ps2 part2

January 21, 2019

1 Ps2 part 2 Numerical Integration

1.0.1 by [Ying Cai]

The code in this Jupyter notebook was written using Python 3.6.

Exercise 2.1. You can verify that the analytical solution to the integral of the function

$$g(x) = 0.1x^4 - 1.5x^3 + 0.53x^2 + 2x + 1$$

between $x = -10$ and $x = 10$ is $\int_{-10}^{10} g(x)dx = 4,373.3\bar{3}$. Write a Python function that will take as arguments an anonymous function that the user specifies representing $g(x)$, integration bounds a and b, the number of intervals N, and

```
method = {'midpoint', 'trapezoid', 'Simpsons'}
```

Using the composite methods, evaluate the numerical approximations of the integral $\int_a^b g(x)dx$ using all three Newton-Cotes methods in your function and compare the difference between the values of these integrals to the true analytical value of the integral.

```
In [8]: true_val = 0.02*(10**5-(-10)**5)+0.53/3*(10**3-(-10)**3)+20
        def Newton_integr(g,a,b,N,method='midpoint'):
            if method not in ['midpoint','trapezoid', 'simpsons']:
                raise ValueError
            else:
                if method == 'midpoint':
                    unit = 0
                    for i in range(N):
                        unit += g(a+(2.0*i+1)*(b-a)/(2*N))
                    return (b-a)*unit/N
                if method == 'trapezoid':
                    unit = g(a)+g(b)
                    for i in range(1,N):
                        unit += 2.0*g(a+i*(b-a)/N)
                    return (b-a)*unit/(2.0*N)
                if method == 'simpsons':
                    unit = g(a)+g(b)+4.0*g(a+(2*N-1)*(b-a)/(2*N))
                    for i in range(1,N):
                        unit += 4.0*g(a+(2*i-1)*(b-a)/(2*N))
```

```
                unit += 2.0*g(a+2*i*(b-a)/(2*N))
            return (b-a)*unit/(6*N)
g_test = lambda x: 0.1*x**4-1.5*x**3+0.53*x**2+2*x+1

for method in ['midpoint','trapezoid', 'simpsons']:
    val_test = Newton_integr(g_test, -10, 10, 100000, method)
    print("approximate integration by",method,"method is:",val_test)
    print("Absolute error of",method,"method is",abs(integr-exact))
```

```
approximate integration by midpoint method is: 4373.333331964723
Absolute error of midpoint method is 1.482476363889873e-10
approximate integration by trapezoid method is: 4373.333336070682
Absolute error of trapezoid method is 1.482476363889873e-10
approximate integration by simpsons method is: 4373.333333333185
Absolute error of simpsons method is 1.482476363889873e-10
```

Exercise 2.2. Write a Python function that makes a Newton-Cotes discrete approximation of the distribution of the normally distributed variable $Z \sim N(\mu, \sigma)$. Let this function take as arguments the mean μ, the standard deviation σ, the number of equally spaced nodes N to estimate the distribution, and the number of standard deviations k away from μ to make the furthest nodes on either side of μ. Use the `scipy.stats.norm.cdf` command for the cdf of the normal distribution to compute the weights ω_n for the nodes x_n. Have this function return a vector of nodes of $[Z_1, Z_2, ...Z_N]$ and a vector of weights $[\omega_1, \omega_2, ...\omega_N]$ such that ω_i is given by the integral under the normal distribution between the midpoints of the two closest nodes. Define $f(Z; \mu, \sigma)$ as the pdf of the normal distribution and $F(Z; \mu, \sigma)$ as the cdf.

