ProblemSet2-Q2

January 21, 2019

0.0.1 Problem Set #2

MACS 30150, Dr. Evans

Due Monday, Jan. 21 at 11:30am

Haowen Shang

2. Numerical integration exercises from Evans: Numerical Integration lab

```
In [1]: import numpy as np
        import pandas as pd
        from scipy.stats import norm
```

Exercise 2.1

```
In [2]: def g_x(x):
            x = 0.1 * (x ** 4) - 1.5 * x ** 3 + 0.53 * x ** 2 + 2 * x + 1
            return x

        def integr_g_x(g, a , b, N, method):
            if method == "midpoint":
                #Calculate vector of N+1 bar bounds
                bin_cuts = np.linspace(a, b, N + 1)

                #Calculate vector of midpoints
                m = (b - a) / (2*N)
                midpoint = np.linspace(a + m, b - m, N)

                #Evaluate the function at the midpoint
                mid_val = g(midpoint)

                #Add up the area of the bins
                approx_integr = (b-a) / N * sum(mid_val)

            if method == "trapezoid":
                #Calculate vector of N+1 bar bounds
                bin_cuts = np.linspace(a, b, N + 1)
                #Evaluate the function at the bar bounds
                bin_val = g(bin_cuts)

                #Add up the area of the bins
                approx_integr = (b-a) / (2 * N) * (bin_val[0] + 2 * sum(bin_val[1 : N]) + bin_

            if method == "Simpsons":
                #Calculate vector of N+1 bar bounds
                bin_cuts = np.linspace(a, b, 2*N + 1)

                #Evaluate the function at the bar bounds
                bin_val = g(bin_cuts)

                #Add up the value at bar bounds
                Sum = bin_val[0] + bin_val[2*N]
                for i in range(1, 2*N):
                    if  i % 2 == 0:
                        Sum += 2 * bin_val[i]
                    else:
                        Sum += 4* bin_val[i]

                #Add up the area of the bins
                approx_integr = (b-a) / (6 * N) * Sum

            return approx_integr
```

```
In [3]: Val = 0.02*(10**5-(-10)**5)+0.53/3*(10**3-(-10)**3)+20
        #Midpoint rule
        M = integr_g_x(g_x, -10, 10, 10000, "midpoint")
        print("The approxiamation value using midpoint rule is: ", M)
        print("The difference with true value is: ", M - Val)
```

```
The approxiamation value using midpoint rule is:  4373.333196466634
The difference with true value is:  -0.00013686669899470877
```

```
In [4]: #Trapezoid rule
        T = integr_g_x(g_x, -10, 10, 10000, "trapezoid")
        print("The approxiamation value using trapezoid rule is: ", T)
        print("The difference with true value is: ", T - Val)
```

```
The approxiamation value using trapezoid rule is:  4373.333607066684
The difference with true value is:  0.00027373335069569293
```

```
In [5]: #Simpson's rule
        S = integr_g_x(g_x, -10, 10, 10000, "Simpsons")
        print("The approxiamation value using Simpson's rule is: ", S)
        print("The difference with true value is: ", S - Val)
```

```
The approxiamation value using Simpson's rule is:  4373.33333333337
The difference with true value is:  3.728928277269006e-11
```

Comparing the values of these integrals to the true analytical value of the integral, we find that Simpson's rule is more accurate.

Exercise 2.2

```
In [6]: def appro_norm(mu, sigma, N, k):
            #Calculate vector of nodes
            z = np.linspace(mu-k*sigma, mu+k*sigma, N)

            #Initialize vector of weights
            weight = np.zeros(N)

            #Calculate vector of weights
            weight[0] = norm.cdf((z[0]+z[1])/2, loc=mu, scale=sigma)
            for i in range(1, N-1):
                weight[i] = integr_g_x(lambda x: norm.pdf(x, loc=mu, scale=sigma), (z[i-1]+z[i
            weight[N-1] = 1-norm.cdf((z[N-2]+z[N-1])/2, loc=mu, scale=sigma)
            n = {'nodes' : z, 'weight' : weight}
            df = pd.DataFrame(n)

            return df

        df = appro_norm(0, 1, 11, 3)
        df
```

```
Out[6]:     nodes    weight
        0    -3.0  0.003467
        1    -2.4  0.014397
        2    -1.8  0.048943
        3    -1.2  0.117253
        4    -0.6  0.198028
        5     0.0  0.235823
        6     0.6  0.198028
        7     1.2  0.117253
        8     1.8  0.048943
        9     2.4  0.014397
        10    3.0  0.003467
```

