Jacob Dennis

Physics 411

4 April 2016

Prof. Gull

Midterm 3

**Random Number Generator**

*Code*

import numpy as np

def LCG(a, c, m, x0, iterations):

X = np.zeros(iterations)

X[0] = x0

for n in range(iterations - 1):

X[n + 1] = (a \* X[n] + c) % m

return X

def FibonacciRandom(A, B, m, iterations):

a, b = min(A, B), max(A, B)

seedList = LCG(16807.0, 0.0, 2.0\*\*31 - 1, 66779.0, b)

X = seedList.tolist()

for n in range(b, iterations + b):

X.append((X[n - a] + X[n - b]) % m)

norm = max(X)

X = np.array(X) / norm

return X[b:]

def Expectance(valuesArray):

p = 1.0 / np.size(valuesArray) #Assuming equal probability for all the various results

return np.sum(valuesArray \* p)

def Covariance(valuesArray):

return (Expectance(valuesArray\*\*2) - Expectance(valuesArray)\*\*2)

def main():

#LCG

xLCG = LCG(16807.0, 0.0, 2.0\*\*31 - 1, 66779.0, 607)

print 'Linear Congruential Generator:\n', xLCG[-10:]

#Lagged Fibonacci Generator

print 'Lagged Fibonacci Generator:\n', FibonacciRandom(273, 607, 2.0\*\*31 - 1, 10)

#Mean, Variance, and Error on the Mean

myRandomSequence = FibonacciRandom(273, 607, 2.0\*\*31 - 1, 100000)

mean = np.sum(myRandomSequence) / np.size(myRandomSequence)

meanArray = np.ones(np.size(myRandomSequence)) \* mean

print 'Mean:', mean

print 'Variance:', Covariance(myRandomSequence)

print 'Error on the Mean:', abs(mean - 0.5)

if \_\_name\_\_ == '\_\_main\_\_':

main()

*Results*

Linear Congruential Generator:

[ 3.10909473e+08 6.27799560e+08 8.40047209e+08 1.11594628e+09 1.73452274e+09 3.32503830e+07 4.93438861e+08 1.79257576e+09 7.72714557e+08 1.17994609e+09]

Lagged Fibonacci Generator:

[ 0.26084745 0.39661785 0.94305276 0.47835772 0.5360287 0.66497594 0.5516199 0.66639108 0.3323461 0.89182325]

Mean: 0.499570110049

Variance: 0.0830759342556

Error on the Mean: 0.000429889950649

**Monte Carlo Integration**

*Code*

import numpy as np

import numpy.random as rand

import matplotlib.pyplot as plt

def HitOrMiss(x0, xf, y0, yf, f, sampleSize):

pointsArray = np.zeros((sampleSize, 2))

insideArea = 0.0

for point in pointsArray:

point[0] = x0 + rand.random() \* (xf - x0) #random x between x0 and xf

point[1] = y0 + rand.random() \* (yf - y0) #random y between y0 and yf

if point[1] <= (f(point[0]) - y0):

insideArea += 1.0

return (insideArea / sampleSize) \* ((xf - x0) \* (yf - y0))

def ExpDistRandomNums(sampleSize, mu):

randomArray = np.zeros(sampleSize)

for i in range(sampleSize):

z = rand.random()

randomArray[i] = -1.0 / mu \* np.log(1.0 - z)

return randomArray

def F(x):

return np.exp(-x\*\*2 / 2.0)

def Expectance(valuesArray):

p = 1.0 / np.size(valuesArray) #Assuming equal probability for all the various results

return np.sum(valuesArray \* p)

def Covariance(valuesArray):

return (Expectance(valuesArray\*\*2) - Expectance(valuesArray)\*\*2)

def ImportanceSampling(f, N):

randomRange = ExpDistRandomNums(N, 1.0)

randomRange = -np.sort(-randomRange) / max(randomRange)

I = 0.0

for x in randomRange:

I += 1.0 / N \* f(x) / np.exp(-x)

return I

def main():

xRange = np.linspace(0.0, 1.0, 100000)

I1 = HitOrMiss(0.0, np.pi, 0.0, 1.0, np.sin, 100000)

print 'Integral with the Hit or Miss Method:', I1

print 'Approximate Error:', np.sqrt(I1 \* abs(2.0 - I1)) / np.sqrt(100000)

print 'Importance Sampling integral:', ImportanceSampling(F, 100000)

print 'Approximate Error:', np.sqrt(Covariance(np.exp(-xRange\*\*2 / 2) / np.exp(-xRange))) / np.sqrt(100000)

if \_\_name\_\_ == '\_\_main\_\_':

main()

