MACS_II_PS2

January 21, 2019

0.1 MACS30150 Problem Set 2

0.2 Dr. Richard Evans

0.2.1 Submitted by Junho Choi

Let us first import the following packages needed for this problem set.

```
In [2]: import numpy as np
        import scipy.stats as sts
        import scipy.optimize as opt
        import math
        import sympy as sp
        import time
        from matplotlib import pyplot as plt
```

0.2.2 Exercise 1. Numerical Differentiation

Problem 1 For this question, I define functions "fn" and "fn_deriv" to represent $f(x) = (\sin(x) + 1)^{(\sin(\cos(x))}$ and its derivative; this is shown below.

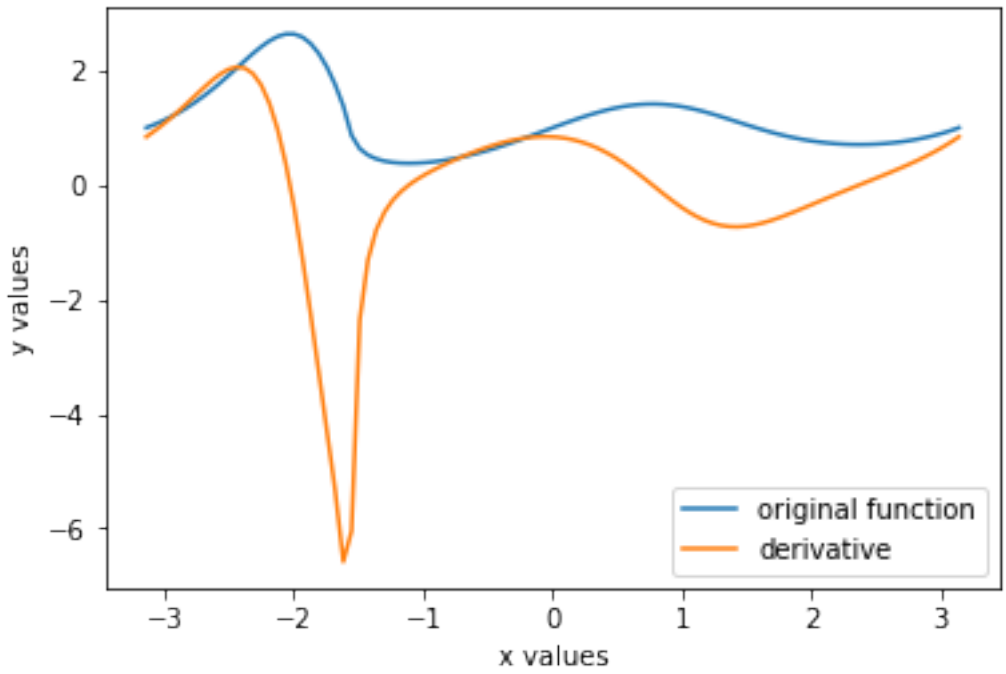
```
In [3]: def fn(x, value_or_not, value = None):
            '''
            representing the function f mentioned above.

            inputs:
            - x: vector of strings to represent the input variable
            - value_or_not: False if value of the function needs not
              to be evaluate at some value, True is needs to be
            - value: if None, returns the lambdified function for
              future use, but if some number is given, evaluate the
              function at that value.

            returns:
            functional form, lambdified function, or function value
            depending on the inputs.

            '''
```

```
    X = sp.symbols(x)
    base = sp.sin(X) + 1
    expo = sp.sin(sp.cos(X))
    fn_form = base ** expo

    if value_or_not:
        f = sp.lambdify(x, fn_form)

        if value is None:
            return f

        else:
            return f(value)

    else:
        return fn_form

def fn_deriv(x, value_or_not, value = None):
    '''
    derivative of the function f mentioned above.

    inputs:
    - x: vector of strings to represent the input variable
    - value_or_not: False if value of the function needs not
      to be evaluate at some value, True is needs to be
    - value: if None, returns the lambdified function for
      future use, but if some number is given, evaluate the
      function at that value.

    returns:
    functional form, lambdified function, or function value
    of the derivative of f depending on the inputs.

    '''


    fn_form = sp.diff(fn(x, False))

    if value_or_not:

        f = sp.lambdify(x, fn_form)

        if value is None:
            return f
        else:
            return f(value)

    else:
```

```
            return fn_form
```