$$\omega_i = \begin{cases} F\left(\frac{Z_1+Z_2}{2}; \mu, \sigma\right) & \text{if} \quad i = 1 \\ \int_{Z_{min}}^{Z_{max}} f(Z; \mu, \sigma) dZ & \text{if} \quad 1 < i < N \\ 1 - F\left(\frac{Z_{N-1}+Z_N}{2}; \mu, \sigma\right) & \text{if} \quad i = N \end{cases}$$

$$\text{where} \quad Z_{min} = \frac{Z_{i-1}+Z_i}{2} \quad \text{and} \quad Z_{max} = \frac{Z_i + Z_{i+1}}{2}$$

What are the weights and nodes $\{\omega_n, Z_n\}_{n=1}^N$ for $N = 11$?

```
In [39]: import numpy as np
         import pandas as pd
         from scipy.stats import norm
         from scipy.integrate import quad
         def Newton_Cotes(N, mu=0, sigma=1, k=4):
             nodes = np.linspace(mu-k*sigma, mu+k*sigma, N)
             weights = np.zeros(N)
             weights[0] = norm.cdf((nodes[0]+nodes[1])/2, loc=mu, scale=sigma)
             for i in range(1,N-1):
                 f = lambda x:norm.pdf(x, loc=mu, scale=sigma)
                 weights[i] = quad(f,(nodes[i-1]+nodes[i])/2,(nodes[i+1]+nodes[i])/
             weights[N-1] = 1-norm.cdf((nodes[N-2]+nodes[N-1])/2, loc=mu, scale=sig
```

```
        return nodes, weights
    nodes, weights = Newton_Cotes(11)

    disp = pd.DataFrame({"Nodes":nodes,"weights":weights})
    print('For N = 11\n', disp)
```

```
For N = 11
     Nodes   weights
0    -4.0  0.000159
1    -3.2  0.002396
2    -2.4  0.020195
3    -1.6  0.092320
4    -0.8  0.229509
5     0.0  0.310843
6     0.8  0.229509
7     1.6  0.092320
8     2.4  0.020195
9     3.2  0.002396
10    4.0  0.000159
```

Exercise 2.3. If $Z \sim N(\mu, \sigma)$, then $A \equiv e^Z \sim LN(\mu, \sigma)$ is distributed lognormally and $\log(A) \sim N(\mu, \sigma)$. Use your knowledge that $A \equiv e^Z$, $\log(A) \sim N(\mu, \sigma)$, and your function from Exercise 2.2 to write a function that gives a discrete approximation to the lognormal distribution. Note: You will not end up with evenly spaced nodes $[A_1, A_2, ...A_N]$, but your weights should be the same as in Exercise 2.2.

```
In [44]: def Newton_Cotes_log(N, mu=0, sigma=1, k=4):
             Z = np.linspace(mu-k*sigma, mu+k*sigma, N)
             nodes = np.e**Z
             weights = np.zeros(N)
             weights[0] = norm.cdf((Z[0]+Z[1])/2, loc=mu, scale=sigma)
             for i in range(1,N-1):
                 func = lambda x:norm.pdf(x, loc=mu, scale=sigma)
                 weights[i] = quad(func,(Z[i-1]+Z[i])/2,(Z[i+1]+Z[i])/2)[0]
             weights[N-1] = 1-norm.cdf((Z[N-2]+Z[N-1])/2, loc=mu, scale=sigma)
             return nodes, weight

         nodes, weights = Newton_Cotes_log(11)

         disp = pd.DataFrame({"Nodes":nodes,"weights":weights})
         print('For N = 11\n', disp)
```

```
For N = 11
          Nodes   weights
0    0.018316  0.000159
1    0.040762  0.002396
2    0.090718  0.020195
3    0.201897  0.092320
```

```
4    0.449329  0.229509
5    1.000000  0.310843
6    2.225541  0.229509
7    4.953032  0.092320
8   11.023176  0.020195
9   24.532530  0.002396
10  54.598150  0.000159
```