*Results*

Integral with the Hit or Miss Method: 2.00015779476

Approximate Error: 5.61795711281e-05

Importance Sampling integral: 1.07440836942

Approximate Error: 0.000632801904236

**Finite Element Method**

*Code*

import numpy as np

import matplotlib.pyplot as plt

def rho(x):

return np.sin(4.0 \* np.pi \* x)

def Poisson(x):

return -4.0 \* np.pi \* rho(x)

def plotInitialChargeDistribution():

xRange = np.linspace(0.0, 1.0, 1000)

plt.figure(0)

plt.clf()

plt.plot(xRange, rho(xRange))

plt.xlabel('x')

plt.ylabel('Charge')

plt.title('Initial Charge Distribution')

plt.savefig('Midterm III - Plot of Initial Charge Distribution.png')

plt.close(0)

return None

def HatFunction(x, xi, dx):

if (x >= (xi - dx)) & (x <= xi):

return (x - (xi - dx)) / dx

elif ((x <= (xi + dx)) & (x > xi)):

return ((xi + dx) - x) / dx

else:

return 0.0

def getCoefficients(x0, xf, dx):

N = int((xf - x0) / dx)

X = np.arange(x0 - dx, xf + dx, dx)

A = np.zeros((N, N))

b = np.zeros(N)

for i in range(N):

for j in range(N):

if i == j:

A[i][j] = -1.0/dx

elif (i == j + 1) | (i == j - 1):

A[i][j] = 2.0 / (dx\*\*2)

b[i] = (4.0 \* np.pi \* np.cos(4.0 \* np.pi \* X[i + 1]) \* (X[i + 2] + X[i] - 2.0 \* X[i + 1]) +

2.0 \* np.sin(4.0 \* np.pi \* X[i + 1]) - np.sin(4.0 \* np.pi \* X[i]) - np.sin(4.0 \* np.pi \* X[i + 2]))

return np.linalg.solve(A, b)

def PhiApprox(x, coeffs, phi0, phi1, (x0, xf), dx, basisFunc):

XiRange = np.arange(x0, xf, dx)

phi = 0.0

for i in range(len(XiRange)):

phi += coeffs[i] \* basisFunc(x, XiRange[i], dx)

phi += phi0 \* (xf - x) + phi1 \* x

return phi

def main():

plotInitialChargeDistribution()

a = getCoefficients(0.0, 1.0, 1.0/64)

xRange = np.linspace(0.0, 1.0, 100)

PhiRange1 = []

PhiRange2 = []

for x in xRange:

PhiRange1.append(PhiApprox(x, a, 0.0, 0.0, (0.0, 1.0), 1.0/64, HatFunction))

PhiRange2.append(PhiApprox(x, a, 0.0, 1.0, (0.0, 1.0), 1.0/64, HatFunction))

plt.figure(1)

plt.clf()

plt.plot(xRange, PhiRange1)

plt.xlabel('x')

plt.ylabel('phi')

plt.title('Approximating Phi(x) with Finite Differences - Run 1')

plt.savefig('Midterm III - Finite Elements 1.png')

plt.close(1)

plt.figure(2)

plt.clf()

plt.plot(xRange, PhiRange2)

plt.xlabel('x')

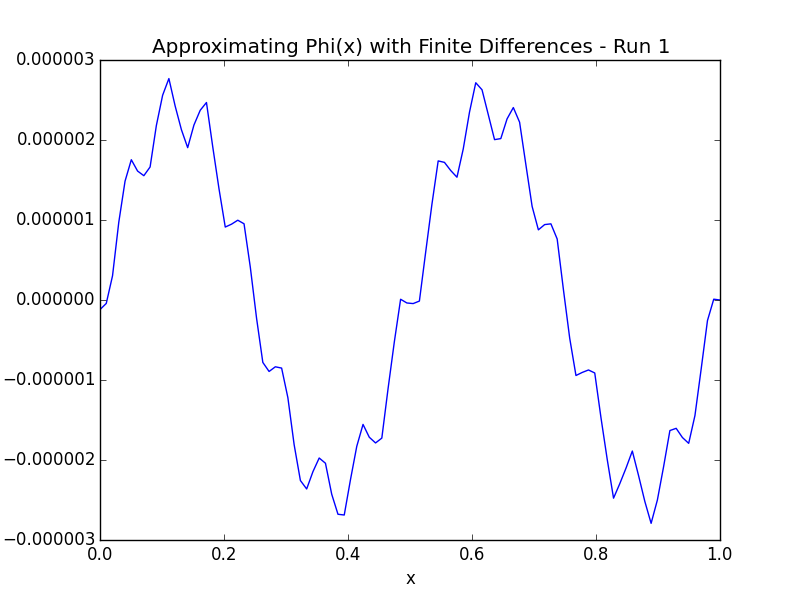
plt.ylabel('phi')

plt.title('Approximating Phi(x) with Finite Differences - Run 2')

plt.savefig('Midterm III - Finite Elements 2.png')

if \_\_name\_\_ == '\_\_main\_\_':

main()



*Results*

