Lesson 3

Ryan Branagan

January 29, 2019

Problem 1

```
#Ryan Branagan
#Collaborators: N/A
#Branagan_hw3_p1.py
#1/22/19

import numpy as np
import pylab as p

#Input
n=10
f=np.sin
a=0
b=np.pi/2
truans=1

#Define the left hand, right hand, and mid point sum
def leftpoint(f,a,b,N):
    h = (b-a)/N

    xax = np.array([])
    for i in range(N):
        y = a + i*h
        xax = np.append(xax,y)

    return np.sum(f(xax) * h)

def midpoint(f,a,b,N):
    h = (b-a)/N

    xax = np.array([])
    for i in range(N):
        y = a+(i+1/2)*h
        xax = np.append(xax,y)

    return np.sum(f(xax) * h)

def rightpoint(f,a,b,N):
    h = (b-a)/N

    xax = np.array([])
    for i in range(N):
        y = a+(i+1)*h
        xax = np.append(xax,y)

    return np.sum(f(xax) * h)

#Make a list of Ns
Nlist = []
s = 1
for i in range(n):
    s = s * 2
    Nlist.append(s)

#Make a list of the sums for each N in the Nlist
ansMP = np.array([])
for N in Nlist:
    y = midpoint(f,a,b,N)
    ansMP = np.append(ansMP,y)

ansLHS = np.array([])
for N in Nlist:
    y = leftpoint(f,a,b,N)
    ansLHS = np.append(ansLHS,y)

#Calculate errors
errMP = (ansMP-truans)/truans

errLHS = (ansLHS-truans)/truans

#Plot the answer for each N
#p.figure(1)
#p.plot(Nlist,ansMP,'o-',label="Midpoint")
#p.plot(Nlist,ansLHS,'o-',label="Left Hand Side")
#p.legend()
```

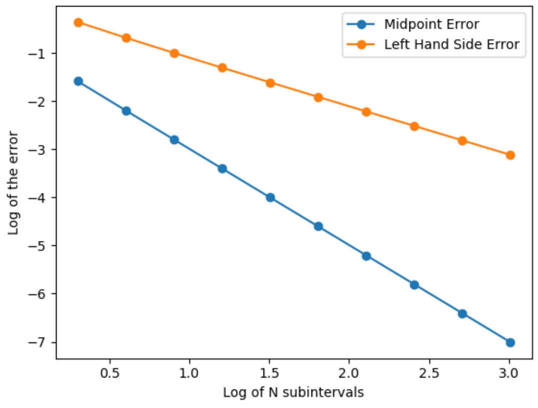

Figure 1: A log-log plot of error vs N

```
#Plotting log-log plot for error
p.figure(2)
p.plot(np.log10(Nlist),np.log10(np.abs(errMP)),'o-',label="Midpoint Error")
p.plot(np.log10(Nlist),np.log10(np.abs(errLHS)),'o-',label="Left Hand Side Error")
p.legend()
p.xlabel('Log of N subintervals')
p.ylabel('Log of the error')
p.show()
```

The routine I wrote takes in some parameters, defines the Riemann sum numerical integration techniques, then outputs a graph of the error vs N rectangles. The error is proportional to some power and by looking at the graph, the power is negative two for the midpoint rule. When comparing which method is better, the midpoint rule is much better. Since the slope is double the left hand method which is negative one and is representing a power, it is an order of magnitude greater.

Problem 2

```
#Ryan Branagan
#Collaborators: N/A
#Branagan_hw3_p2.py
#1/23/19

import numpy as np
import pylab as p
```

```
#Define a function for our integral we want to integrate
def sinx2(x):
    return np.sin(x**2)

#Input
n=6
f=sinx2
a=0
b=100

#Define numerical integration techniques
def leftpoint(f,a,b,N):
    h = (b-a)/N

    xax = np.array([])
    for i in range(N):
        y = a + i*h
        xax = np.append(xax,y)

    return np.sum(f(xax) * h)

def midpoint(f,a,b,N):
    h = (b-a)/N

    xax = np.array([])
    for i in range(N):
        y = a+(i+1/2)*h
        xax = np.append(xax,y)

    return np.sum(f(xax) * h)

def rightpoint(f,a,b,N):
    h = (b-a)/N

    xax = np.array([])
    for i in range(N):
        y = a+(i+1)*h
        xax = np.append(xax,y)

    return np.sum(f(xax) * h)

#Make a list of Ns
Nlist = []
s = 2048
```

```
for i in range(n):
    s = s * 2
    Nlist.append(s)

#Make a list of the sums for each N in the Nlist
ansMP = np.array([])
for N in Nlist:
    y = midpoint(f,a,b,N)
    ansMP = np.append(ansMP,y)

#Find Error
print('Sorry this will take a little bit (less than a minute)')
truans = midpoint(f,a,b,200000)
errMP = (ansMP-truans)/truans

#Print Nlist and answers
print(Nlist)
print(ansMP)

#Plot error vs N to find what Ns are needed to have a certain error
p.plot(np.log10(Nlist),np.log10(np.abs(errMP)),'o-',label="Midpoint Error")
#p.plot(Nlist,ansMP,'o-')
p.show()
```

I mostly reused code from the previous question to complete this one. This routine does almost exactly what the first one does, defines the function we are trying to integrate then prints out values for each method we are observing and graphs the error with respect to a reference value using the maximum amount of sub-intervals I could use. The below table contains a few N values.