Now, we are to plot the functions $f(x)$ and its derivative for the domain $[-\pi, \pi]$. This is shown in the below parts.

```
In [190]: pi = math.pi
          xval = np.linspace((-1)*pi, pi, 100)
          yval_fn = fn('x', True, xval)
          yval_fn_deriv = fn_deriv('x', True, xval)

In [191]: plt.plot(xval, yval_fn, label='original function')
          plt.plot(xval, yval_fn_deriv, label='derivative')
          plt.xlabel('x values')
          plt.ylabel('y values')
          plt.legend()
          plt.show()
```



Problem 2 Below functions, from "fwd_1" to "cen_4," represent the six finite difference quotients in Table 8.1.

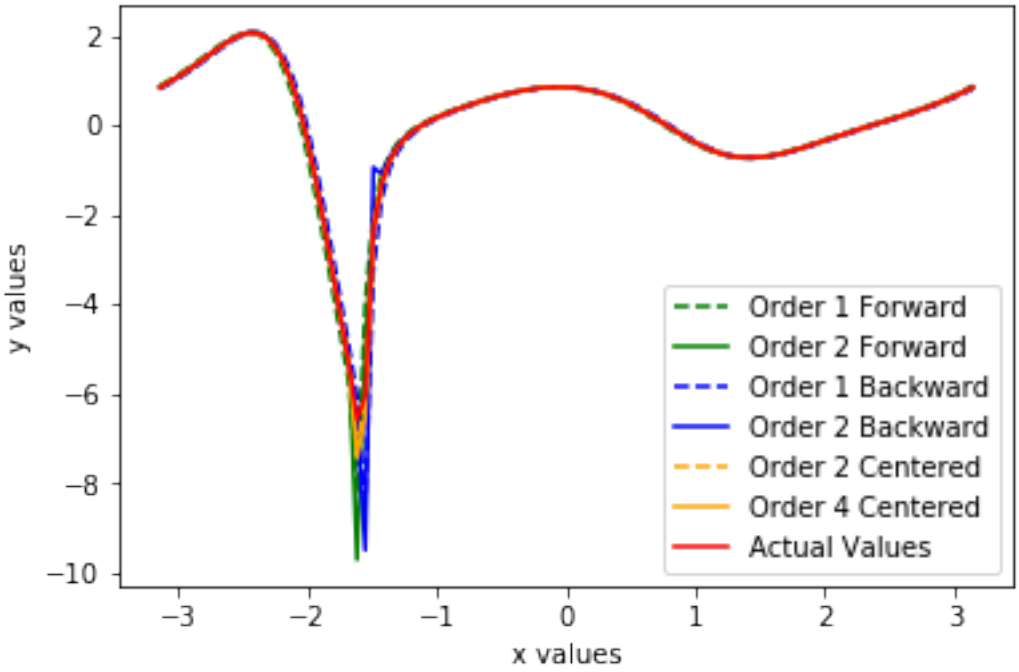
```
In [12]: def fwd_1(f, x, h):
             up = f(x+h) - f(x)
             return up / h

         def fwd_2(f, x, h):
```

```
        up = (-3)*f(x) + 4*f(x+h) - f(x+(2*h))
        return up / (2*h)

    def bwd_1(f, x, h):
        up = f(x) - f(x-h)
        return up / h

    def bwd_2(f, x, h):
        up = 3*f(x) - 4*f(x-h) + f(x-(2*h))
        return up / (2*h)

    def cen_2(f, x, h):
        up = f(x+h) - f(x-h)
        return up / (2*h)

    def cen_4(f, x, h):
        up = f(x-(2*h)) - 8*f(x-h) + 8*f(x+h) - f(x+(2*h))
        return up / (12*h)
```

