Assignment - Random Number Generation

Year 2017-2018 - Semester II

CCE3501 / CCE3502

Originally devloped by - Adrian Muscat, 2018

Franklyn Sciberras, 0441498M, BSc CS, Yr II

It is desired to generate random numbers from the continuous distribution function given by,

$$f(x) = sin(\pi x)$$

defined over the interval $(0 \leq x < 1)$.

In [5]:

```
# import useful libraries
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
# this line plots graphs in line
%matplotlib inline
```

Throughout this assignment use the function

np.random.rand( )

to generate random numbers in the range [0.0,1.0)

In [6]:

```
# e.g
print np.random.rand() #generates one number
print np.random.rand(5)  # generates a vector of 5 numbers
print np.random.rand(3,3) # generates 3x3 array of numbers
```

```
0.157983539773
[ 0.34548805  0.4174302   0.12850901  0.49748599  0.39139478]
[[ 0.38363136  0.93054601  0.80659692]
 [ 0.41385481  0.19562796  0.72831498]
 [ 0.83859592  0.13684129  0.14492102]]
```

Task 1:

Develop an algorithm based on the analytical inversion technique to generate random numbers from $f(x)$.

Add your answer in the Markdown cell below

- (Check that Continuous Distribution is proper).
- Obtain Cumulative Distribution Function.
- Obtain Inverse Cumulative Distribution Function.
- Generate a random value U in the interval of [0,1], from a uniform distribution.
- Input U in the Inverse Cumulative Distribution Function and return the randomly generated number.

Task 2:

Code the algorithm in Python

In [7]:

```
#Checking continuous distribution for proper property

#importing integration function
import scipy.integrate as integrate

#constant for function is by default 1
constant = 1

#defining continuous function given
def f_x(x):
    return constant * np.sin(np.pi*x)

#check whether it is a proper function in the given bounds
#(if not make it)
area, error = integrate.quad(f_x,0,1)

if area != 1:
    constant = area**-1

area2,error2 = integrate.quad(f_x,0,1)

print ("Area of proper function: ")
print area2
print ("Constant added to make function proper: ")
print constant
```

```
Area of proper function:
1.0
Constant added to make function proper:
1.57079632679
```

In [8]:

```
#Obtaining Cumulative Distribution Function for Proper Distribtuion

#Integration of f_x(x) is -0.5cos(pi*x)
def integrated_f_x(x):
    return -0.5* np.cos(np.pi*x)
```

In [9]:

```
#Obtain Inverse Cumulative Distribution Function for Proper Distirbution

#Inverse of Integration of f_x(x) is ±(ArcCos(-2x)/pi)
#however this is not within required bounds, and thus we do a time shift
#Thus our cdf will be: ±(ArcCos(1-2x)/pi)
def inv_integrated_f_x(x):
    return np.arccos(1-(2*x))/np.pi
```

In [10]:

```
#Generating a random number from distribution, using uniformly distributed
random number generator
def AIM_generator():
    return inv_integrated_f_x(np.random.rand())

def print_random():
    print ("Random Number Generated using AIM: ")
    print AIM_generator()

print_random()
```

```
Random Number Generated using AIM:
0.512722604852
```

Task 3:

Using your mathematical algorithm generate 1000 numbers

Plot a histogram that represents the distribution (use 11 bins)