Exercise 2.3

```
In [7]: def appro_log_norm(mu, sigma, N, k):
            #Calculate vector of nodes
            z = np.linspace(mu-k*sigma, mu+k*sigma, N)
            a = np.e**z
```

```
        #Initialize vector of weights
        weight = np.zeros(N)

        #Calculate vector of weights
        weight[0] = norm.cdf((z[0]+z[1])/2, loc=mu, scale=sigma)
        for i in range(1, N-1):
            weight[i] = integr_g_x(lambda x: norm.pdf(x, loc=mu, scale=sigma), (z[i-1]+z[i
        weight[N-1] = 1-norm.cdf((z[N-2]+z[N-1])/2, loc=mu, scale=sigma)
        n = {'node' : a, 'weight' : weight}
        df = pd.DataFrame(n)
        approxiamation = sum(df["node"]*df["weight"])
        return approxiamation, df

    appro, df = appro_log_norm(0, 1, 11, 3)
    appro
```

```
Out[7]: 1.6639409483466545
```

```
In [8]: df
```

```
Out[8]:          node    weight
        0    0.049787  0.003467
        1    0.090718  0.014397
        2    0.165299  0.048943
        3    0.301194  0.117253
        4    0.548812  0.198028
        5    1.000000  0.235823
        6    1.822119  0.198028
        7    3.320117  0.117253
        8    6.049647  0.048943
        9   11.023176  0.014397
        10  20.085537  0.003467
```

Exercise 2.4

```
In [10]: appro, df = appro_log_norm(10.5, 0.8, 1000, 8)
         EI = np.e**(10.5+0.5*(0.8**2))
         difference = abs(appro - EI)

         print("approximation of the expected value of income: ", appro)
         print("expected value of income: ", EI)
         print("difference between these two values: ", difference)
```

```
approximation of the expected value of income:  50011.429102086586
expected value of income:  50011.08700852173
difference between these two values:  0.34209356485371245
```

Exercise 3.1

```
In [13]: from scipy.special.orthogonal import p_roots

         def gauss(f,a,b,n):
             # Find optimal weights and nodes using p_roots
             [node,weight] = p_roots(n)
             # Compute weight*node and convert the limits of [a, b] to [-1,1]
             G=0.5*(b-a)*sum(weight*f(0.5*(b-a)*node+0.5*(b+a)))
             return G

         Gauss = gauss(lambda x: 0.1*x**4-1.5*x**3+0.53*x**2+2*x+1, -10, 10, 3)
         Newton = integr_g_x(lambda x: 0.1*x**4-1.5*x**3+0.53*x**2+2*x+1, -10, 10, 10000, "Sim
         Val = 0.02*(10**5-(-10)**5)+0.53/3*(10**3-(-10)**3)+20
         print("The result of Gaussian approximation is", Gauss, 'and the absolute error is', a
         print("The result of Newton-Cotes approximation is", Newton, 'and the absolute error i

         #Reference : https://stackoverflow.com/questions/27115917/gauss-legendre-quadrature-i
```

```
The result of Gaussian approximation is 4373.333333333335 and the absolute error is 1.81898940
The result of Newton-Cotes approximation is 4373.33333333337 and the absolute error is 3.728928
```

Compared with the results in Exercise 2.1, Gaussian approximate is more accurate.