To see some difference between each method, I set the value of h to be $h = 0.05$ and plot the graph as shown below. While there certainly is some difference, the overall shapes of the curves indicate that the approximations are more or less close to the actual value (of the derivative).

```
In [194]: f = fn('x', True)
          h = 0.05

          yval_fwd_1 = fwd_1(f, xval, h)
          yval_fwd_2 = fwd_2(f, xval, h)
          yval_bwd_1 = bwd_1(f, xval, h)
          yval_bwd_2 = bwd_2(f, xval, h)
          yval_cen_2 = cen_2(f, xval, h)
          yval_cen_4 = cen_4(f, xval, h)
          trueval = fn_deriv('x', True, xval)

          plt.plot(xval, yval_fwd_1, color = 'green',
                   linestyle = 'dashed', label='Order 1 Forward')
          plt.plot(xval, yval_fwd_2, color = 'green',
                   label='Order 2 Forward')
          plt.plot(xval, yval_bwd_1, color = 'blue',
                   linestyle = 'dashed', label='Order 1 Backward')
          plt.plot(xval, yval_bwd_2, color = 'blue',
                   label='Order 2 Backward')
          plt.plot(xval, yval_cen_2, color = 'orange',
                   linestyle = 'dashed', label='Order 2 Centered')
          plt.plot(xval, yval_cen_4, color = 'orange',
                   label='Order 4 Centered')
          plt.plot(xval, yval_fn_deriv, color = 'red',
                   label='Actual Values')
```

```
plt.legend()
plt.xlabel('x values')
plt.ylabel('y values')
plt.show()
```



0.2.3 Problem 3

As we had already wrote the function "fn_deriv" to compute the derivative of $f(x)$ at some value x_0, I use this function to evaluate $f'(1)$. This is shown to be approximately -0.3965.

```
In [237]: ## setting x0 = 1 as in the problem
          func = fn('x', True) ## Lambdified
          actual_val = fn_deriv('x', True, 1)
          print(actual_val)
```

```
-0.3965403874194623
```

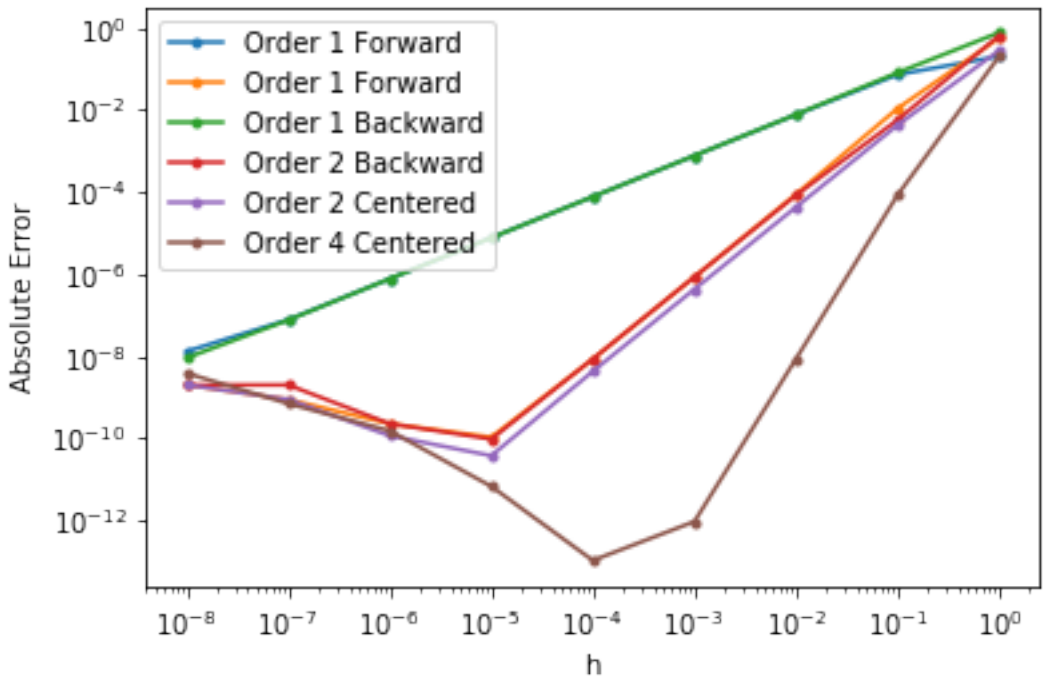
Now let us calculate the absolute errors of the approximations using the aforementioned six methods (from the actual value calculated above) at $x_0 = 1$.

```
In [271]: h_arr = np.logspace(-8, 0, 9)

          f1 = abs(fwd_1(func, 1, h_arr) - actual_val)
          f2 = abs(fwd_2(func, 1, h_arr) - actual_val)
```

```
b1 = abs(bwd_1(func, 1, h_arr) - actual_val)
b2 = abs(bwd_2(func, 1, h_arr) - actual_val)
c2 = abs(cen_2(func, 1, h_arr) - actual_val)
c4 = abs(cen_4(func, 1, h_arr) - actual_val)
```