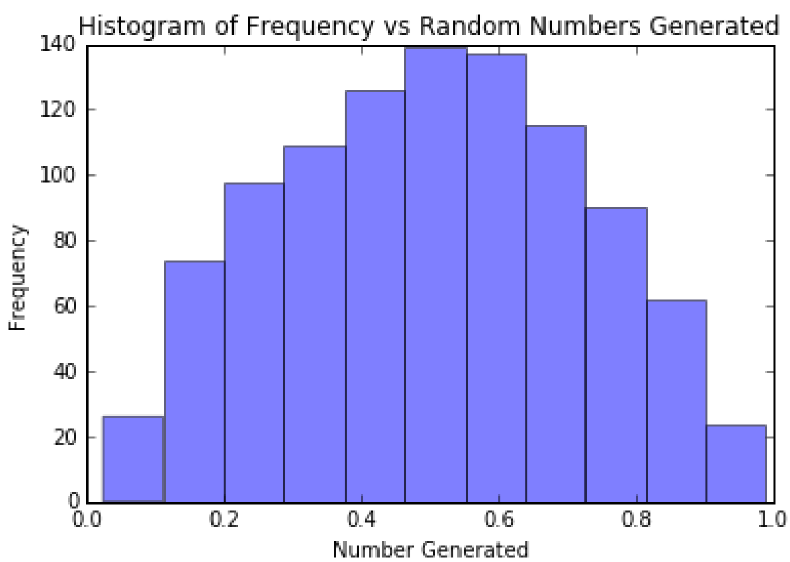
Obtain a goodness of fit statistic for the distribution

Plot a two variable scatter-plot to demonstrate that the numbers are not correlated

In [11]:

```
#Generating 1000 random numbers and storing them in an array
def generate_1000_AIM():
    random_numbers_gen = []
    for i in range (0, 1000):
        random_numbers_gen.append(AIM_generator())
    return random_numbers_gen

#Generating Expected Frequency - 11 times since 11 bins
#We use the given function const*sin(pi*x) in order to obtain these
#results. (by calculating area with limits, depending on bins req^d)
def generate_expected_freq_AIM():
    random_numbers_exp = []
```

```
    lower_lim = 0.0
    upper_lim = 0.0
    for i in range (0,11):
        lower_lim = upper_lim
        upper_lim += 1.0/11.0
        area, error = integrate.quad(f_x,lower_lim,upper_lim)
        random_numbers_exp.append(1000*area)
    return random_numbers_exp

#Generating Histogram and plotting it
generated_list_AIM = generate_1000_AIM()
AIM_random_numbers_obs, bins_obs = np.histogram(generated_list_AIM, 11)
plt.hist(generated_list_AIM, 11, facecolor='blue', alpha=0.5)
plt.title("Histogram of Frequency vs Random Numbers Generated")
plt.xlabel("Number Generated")
plt.ylabel("Frequency")
plt.show()

#Calculating Chi Squared Value using : Sum((Observed-Expected)^2/Exp)
chi_square_obtained_AIM = 0.0
expected_list_AIM = generate_expected_freq_AIM()
for i in range(0, len(expected_list_AIM)):
    chi_square_obtained_AIM += ((AIM_random_numbers_obs[i] -
expected_list_AIM[i])**2)/expected_list_AIM[i]

print ("Chi-Square Value: ")
print chi_square_obtained_AIM

#Degrees of freedom are calculated by number of bins - 1, thus df = 10
#Calculating Crtical Value to test whether to accept the null hypothesis
#aka. both distributions are the same
critical_val = stats.chi2.ppf(q = 0.95, df = 10)

print ("Critical Value: ")
print critical_val

#Checking whether to accept the hypothesis or not
if chi_square_obtained_AIM > critical_val:
    print("Hypothesis Rejected: The distributions are not similar.")
else:
    print("Hypothesis Accepted: The distributions are similar with a 95% co
nfidence level.")
```

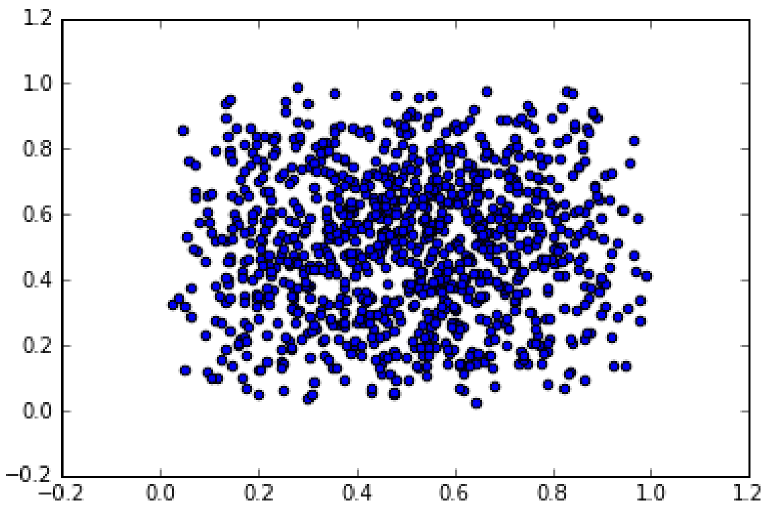


```
Chi-Square Value:
8.60683582903
Critical Value:
18.3070380533
Hypothesis Accepted: The distributions are similar with a 95% confidence le
vel.
```

In [14]:

```
#Checking Correlation between two generations using Pearson Correlation
def shifting_gen_list(arraylist):
    array = []
    i = 0
    array.append(arraylist[len(arraylist)-1])
    while i < len(arraylist) - 1:
        array.append(arraylist[i])
        i+=1
    return array

generated_list_AIM_2 = shifting_gen_list(generated_list_AIM)
coeff, two_tailed_p = stats.pearsonr(generated_list_AIM,
generated_list_AIM_2)

print "Level of Correlation Between Observed and Expected: " , coeff

if coeff > 0.25:
    print "Positively correlated"
elif coeff< -0.25 :
    print "Negatively correlated"
else:
    print "Not correlated"

#Plotting Scatter Plot
plt.scatter(generated_list_AIM,generated_list_AIM_2)
plt.show()
```

```
Level of Correlation Between Observed and Expected:  0.0368273007524
Not correlated
```



Task 4:

Write down an algorithm (in pseudo code) based on the

accept-reject method to generate random numbers described by $f(x)$.

Add your answer below in the Markdown cell below

- (Check that Continuous Distribution is proper).
- Define a function h where h(t) > f(t) and max of h(t)=m (y is the proper distribution).
- i) Generate a random value U in the given range, using a uniform distribution.
- ii) Generate a random value V in the interval of [0,m], using a uniform distribution
- ii) Input U in the distribtuion y. If V < f(U), accept U, else go to i) and repeat.

Task 5:

Code the algorithm in Python

In [10]:

```
#uses accept-reject method to generate random numbers
def ARM_generator():
    #since our function oscillates between 0 and 1 in the range of 0 and 0
and pi/2
    m = 2
    h_x = m
    while(True):
        U = np.random.rand()
        V = np.random.rand() * m
        generated = f_x(U)
        if V < generated:
            return U

print ARM_generator()
```

0.54885816738

Task 6

Using your accept-reject algorithm generate 1000 numbers

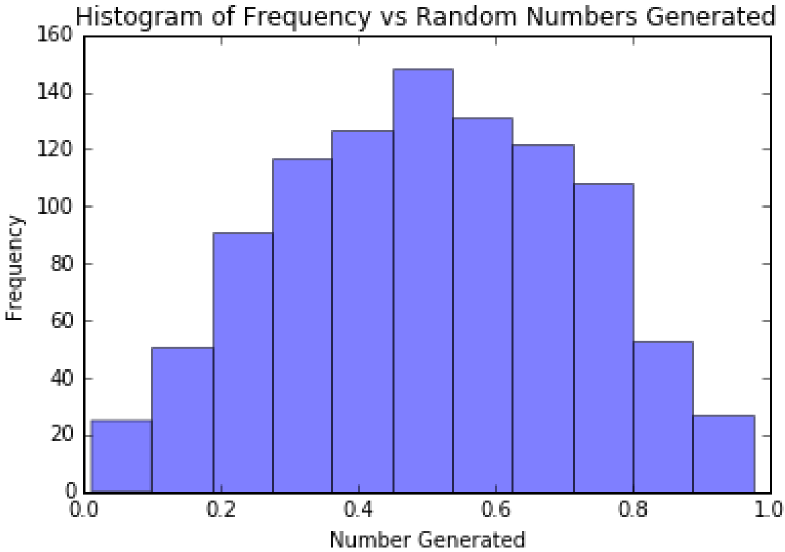
Plot a histogram that represents the distribution (use 11 bins)

Obtain a goodness of fit statistic for the distribution

In [18]:

```
#Generating 1000 numbers
def generate_1000_ARM():
    ARM_gen = []
    for f in range(0,1000):
```

```
        ARM_gen.append(ARM_generator())
    return ARM_gen

generated_list_ARM = generate_1000_ARM()
#Generating Histogram and plotting it
ARM_random_numbers_obs, ARM_bins_obs = np.histogram(generated_list_ARM, 11)
plt.hist(generated_list_ARM, 11, facecolor='blue', alpha=0.5)
plt.title("Histogram of Frequency vs Random Numbers Generated")
plt.xlabel("Number Generated")
plt.ylabel("Frequency")
plt.show()

#Calculating Chi Squared Value using : Sum((Observed-Expected)^2/Exp)
chi_square_obtained_ARM = 0.0
for i in range(0, len(expected_list_AIM)):
    chi_square_obtained_ARM += ((ARM_random_numbers_obs[i] -
expected_list_AIM[i])**2)/expected_list_AIM[i]

print expected_list_AIM

print "Chi Square Obtained: ", chi_square_obtained_ARM

print "Critical Value: ", critical_val

#Checking whether to accept the hypothesis or not
if chi_square_obtained_ARM > critical_val:
    print("Hypothesis Rejected: The distributions are not similar.")
else:
    print("Hypothesis Accepted: The distributions are similar with a 95% co
nfidence level.")
```




```
[20.25351319275131, 59.119720391658106, 93.19639944294802,
119.72286047169935, 136.55008736430068, 142.31483827328518,
136.55008736430065, 119.72286047169933, 93.19639944294799,
59.11972039165809, 20.253513192751214]
Chi Square Obtained:  8.73735549135
Critical Value:  18.3070380533
Hypothesis Accepted: The distributions are similar with a 95% confidence le
vel.
```

Experimentally measure the computational time for both algorithms and test for any significant difference

algorithms and test for any significant difference

In [12]:

```
import time as t
#Computing time taken for the algorithm to compute, multiple times
def compute_time(algorithm):
    computation_time = []
    for _ in range (0, 1000):
        start = t.time()
        algorithm()
        end = t.time()
        computation_time.append((end-start)/1000)
    return (computation_time)

time_AIM = np.array(compute_time(generate_1000_AIM))
time_ARM = np.array(compute_time(generate_1000_ARM))

#Computing average time taken to find 1000 random numbers, per respective algorithm
avrg_time_AIM = np.average(compute_time(generate_1000_AIM))
avrg_time_ARM = np.average(compute_time(generate_1000_ARM))

print "Time taken for AIM to generate 1000 values: ", avrg_time_AIM
print "Time taken for ARM to generate 1000 values: ", avrg_time_ARM

#Comparing difference in means using one-way anova
f_val, p_val = stats.f_oneway(time_AIM, time_ARM)

if p_val > 0.05:
    print "There is no significant difference between the two alogrithms"
else:
    print "The two algorithms are 95% of the time different, with their variation in computational time being: ", f_val
```

```
Time taken for AIM to generate 1000 values:  3.16356873512e-06
Time taken for ARM to generate 1000 values:  6.60986423492e-06
1.13434178991e-248
The two algorithms are 95% of the time different, with their variation in computational time being:  1527.49651089
```