Exercise 2.4. Let Y_i represent the income of individual i in the United States for all individuals i. Assume that income Y_i is lognormally distributed in the U.S. according to $Y_i \sim LN(\mu, \sigma)$, where the mean of log income is $\mu = 10.5$ and the standard deviation of log income is $\sigma = 0.8$. Use your function from Exercise 2.3 to compute an approximation of the expected value of income or average income in the U.S. How does your approximation compare to the exact expected value of $E[Y] = e^{\mu + \frac{\sigma^2}{2}}$?

```
In [16]: import array as arr
         nodes, weights = Newton_Cotes_log(100, mu=10.5, sigma=0.8, k=4)
         val_exact = np.exp(10.5+0.8*0.8/2)
         approx = sum(nodes*arr.transpose(weights))

         print('The difference between my result and the exact expectation is:')
         print(abs)
```

```
The difference between my result and the exact expectation is:
0.5334533017885406
```

Exercise 3.1. Approximate the integral of the function in Exercise 2.1 using Gaussian quadrature with $N = 3$, $(\omega_1, \omega_2, \omega_3, x_1, x_2, x_3)$. Use the class of polynomials $h_i(x) = x^i$. How does the accuracy of your approximated integral compare to the approximations from Exercise 2.1 and the true known value of the integral?

```
In [19]: import scipy as sp
         def Gaussian(g,a,b,N=3):
             init_weight = [1/N for i in range(N)]
             init_x = [a+i*(b-a)/(N-1) for i in range(N)]
             init = init_weight+init_x
             def func(x):
                 result = []
                 for i in range(2*N):
                     weight = x[:N]
                     node = x[N:]
                     Sum = sum(weight[k]*(node[k]**i) for k in range(N))
                     result.append((b**(i+1)-a**(i+1))/(i+1)-Sum)
                 return tuple(k for k in result)
             Vector = [k for k in sp.optimize.root(func, init)['x']]
             weight = Vector[:N]
```

```
        node = Vector[N:]
        counter = 0
        for i in range(N):
            counter += weight[i]*g(node[i])
        return counter
    test_func = lambda x: 0.1*x**4-1.5*x**3+0.53*x**2+2*x+1
    Gauss = Gaussian(test_func, -10, 10)
    Newton = integrate(test_func, -10, 10, 10000, "Simpsons")
    Exact = 0.02*(10**5-(-10)**5)+0.53/3*(10**3-(-10)**3)+20
    print("The result of Gaussian approximate is", Gauss)
    print('The absolute error of Gaussian approximate is', abs(Gauss-Exact))
    print("The result of Newton-Cotes approximate is", Newton)
    print('The absolute error of Newton-Cotes approximate is', abs(Newton-Exac
```

```
The result of Gaussian approximate is 4373.333333189591
The absolute error of Gaussian approximate is 1.4374199963640422e-07
The result of Newton-Cotes approximate is 4373.33333333337
The absolute error of Newton-Cotes approximate is 3.728928277269006e-11
```

Exercise 3.2. Use the Python Gaussian quadrature command `scipy.integrate.quad` to numerically approximate the integral from Exercise 2.1.

$$\int_{-10}^{10} g(x)dx \quad \text{where} \quad g(x) = 0.1x^4 - 1.5x^3 + 0.53x^2 + 2x + 1$$

How does the approximated integral using the `scipy.integrate.quad` command compare to the exact value of the function?

```
In [20]: Quad = quad(lambda x: 0.1*x**4-1.5*x**3+0.53*x**2+2*x+1, -10, 10)[0]
         print("The result of Python Gaussian approximate is", Quad)
         print('The absolute error of Python Gaussian approximate is', abs(Quad-Exa
```

```
The result of Python Gaussian approximate is 4373.333333333334
The absolute error of Python Gaussian approximate is 9.094947017729282e-13
```