4096	0.64469837
8192	0.63284394
16384	0.63172688
32768	0.63149263
65536	0.63143645
131072	0.63142254

I also plotted the error in order to estimate for what values of N got a certain precision. The results were;

3 Decimal Places	12,500
4 Decimal Places	36,000
5 Decimal Places	100,000

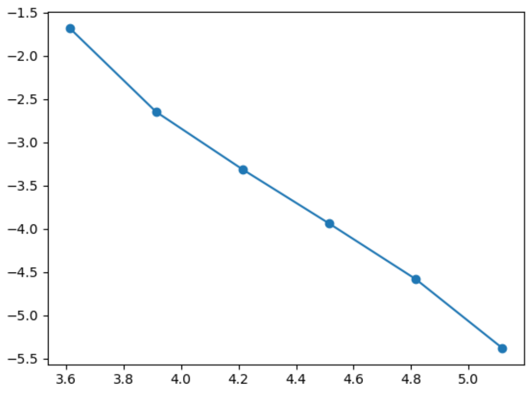

Figure 2: Log of error vs Log of N

Problem 3

```
#Ryan Branagan
#Collaborators: N/A
#Branagan_hw3_p3.py
#1/24/19

import numpy as np

#Define a function for our integral we want to integrate
def cosq(x):
    return np.cos(x**2-x)

#Input
n=15
f=cosq
a=-1
b=1
j=30000

#Define numerical integration techniques
def leftpoint(f,a,b,N):
    h = (b-a)/N

    xax = np.array([])
    for i in range(N):
        y = a + i*h
        xax = np.append(xax,y)
    return np.sum(f(xax) * h)

def midpoint(f,a,b,N):
    h = (b-a)/N

    xax = np.array([])
    for i in range(N):
        y = a+(i+1/2)*h
        xax = np.append(xax,y)

    return np.sum(f(xax) * h)

def rightpoint(f,a,b,N):
    h = (b-a)/N

    xax = np.array([])
    for i in range(N):
        y = a+(i+1)*h
        xax = np.append(xax,y)

    return np.sum(f(xax) * h)

def trap(f,a,b,N):
    return 0.5*(leftpoint(f,a,b,N)+rightpoint(f,a,b,N))

print(leftpoint(f,a,b,j))
print(midpoint(f,a,b,j))
print(trap(f,a,b,j))

#Results
print('The number of subintervals needed to have an error of 10^-4 for Left Hand Rule is
print('The number of subintervals needed to have an error of 10^-4 for Mid-Point Rule is
print('The number of subintervals needed to have an error of 10^-4 for Trapezoid Rule is
```

For this problem I mostly reused code from previous problems. I manually went through different values of N to find when each method was accurate to 4 decimal places. My results were;

Left Hand Sum	20,000
Midpoint Rule	124
Trapezoid rule	114

Problem 4

```
#Ryan Branagan
#Collaborators: Jack Featherstone
#Branagan_hw3_p4.py
#1/24/19

import numpy as np
import pylab as p

#Define a function for our integral we want to integrate
def erf(x):
    return (2/np.sqrt(np.pi))*np.exp(-(x**2))

#Input
n=15
f=erf
a=0
b=3
j=300

#Define numerical integration techniques
def leftpoint(f,a,b,N):
    h = (b-a)/N

    xax = np.array([])
    for i in range(N):
        y = a + i*h
        xax = np.append(xax,y)

    return np.sum(f(xax) * h)

def midpoint(f,a,b,N):
    h = (b-a)/N

    xax = np.array([])
    for i in range(N):
        y = a+(i+1/2)*h
        xax = np.append(xax,y)

    return np.sum(f(xax) * h)

def rightpoint(f,a,b,N):
    h = (b-a)/N
    xax = np.array([])
    for i in range(N):
        y = a+(i+1)*h
        xax = np.append(xax,y)

    return np.sum(f(xax) * h)

def trap(f,a,b,N):
    return 0.5*(leftpoint(f,a,b,N)+rightpoint(f,a,b,N))

def simp(f,a,b,N):
    return (trap(f,a,b,N)/3)+((2*midpoint(f,a,b,N))/3)

#An array of "x"s
x = np.linspace(a,b,j)

#Make an array of answers
ans = np.array([])
for b in x:
    y = simp(f,a,b,j)
    ans = np.append(ans,y)

#Plotting
p.plot(x,ans,'o-')
p.show()
```

At first I was having a lot of trouble with this problem because I was tired and not really thinking straight but with some help from Jack realized what I was doing. At first I didn't even use an integration techniques at all and was just evaluating the error function as if it wasn't an integral. Then, I fixed it and used Simpson's rule to correctly evaluate the error function.

Problem 5

```
#Ryan Branagan
#Collaborators: N/A
#Branagan_hw3_p5.py
#1/24/19

from scipy.integrate import quad

def square(x):
    return x**2
```

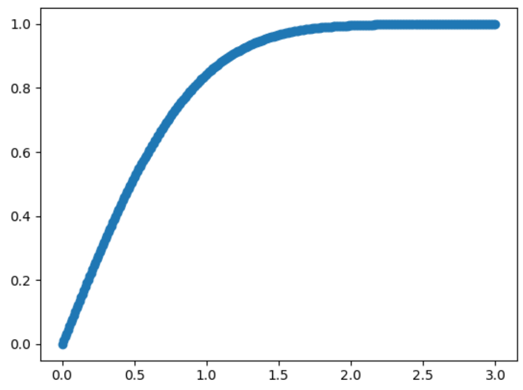

Figure 3: The error function evaluated using Simpson's Rule for upper bound values ranging from 0 to 3.

```
print(quad(square,0,6))
```

This problem was very simple. From this problem I can see that the quad method is much more powerful and efficient than any of the methods we used.