If plotted (with h on the x-axis, absolute error on the y-axis), the graph will be as follows.

```
In [277]: plt.plot(h_arr, f1, label='Order 1 Forward', marker='.')
          plt.plot(h_arr, f2, label='Order 1 Forward', marker='.')
          plt.plot(h_arr, b1, label='Order 1 Backward', marker='.')
          plt.plot(h_arr, b2, label='Order 2 Backward', marker='.')
          plt.plot(h_arr, c2, label='Order 2 Centered', marker='.')
          plt.plot(h_arr, c4, label='Order 4 Centered', marker='.')
          plt.loglog()
          plt.legend(loc = 2)
          plt.xlabel('h')
          plt.ylabel('Absolute Error')
          plt.show()
```



0.2.4 Problem 4

Firstly, I load the required file "plane.npy" to read in the data.

```
In [228]: plane = np.load('plane.npy')
```

```
Out[228]: array([[ 7.  , 56.25, 67.54],
                 [ 8.  , 55.53, 66.57],
                 [ 9.  , 54.8 , 65.59],
                 [10.  , 54.06, 64.59],
                 [11.  , 53.34, 63.62],
                 [12.  , 52.69, 62.74],
                 [13.  , 51.94, 61.72],
                 [14.  , 51.28, 60.82]])
```

Below function, using the above information in "plane," will approximate the plane speed with forward quotient for $t = 7$, backward for $t = 14$, and centered for all other t. Notice that since we observe the degrees at integer times, h (used for approximation) has to be $h = 1$.

```
In [232]: def plane_speed(plane):
              '''
              Given list of triple (time, alpha, and beta (latter two being
              in degrees)), returns a tuple of time and speed of a plane.

              input:
              plane: list of lists [time, alpha, beta]

              output
              tuple of time and speed of a plane at that time

              '''
              ## distance between the radars
              a = 500

              speed = []

              for i, tup in enumerate(plane):
                  t, al, be = tup
                  al = np.deg2rad(al)
                  be = np.deg2rad(be)
                  x = a * np.tan(al) / (np.tan(be) - np.tan(al))
                  y = x * np.tan(be)

                  ## forward quotient for time t = 7
                  if i == 0:
                      tf, alf, bef = plane[i+1]
                      alf = np.deg2rad(alf)
                      bef = np.deg2rad(bef)
                      xf = a * np.tan(alf) / (np.tan(bef) - np.tan(alf))
                      yf = x * np.tan(bef)

                      speed.append([t, ((xf-x)**2 + (yf-y)**2)**0.5])

                  else:
```

```
            tp, alp, bep = plane[i-1]
            alp = np.deg2rad(alp)
            bep = np.deg2rad(bep)
            xp = a * np.tan(alp) / (np.tan(bep) - np.tan(alp))
            yp = x * np.tan(bep)

            ## backward quotient for time t = 14
            if i == (len(plane) - 1):
                speed.append([t,((x-xp)**2 + (y-yp)**2)**0.5])

            ## centered quotient for time t != 14 or 7
            else:
                tf, alf, bef = plane[i+1]
                alf = np.deg2rad(alf)
                bef = np.deg2rad(bef)
                xf = a * np.tan(alf) / (np.tan(bef) - np.tan(alf))
                yf = x * np.tan(bef)

                speed.append([t, (((xf-xp)/2)**2 + ((xf-xp)/2)**2)**0.5])

    return speed
```

Below represents the list of lists showing time and speed at that time together.

```
In [233]: plane_speed(plane)

Out[233]: [[7.0, 100.83999912507402],
           [8.0, 64.09715203583406],
           [9.0, 66.86439206122877],
           [10.0, 68.39274812849645],
           [11.0, 65.97444035560711],
           [12.0, 70.28714345819041],
           [13.0, 73.39566035331029],
           [14.0, 92.43996800628017]]
```

Problem 5 The below function "jacobianCalc" uses the second-order centered difference quotient to calculate the Jacobian of some function f at the value $x0$, which h being the distance for approximation.

```
In [204]: def jacobianCalc(f, x0, h):
              '''
              Approximates Jacobian of function f at x0 given h.

              Inputs:
              f  (function) : function that takes in numerical values,
                              returning numerical values
              x0 (float or lst) : point to be evaluated at
              h  (float) : distance for approximation
```

```
            Output:

            NumPy array representing the Jacobian matrix
            evaluated at x0 (given h)

            '''

            jacob_list = []
            len_f, len_x0 = len(f(x0)), len(x0)

            x_vec = np.array(x0)
            zero_vec = [0] * len_x0

            for i in range(1, len_x0+1):
                basis_vec = zero_vec[:]
                basis_vec[i-1] = 1
                basis_vec = np.array(basis_vec)

                up = np.array(f(x0 + h * basis_vec)) - np.array(f(x0 - h * basis_vec))
                jacob_list.append(list(up / (2 * h)))

            jacob_list = np.array(jacob_list)
            jacob_list = jacob_list.transpose()
            return jacob_list
```

To make sure this function works, let us define the below "simple" function $f(x,y) = [x^2, x^3 - y]^T$, and check whether the Jacobian has been produced correctly. We can see that at least for this function, the Jacobian at $(x,y) = (2,3)$ has been calculated (almost) correctly.

```
In [205]: def simple(in_put):

              x, y = in_put[0], in_put[1]
              lst = [x ** 2, x ** 3 - y]

              return lst
```

```
In [206]: print(jacobianCalc(simple, [2, 3], 0.01))
```

```
[[ 4.      0.    ]
 [12.0001 -1.    ]]
```

Problem 7 In the below section, I create a function to create a list of N randomly (and uniformly) drawn values, given some upper and lower bounds. $-\pi$ to π.

```
In [6]: def rando(N, lb, ub, seed=60637):
            '''
```

```
        Returns a list of N randomly and uniformly drawn values from
        [lb, ub].

        Inputs:
        lb, ub: lower and upper bounds of the draw
        N: number of draws to be performed
        seed: seed for random and uniform draw

        Returns:
        list of N randomly and uniformly drawn values

        '''

        np.random.seed(seed)
        rand_lst = []

        for _ in range(0, N):
            randnum = np.random.random() * (ub - lb) + lb
            rand_lst.append(randnum)

        return rand_lst
```

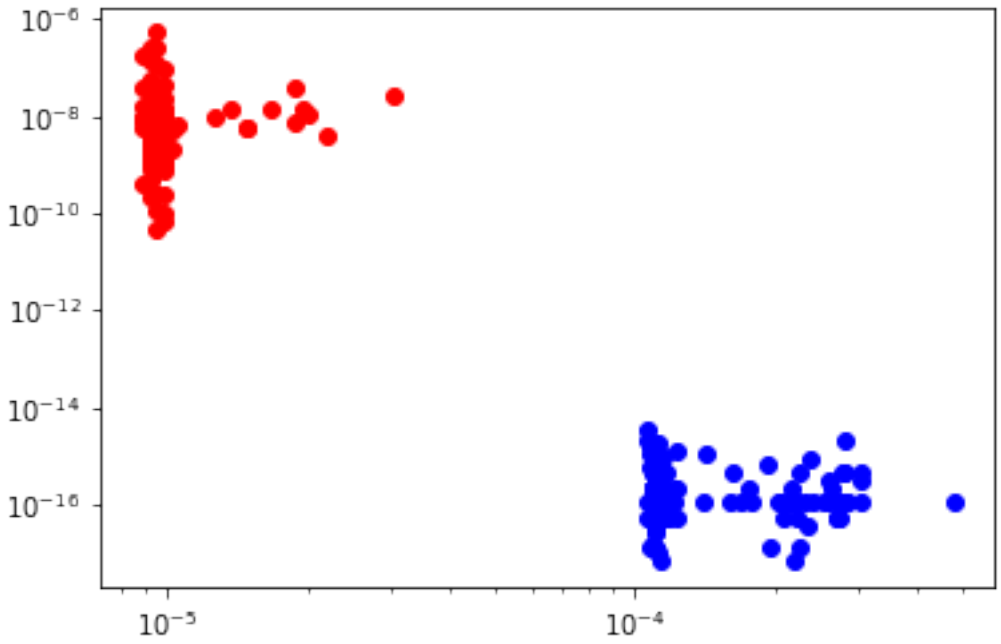
In [18]:
```
from autograd import numpy as anp
from autograd import grad

func = fn('x', True)

def experiment(N, lbub=(-math.pi, math.pi)):
    ## random draws
    xvec = rando(N, lbub[0], lbub[1])

    ## using SymPy
    actual_val = []
    sympy_time = []
    for x in xvec:
        start_time = time.clock()
        val = fn_deriv('x', True, x)
        elapsed = time.clock() - start_time
        sympy_time.append(elapsed)
        actual_val.append(val)

    actual_val = np.array(actual_val)

    ## using centered, order 4
    cen4_val = []
    cen4_time = []
    for x in xvec:
        start_time = time.clock()
```

```
        val = cen_4(func, x, 1e-8)
        elapsed = time.clock() - start_time
        cen4_time.append(elapsed)
        cen4_val.append(val)

    cen4_val = np.array(cen4_val)
    cen4_error = abs(cen4_val - actual_val)

    ## using AutoGrad's grad

    for_grad = lambda x: (anp.sin(x) + 1) ** (anp.sin(anp.cos(x)))

    grad_val = []
    grad_time = []

    for x in xvec:
        start_time = time.clock()
        grad_diff = grad(for_grad)
        val = grad_diff(x)
        elapsed = time.clock() - start_time
        grad_time.append(elapsed)
        grad_val.append(val)

    grad_val = np.array(grad_val)
    grad_error = abs(grad_val - actual_val)

    return sympy_time, cen4_time, cen4_error, grad_time, grad_error
```

In [20]:
```
sp_t, c4_t, c4_e, gr_t, gr_e = experiment(200)
```

In [28]:
```
plt.plot(c4_t, c4_e, 'ro')
plt.plot(gr_t, gr_e, 'bo')
plt.loglog()
```

