic spline interpolant is how well it approximates a function with little error. From this cubic spline function I am able to use forward, backward, and center differences to find the **derivative** of my hand data. I can see that the $f'(x)$ found is accurate due to when the slope of $f(x)$ is increasing $f'(x)$ is positive and when $f(x)$ is decreasing $f'(x)$ is negative.

Problem 5

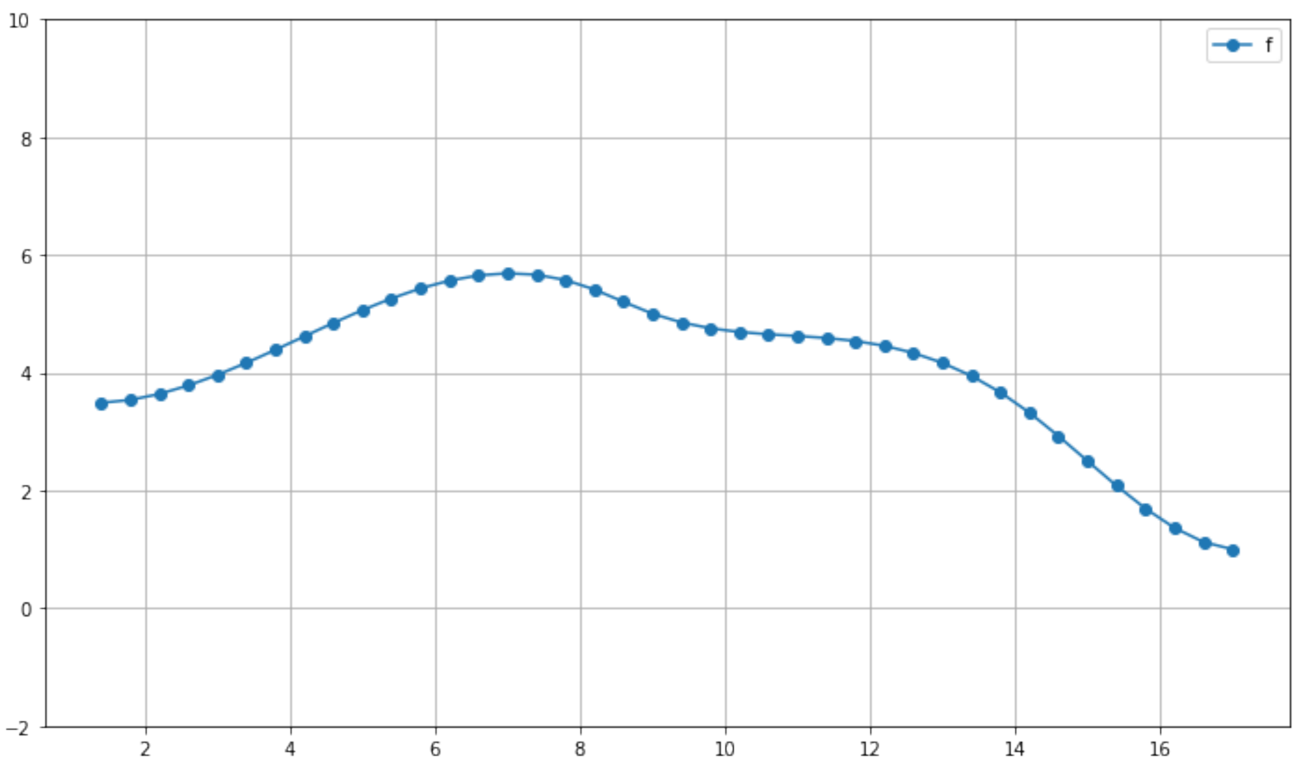
As in the previous problem, I will fit a cubic spline f to my hand profile data. Then I will use the function f to compute the **right sum value** for the area under my hand profile curve. To do this, I will adapt the vectorized left sum program from the Ch19.5 Part 1 notes (the cubic spline f will replace the `lambda` function). To run this program I will have a separate command in the form of `rightsum(a,b,40)` , where $a = x_0$ and $b = x_{n-1}$ represent my data, using $n = 40$ rectangles.

Python Vectorized Right Sum Program

In [9]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

def rightsum(a,b,n):
    f = interp1d(xdata,ydata,kind='cubic')
    nodes = np.linspace(a,b,n+1)
    xn = nodes[1:n+1]
    fn = f(xn)
    h = (b-a)/n
    S = sum(fn)*h
    print(S)
    Q = quad(f,a,b)
    Q = Q[0]
    PRE = (Q-S/Q*100)
    print('S = %.6f, Q = %.6f, PRE = %.6f' % (S,Q,PRE))
    plt.figure(figsize=(12,7))
    plt.plot(xn,fn,'-o')
    plt.legend(('f'),loc=0)
    plt.grid(True)
    plt.ylim(-2,10)
    plt.show()

rightsum(1,17,40)
```

```
66.0564460974134
S = 66.056446, Q = 66.556862, PRE = -32.691277
```



Discussion of Results

Using the function of the **cubic spline** found earlier I am able to apply the **Right Sum Rule** as a means of approximating the area under the curve ($f'(x)$). According to what the **Right Sum Rule** calulated $f'(x) = 66.0564$ compared to the actual 66.5569 making the *Percent Relative Error* -32.6913. By looking at the graph and at the *PRE* the **Right Sum Rule** can give us an idea of what the area under the curve is approximately equivalent to but it is definitley not the most accurate.

In [ ]:

In [ ]: