0.1 Homework 3

0.1.1 Problem 1

[1]: import numpy as np
    import math

[2]: # The random number seed
np.random.seed(1234756)

    # The number of pairs to pick
Npairs = [1e2, 1e3, 1e4, 1e5, 1e6, 1e7]

    # The number of tries for each pair
Ntries = 50

    # Area of a square delimited by (x,y) = (-1,-1) --> (1,1)
Area = 4.

    # Loop over the number of pairs
for N in Npairs:

        # A numpy array to store the result of tries for each pair
        piResults = np.empty(Ntries)

        # Loop over the tries
        for i in range(Ntries):
            x = -1 + 2* np.random.rand(int(N))  # random btw -1 and 1
            y = -1 + 2* np.random.rand(int(N))  # random btw -1 and 1
            Rsq = x*x + y*y
            Inside = len(Rsq[Rsq<1])
            piResults[i] = Area*Inside/N

        # calculate mean, std, precision
        mean = np.mean(piResults)
        std = np.std(piResults)
        precision = 100 * std / math.pi
        print("N pairs=\%.e Pi=\%.7f precision=\%.5f\%" %(N, mean, precision))
print("BTW: pi = \%.7f" % math.pi)

N_pairs=1e+02 Pi=3.1360000 precision=5.56742%
N_pairs=1e+03 Pi=3.1500800 precision=1.47986%
N_pairs=1e+04 Pi=3.1413200 precision=0.55550%
N_pairs=1e+05 Pi=3.1423456 precision=0.16208%
N_pairs=1e+06 Pi=3.1416570 precision=0.05585%
N_pairs=1e+07 Pi=3.1415120 precision=0.01561%
BTW: pi = 3.1415927

[3]: # Expected precision
    p = math.pi/4
    for N in Npairs:
        expPrec = 100* math.sqrt(p*(1-p)/N) / p
        print("N pairs=%e expected precision=%5f\%%" %(N, expPrec))

N pairs=1e+02 expected precision=5.22723%
N pairs=1e+03 expected precision=1.65300%
N pairs=1e+04 expected precision=0.52272%
N pairs=1e+05 expected precision=0.16530%
N pairs=1e+06 expected precision=0.05227%
N pairs=1e+07 expected precision=0.01653%

0.1.2 Problem 2

[4]: # The function to integrate. Note: this operates on numpy arrays
    def func(x):
        return x*np.cos(np.sqrt(x))/np.log(2.5+x)

[5]: # We are going to try several numbers of small intervals (bins)
    Nbins = [5, 10, 20, 50, 100, 500, 1000]
    for thisN in Nbins:
        # Bin boundaries
        # Careful: for N bins, I need N+1 boundaries
        binEdges = np.linspace(0, math.pi/2, thisN+1)

        # Bin centers
        binCenters = 0.5 * (binEdges[1:] + binEdges[:-1])

        # Bin size
        deltaX = binEdges[1]-binEdges[0]

        # The value of the function at each bin center
        y = func(binCenters)

        # The (approximate) integral
        I = deltaX * y.sum()}
Using 5 bins and the midpoint rule, the integral is 0.5387052
Using 10 bins and the midpoint rule, the integral is 0.5345636
Using 20 bins and the midpoint rule, the integral is 0.5335216
Using 50 bins and the midpoint rule, the integral is 0.5332294
Using 100 bins and the midpoint rule, the integral is 0.5331877
Using 500 bins and the midpoint rule, the integral is 0.5331743
Using 1000 bins and the midpoint rule, the integral is 0.5331739

Using 5 bins and the trapezoidal rule, the integral is 0.5220789
Using 10 bins and the trapezoidal rule, the integral is 0.5303920
Using 20 bins and the trapezoidal rule, the integral is 0.5324778
Using 50 bins and the trapezoidal rule, the integral is 0.5330624
Using 100 bins and the trapezoidal rule, the integral is 0.5331459
Using 500 bins and the trapezoidal rule, the integral is 0.5331726
Using 1000 bins and the trapezoidal rule, the integral is 0.5331735

Using 5 bins and the Simpson rule, the integral is 0.5320789
Using 10 bins and the Simpson rule, the integral is 0.5303920
Using 20 bins and the Simpson rule, the integral is 0.5324778
Using 50 bins and the Simpson rule, the integral is 0.5330624
Using 100 bins and the Simpson rule, the integral is 0.5331459
Using 500 bins and the Simpson rule, the integral is 0.5331726
Using 1000 bins and the Simpson rule, the integral is 0.5331735
# Careful: for $N$ bins, I need $N+1$ boundaries
binEdges = np.linspace(0, math.pi/2, thisN+1)

# The value of the function at each bin
y = func(binEdges)

# The (approximate) integral
I = integrate.simps(y, binEdges)

# Print the result
print("Using %.i bins and the Simpson rule, the integral is %.7f" %(thisN, I))

Using 5 bins and the Simpson rule, the integral is 0.5331781
Using 10 bins and the Simpson rule, the integral is 0.5331631
Using 20 bins and the Simpson rule, the integral is 0.5331731
Using 50 bins and the Simpson rule, the integral is 0.5331737
Using 100 bins and the Simpson rule, the integral is 0.5331737
Using 500 bins and the Simpson rule, the integral is 0.5331737
Using 1000 bins and the Simpson rule, the integral is 0.5331737

0.1.3 Problem 3
# Inspired by Problem 1, I will throw random points \((x,y)\) in
# the rectangle bounded by \(x=0\rightarrow x=\pi/2\) and \(y=0 \rightarrow y=0.5\) of Area=\(\pi/4\)
# I will find the fraction of points that lie below the curve \(\text{func}(x)\)

```
# The random number seed
np.random.seed(1234756)

# The number of pairs to pick
Npairs = [1e2, 1e3, 1e4, 1e5, 1e6, 1e7]

# The area of the rectangle
Area = math.pi/4

# Loop over Npairs
for N in Npairs:
    x = (math.pi/2)*np.random.rand(int(N))  # random between 0 and \(\pi/2\)
    y = 0.5*np.random.rand(int(N))          # random between 0 and 0.5
    f = func(x)

    Inside = (f>y).sum()  # number with \(y<\text{func}(x)\)

    I = Area*Inside/N
    print("Using \%.i random throws, the integral is \%.7f\" %(N, I))
```

Using 100 random throws, the integral is 0.5890486
Using 1000 random throws, the integral is 0.5152212
Using 10000 random throws, the integral is 0.5331283
Using 100000 random throws, the integral is 0.5323979
Using 1000000 random throws, the integral is 0.5330521
Using 10000000 random throws, the integral is 0.5333678

0.1.4 Problem 4

```
# Setup the problem as \(A \times X = D\) which gives \(X = A^{-1} \times D\)
# where
# \(X\) is a column vector that contains \((x,y,z)\)
# \(A\) is a 3\times3 matrix
# \(D\) is a column vector that contains the right hand side of the three equations

# Setup the matrices
A = np.matrix([[1,1,1],[0,2,5],[2,5,-1]])
D = np.matrix([[6],[-4],[27]])
print(A)
print(‘---‘)
print(D)
```
# Invert A
invA = np.linalg.inv(A)

# Solve
X = np.matmul(invA,D)

print(type(X))
print(X)

<class 'numpy.matrix'>
[[ 5.]
 [ 3.]
 [-2.]]

# Extracting the elements of this numpy matrix X into simple numbers is ugly
# There must be a nicer way? But this works. Who am I to complain?
print("The solution is")
print("x = %.3f" % X.item(0))
print("y = %.3f" % X.item(1))
print("z = %.3f" % X.item(2))

The solution is
x = 5.000
y = 3.000
z = -2.000