Exercise 3.2

```
In [14]: from scipy import integrate

         GQ, err = integrate.quad(lambda x: 0.1*x**4-1.5*x**3+0.53*x**2+2*x+1, -10, 10)

         print("The result of Python Gaussian approximate is", GQ, 'and the absolute error is'
```

```
The result of Python Gaussian approximate is 4373.333333333334 and the absolute error is 9.094
```

Exercise 4.1

```
In [15]: def circle(x, y):
                 if x ** 2 + y ** 2 <= 1:
                     return 1
                 else:
                     return 0

         def MC(fn, omega, N):
             counter = 0
             xran = np.random.uniform(omega[0], omega[1], N)
             yran = np.random.uniform(omega[2], omega[3], N)
```

```
    for i in range(N):
        x = xran[i]
        y = yran[i]
        counter += fn(x, y)
    area = (omega[1] - omega[0]) * (omega[3] - omega[2])
    return area*counter/N

np.random.seed(seed=25)
N = 1
while round(MC(circle, [-1, 1, -1, 1], N), 4) != 3.1415:
    N += 1

print("The smallest number of random draws N is", N)
```

```
The smallest number of random draws N is 615
```

Exercise 4.2

```
In [16]: def isPrime(n):
             '''
             --------------------------------------------------------------------
             This function returns a boolean indicating whether an integer n is a
             prime number
             --------------------------------------------------------------------
             INPUTS:
             n = scalar, any scalar value

             OTHER FUNCTIONS AND FILES CALLED BY THIS FUNCTION: None

             OBJECTS CREATED WITHIN FUNCTION:
             i = integer in [2, sqrt(n)]

             FILES CREATED BY THIS FUNCTION: None

             RETURN: boolean
             --------------------------------------------------------------------
             '''
             for i in range(2, int(np.sqrt(n) + 1)):
                 if n % i == 0:
                     return False

             return True

In [17]: def primes_ascend(N, min_val=2):
             '''
             --------------------------------------------------------------------
             This function generates an ordered sequence of N consecutive prime
```

```
         numbers, the smallest of which is greater than or equal to 1 using
         the Sieve of Eratosthenes algorithm.
         (https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes)
         --------------------------------------------------------------------
         INPUTS:
         N       = integer, number of elements in sequence of consecutive
                   prime numbers
         min_val = scalar >= 2, the smallest prime number in the consecutive
                   sequence must be greater-than-or-equal-to this value

         OTHER FUNCTIONS AND FILES CALLED BY THIS FUNCTION:
             isPrime()

         OBJECTS CREATED WITHIN FUNCTION:
         primes_vec     = (N,) vector, consecutive prime numbers greater than
                          min_val
         MinIsEven      = boolean, =True if min_val is even, =False otherwise
         MinIsGrtrThn2  = boolean, =True if min_val is
                          greater-than-or-equal-to 2, =False otherwise
         curr_prime_ind = integer >= 0, running count of prime numbers found

         FILES CREATED BY THIS FUNCTION: None

         RETURN: primes_vec
         --------------------------------------------------------------------
         '''
         primes_vec = np.zeros(N, dtype=int)
         MinIsEven = 1 - min_val % 2
         MinIsGrtrThn2 = min_val > 2
         curr_prime_ind = 0
         if not MinIsGrtrThn2:
             i = 2
             curr_prime_ind += 1
             primes_vec[0] = i
         i = min(3, min_val + (MinIsEven * 1))
         while curr_prime_ind < N:
             if isPrime(i):
                 curr_prime_ind += 1
                 primes_vec[curr_prime_ind - 1] = i
             i += 2

         return primes_vec
```

```
In [18]: def equidis(n,d,Type):
         prime = primes_ascend(d)
         if Type == 'Weyl':
             return list((i - i // 1) for i in np.sqrt(prime)*n)
         elif Type == 'Haber':
             return list((i - i // 1) for i in np.sqrt(prime)*n*(n+1)/2)
         elif Type == 'Niederreiter':
             return list((i - i // 1) for i in [n*(2**(j/(n+1))) for j in range(1, d+1)])
         elif Type == 'Baker':
             return list((i - i // 1) for i in [n*np.exp(1/j) for j in range(1, d+1)])
```

```
In [19]: equidis(10, 4, 'Weyl')

Out[19]: [0.142135623730951,
          0.32050807568877104,
          0.36067977499789805,
          0.4575131106459054]

In [20]: equidis(10, 4, 'Haber')

Out[20]: [0.7817459305202306,
          0.2627944162882443,
          0.9837387624884428,
          0.5163221085524867]

In [21]: equidis(10, 4, 'Niederreiter')

Out[21]: [0.6504108943996272,
          0.3431252219546259,
          0.08089444404447121,
          0.8666489800943182]

In [22]: equidis(10, 4, 'Baker')

Out[22]: [0.18281828459045002,
          0.4872127070012837,
          0.9561242508608956,
          0.8402541668774148]
```