Out[28]:
```
[]
```



0.2.5 Exercise 2. Numerical Integration

Problem 2.1 In the parts below, I firstly define the function "g_x" to represent $g(x)$ in the question and function "integr" for evaluting the integral using the three Newton-Cotes methods.

```
In [104]: def g_x(x):
              '''
              Function to return the value(s) of g when given input x where
              g is given in the exercise.

              input: number, sympy symbol, or vector containing them
              output: evaluated value

              '''

              return 0.1 * (x**4) - 1.5 * (x**3) + 0.53 * (x**2) + 2 * x + 1
```

```
In [105]: def integr(func, a, b, N, method='midpoint'):
              '''
              Function to return Newton-Cotes approximated values of integration.

              inputs:
              func: function to be used in the approximation
              a, b: lower and upper bounds of the integral
              N: number of points used in approximation
```

```
        method: a Newton-Cotes method, either midpoint, Simpsons, or
                trapezoid

        output:
        approximated integral value

        '''

        # calculate vector of N + 1 bar bounds
        bin_cuts = np.linspace(a, b, N + 1)
        binsize = (b-a)/N

        method = method.lower()

        if method == 'midpoint':
            midpoints = bin_cuts[1:] - binsize/2
            mid_vals = func(midpoints) * binsize
            return mid_vals.sum()

        elif method == 'trapezoid':
            bin_fvals = func(bin_cuts)
            bin_vals = (bin_fvals[1:] + bin_fvals[0:N]) / 2 * binsize
            return bin_vals.sum()

        elif method == 'Simpsons':
            bin_fvals = func(bin_cuts)
            bin_ones = (bin_fvals[1:] + bin_fvals[0:N])
            midpoints = bin_cuts[1:] - binsize/2
            mid_ones = func(midpoints) * 4
            total = bin_ones + mid_ones
            total_vals = total / 6 * binsize
            return total_vals.sum()

        else:
            raise ValueError('Method name should either be: Simpsons, trapezoid, or midpo
```

For demonstration, I set $N = 1000$ and present the results below. For my trials, at least, it seems that the Simpson's rule works the best in terms of having the least absolute error.

```
In [92]: print("Method: Midpoint")
         mid_approx = integr(g_x, -10, 10, 1000, method='midpoint')
         abs_error = abs(mid_approx - 4373.33)
         print("Approximation:", mid_approx, ";", "Absolute error:", abs_error)
         print()
         print("Method: Trapezoid")
         trap_approx = integr(g_x, -10, 10, 1000, method='trapezoid')
         abs_error = abs(trap_approx - 4373.33)
         print("Approximation:", trap_approx, ";", "Absolute error:", abs_error)
```

```
print()
print("Method: Simpsons")
Simp_approx = integr(g_x, -10, 10, 1000, method='Simpsons')
abs_error = abs(Simp_approx - 4373.33)
print("Approximation:", Simp_approx, ";", "Absolute error:", abs_error)
```

```
Method: Midpoint
Approximation: 4373.319646676 ; Absolute error: 0.010353323999879649

Method: Trapezoid
Approximation: 4373.360706656001 ; Absolute error: 0.030706656000802468

Method: Simpsons
Approximation: 4373.333333336001 ; Absolute error: 0.0033333360006508883
```

Problem 2.2 Firstly, I use the fact that $\int_{Z_{min}}^{Z_{max}} f(Z;\mu,\sigma)dZ = F(Z_{max};\mu,\sigma) - F(Z_{min};\mu,\sigma)$ by definition of CDFs. Using this, I write the function "ncNormal" in the below space (for "nc" for Newton-Cotes).

```
In [121]: from scipy.stats import norm as nr
```

```
def ncNormal(mu, sig, N, k):
    '''
    Given mean and standard deviation of a normal distribution, along with
    the number of nodes (to be equally spaced) and number of standard
    deviations to be away from the mean, returns vector of weights and
    vector of nodes for Newton-Cotes quadrature method.

    input:
    mu: mean of the normal distribution
    sig: standard deviation of the normal distribution
    N: number of nodes
    k: number of standard deviations to be away from the mean

    output:
    tuple (double) containing
    - vector of nodes
    - vector of weights

    '''

    # Lower and upper bounds (i.e. furthest nodes)
    lb, ub = mu - sig*k, mu + sig*k

    weights = []

    nodes = np.linspace(lb, ub, N)
```