```
import scipy.integrate as integrate
import numpy as np
import matplotlib.pyplot as plt
# ==============================================================================
# Problem 0
# ==============================================================================
#First, define the function we wish to analyze
def f(x):
    return np.exp(-x)*np.sin(10*x)
#All steps below pertain to plotting said function
N=100
a = -1.0
b = 1.0
L = np.linspace(a,b,N)
y = np.zeros(N)
for j in np.arange(N):
    y[j]=f(L[j])
plt.figure(1)
plt.plot(L,y,'b-',linewidth=1)
plt.title('f(x)=exp(-x)sin(10*x)')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.savefig('C:/Users/Parma_Shon/Desktop/Math 514/HW//HW5/figure_1.png',
bbox_inches = 'tight')
#plt.show()
# ==============================================================================
# Problem 1
# ==============================================================================
#defining the composite trapezoid rule using integrate's built-in trapz fnctn
def CompTrapRule(a,b,n):
    x = np.linspace(a,b,n+1)
    h = float(b-a)/n
    return integrate.trapz(f(x),x,h)
# ==============================================================================
# Problem 2
# ==============================================================================
#Same as above except need 2n+1 points & use integrate's built-in simps fnctn
def CompSimpRule(a,b,n):
    x = np.linspace(a,b,2*n+1)
    h = float(b-a)/2*n
    return integrate.simps(f(x),x,h,even = 'avg')
# ==============================================================================
# Problem 3
# ==============================================================================
#Same as both of above, except by nature of the quadrature points we need
#not make a list of equally spaced points...fixed_quad finds them for us!
#Note: needed to return the 0th element as fixed_quad returns an array of two
#objects and we need only the first object, of type numpy.float64
def Gauss(a,b,n):
    return integrate.fixed_quad(f,a=a,b=b,n=n)[0]
# ==============================================================================
# Problem 4
# ==============================================================================
#Here we define a relative error function to compactify our code (here we use
#The 'exact' value of I_Star to 15 digits calculated via Mathematica
def err(I):
    I_Star = -0.178639805625499
    return np.abs((I-I_Star)/I_Star)
#Just assigning each quadrature result to a dummy variable...
a = CompTrapRule(-1.0,1.0,100)
b = CompTrapRule(-1.0,1.0,200)
```

```
c = CompSimpRule(-1.0,1.0,100)
d = CompSimpRule(-1.0,1.0,200)
e = Gauss(-1.0,1.0,10)
g = Gauss(-1.0,1.0,20)
#...and here we find the relative error for each calculation and print them
print err(a) ,'\n', err(b) ,'\n', err(c)
print err(d) ,'\n', err(e) ,'\n', err(g)
# ============================================================================
# Problem 5
# ============================================================================
#For the Trapezoidal Rule, we refer to eqn (7.16) in Sueli & Mayers and find:
#|E(f)| -> 1/4|E(f)| for n -> 2n
#For Simpson's Rule, we refer to eqn (7.18) and find:
#|E(f)| -> 1/16|E(f)| for n -> 2n
#Finally for Gaussian quadrature, we refer to eqn (10.18) in text:
#|E(f)| -> (2n+2)!/(4n+2)!|E(f)| for n -> 2n
#
#Onto the actual results:
#Trapezoidal Rule: n =100 gave E = 0.0039962371075 while n = 200 gave
#E = 0.000998495271808 Their ratio is 0.249859 ~ 0.25 = 1/4
#Simpson's Rule: n = 100 gave E = 7.52006756979e-07 while n = 200 gave
#E = 4.69552635179e-08 Their ratio is 0.0624399 ~ 0.0625 = 1/16
#Gaussian: n = 10 gave E = 0.000231412054226 and n = 20 gave
#E = 9.47767551932e-15 Their ratio is 4.09558e-11 which is nearly vanishingly
#small. Thus all the quadrature rules give exact error results w.r.t. the
#theoretical values
# ============================================================================
# Problem 6
# ============================================================================
E = np.zeros(20)
n = np.linspace(1,20,20)
for j in np.arange(20):
    E[j]=np.log10(err(Gauss(-1.0,1.0,j+1)))
plt.figure(2)
plt.plot(n,E,'bo')
plt.title('log10 of relative error for Gaussian Quadrature as a function of n')
plt.xlabel('n')
plt.ylabel('log10(RelError_Gauss)')
plt.savefig('C:/Users/Parma_Shon/Desktop/Math 514/HW//HW5/figure_2.png',
bbox_inches = 'tight')
#plt.show()

#The plot of the log of the relative error suggests that as a function of n,
#Gaussian quadrature reduces the error rapidly for n in the range ~4-16. For n
#smaller and larger than this range the errors are approximately constant,
#which seems rather surprising, as one may expect the errors (at least for
#larger and larger n) to decrease ever more rapidly. We must wonder what would
#happen if we plotted the relative errors for much higher order Gaussian
#Quadrature.
```