Exercise 4.3

```
In [23]: def g(x,y):
             if x**2+y**2<1:
                 return 1
             else:
                 return 0

         np.random.seed(25)
         def QMC(fn, omega, N, Type):
             x1, x2=omega[0],omega[1]
             y1, y2=omega[2],omega[3]
             if Type=='Weyl':
                 xran=[2*np.array(equidis(i,2,'Weyl'))[0]-1 for i in range(N)]
                 yran=[2*np.array(equidis(i,2,'Weyl'))[1]-1 for i in range(N)]
```

```
            elif Type=='Haber':
                xran=[2*np.array(equidis(i,2,'Haber'))[0]-1 for i in range(N)]
                yran=[2*np.array(equidis(i,2,'Haber'))[1]-1 for i in range(N)]
            elif Type=='Niederreiter':
                xran=[2*np.array(equidis(i,2,'Niederreiter'))[0]-1 for i in range(N)]
                yran=[2*np.array(equidis(i,2,'Niederreiter'))[1]-1 for i in range(N)]
            elif Type=='Baker':
                xran=[2*np.array(equidis(i,2,'Baker'))[0]-1 for i in range(N)]
                yran=[2*np.array(equidis(i,2,'Baker'))[1]-1 for i in range(N)]
            counter = 0
            for i in range(N):
                x = xran[i]
                y = yran[i]
                counter += fn(x, y)
            area = (omega[1] - omega[0])*(omega[3] - omega[2])
            return area*counter/N
```

```
In [24]: def f(x, y):
             if x ** 2 + y ** 2 <= 1:
                 return 1
             else:
                 return 0


         N = 1
         while round(QMC(f, [-1, 1, -1, 1], N, "Weyl"),4) != 3.1415:
             N += 1
         print("The smallest number of random draws N with Weyl sequence is", N)
```

```
The smallest number of random draws N with Weyl sequence is 1230
```

```
In [25]: N = 1
         while round(QMC(f, [-1, 1, -1, 1], N, "Haber"),4) != 3.1415:
             N += 1

         print("The smallest number of random draws N with Haber sequence is", N)
```

```
The smallest number of random draws N with Haber sequence is 2064
```

```
In [27]: N = 1
         while round(QMC(f, [-1, 1, -1, 1], N, "Niederreiter"),4) != 3.1415:
             N += 1
             if N > 5000:
                 print("The smallest number of random draws N with Niederreiter sequence excee
                 break
         else:
             print("The smallest number of random draws N with Niederreiter sequence is", N)
```

```
The smallest number of random draws N with Niederreiter sequence exceeds 5000.
```

```
In [29]: N = 1
         while round(QMC(f, [-1, 1, -1, 1], N, "Baker"),4) != 3.1415:
             N += 1

         print("The smallest number of random draws N with Baker sequence is", N)
```

```
The smallest number of random draws N with Baker sequence is 205
```

From the above results, we can see that Baker sequence is the fastest one to converge to 3.1415, Weyl sequence is the second one, Haber sequence is the third one and Niederreiter sequence is the slowest one.

```
In [ ]:
```