Exercise 4.1. Use Monte Carlo integration to approximate the value of π. Define a function in that takes as arguments a function $g(\mathbf{x})$ of a vector of variables $\mathbf{x}$, the domain Ω of $\mathbf{x}$, and the number of random draws N and returns the Monte Carlo approximation of the integral $\int_\Omega g(\mathbf{x})d\mathbf{x}$. Let Ω be a generalized rectangle–width x and height y. In order to approximate π, let the functional form of the anonymous function be $g(x, y)$ from Section 4.1 with domain $\Omega = [-1, 1] \times [-1, 1]$. What is the smallest number of random draws N from Ω that matches the true value of π to the 4th decimal 3.1415? Set the random seed in your uniform random number generator to 25. This will make the correct answer consistent across submissions.

```
In [27]: 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

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 [22]: def M_C(N, func=None, omega=[-1,1,-1,1]):
             counter = 0
             x_1 = np.random.uniform(omega[0],omega[1],size=N)
             x_2 = np.random.uniform(omega[2],omega[3],size=N)
             def g(x,y):
                 if x**2+y**2<=1:
                     return 1
                 else:
                     return 0
             for i in range(N):
                 x,y = x_1[i],x_2[i]
                 if func is None:
                     counter += g(x,y)
                 else:
                     counter += func(x,y)
             return 4*counter/N
         np.random.seed(25)
         judge = False
         min_N = 0
         while judge is False:
             min_N += 1
             judge = (round(M_C(min_N),4)==3.1415)
         print("The smallest number of random draws N is", min_N)
```

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

Exercise 4.2. Define a function in that returns the n-th element of a d-dimensional equidistributed sequence. It should have support for the four sequences in the Table in Section 4.2.

```
In [23]: def equidistribution(n,d,Type='weyl'):
             prime_vector = primes_ascend(d)
             def rational_list(d):
                 return [1/(i+1) for i in range(d)]
             def cut(x):
                 return x-x//1
             if Type == 'weyl':
                 return tuple(cut(n*np.sqrt(prime_vector[i])) for i in range(d))
             elif Type == 'haber':
                 return tuple(cut(n*(n+1)*0.5*np.sqrt(prime_vector[i])) for i in ra
             elif Type == 'nie':
                 return tuple(cut(n*(2**(i/(n+1)))) for i in range(d))
             elif Type == 'baker':
                 return tuple(cut(n*(np.e**(rational_list(d)[i]))) for i in range(c
```

Exercise 4.3 Repeat Exercise 4.1 to approximate the value of π, this time using quasi-Monte Carlo integration. You will need to appropriately scale the equidistributed sequences. Compare the rates of convergence. What is the smallest number of random draws N from Ω for the quasi-Monte Carlo integration that matches the true value of π to the 4th decimal 3.1415?. Set the seed in your uniform random number generator to 25. This will make the correct answer consistent across submissions.

```
In [24]: def quasi_M_C(N,Type, func=None,omega=[-1,1,-1,1]):
             counter = 0
             x = []
             for k in range(N):
                 x.append((2*equidistribution(k,2,Type)[0]-1,2*equidistribution(k,2
             def g(X):
                 x,y = X[0], X[1]
                 if x**2+y**2<=1:
                     return 1
                 else:
                     return 0
             for i in range(N):
                 X = x[i]
                 if func is None:
                     counter += g(X)
                 else:
                     counter += func(X)
             return 4*counter/N
         Text = "The smallest number of random draws N"
         print(Text, "for M-C method is", min_N)
         for method in ['weyl','haber', 'baker']:
             judge = False
             min_N2 = 0
             while judge is False:
                 min_N2 += 1
                 judge = (round(quasi_M_C(min_N2,Type = method),4)==3.1415)
             print(Text, "for quasi-M-C method with type",method, "is", min_N2)
```

```
The smallest number of random draws N for M-C method is 615
The smallest number of random draws N for quasi-M-C method with type weyl is 1230
The smallest number of random draws N for quasi-M-C method with type haber is 2064
The smallest number of random draws N for quasi-M-C method with type baker is 205
```

```
In [25]: round(quasi_M_C(100000,Type = method),4)-3.1415

Out[25]: -0.000300000000000189

In [ ]:
```