Siddhanth Kumar Section 102

Workshop 7: Monte Carlo techniques

Submit this notebook to bCourses to receive a grade for this Workshop.

Please complete workshop activities in code cells in this iPython notebook. The activities titled **Practice** are purely for you to explore Python. Some of them may have some code written, and you should try to modify it in different ways to understand how it works. Although no particular output is expected at submission time, it is *highly* recommended that you read and work through the practice activities before or alongside the exercises. However, the activities titled **Exercise** have specific tasks and specific outputs expected. Include comments in your code when necessary. Enter your name in the cell at the top of the notebook.

The workshop should be submitted on bCourses under the Assignments tab (both the .ipynb and .pdf files).

In [1]:
```
# standard preamble
import numpy as np
import scipy as sp
from scipy import stats
import matplotlib.pyplot as plt
%matplotlib inline
```

So far, you have seen the following functions for drawing random numbers from various distributions:

```
np.random.rand()
np.random.randint()
np.random.uniform()
np.random.standard_normal()
np.random.exponential()
np.random.binomial()
np.random.poisson()
```

See examples in `Lecture07.ipynb` and the material from last two workshops. Now let's see how to sample probability distributions for which a function may not already exist.

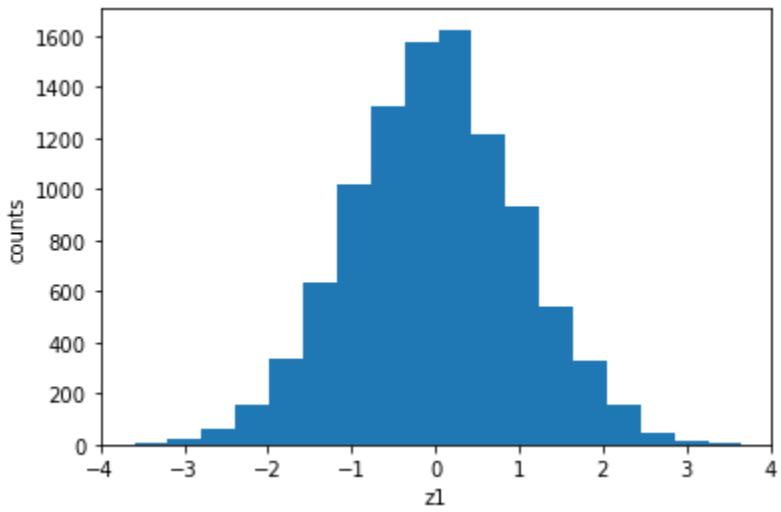
Exercise 1: Generating an arbitrary distribution

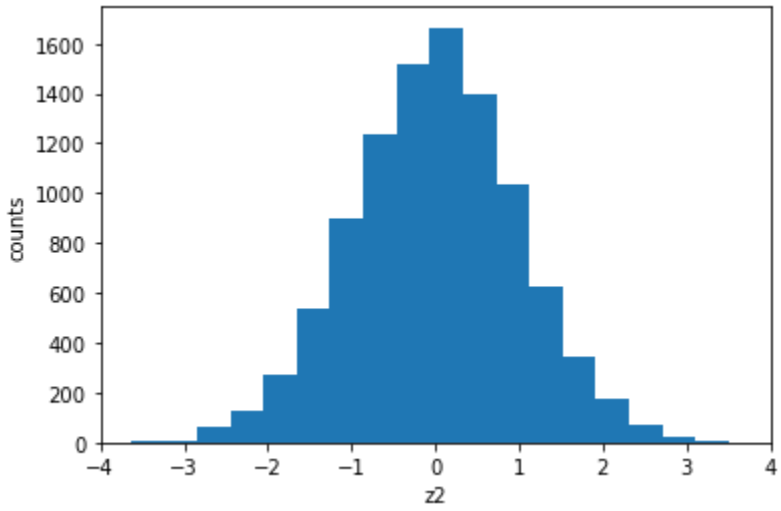
A problem that is frequently encountered is one of sampling one distribution but only having direct access to sampling another distribution (often the uniform distribution). As an example, we will try out one way to draw numbers from a Gaussian distribution starting with numbers drawn only from a uniform distribution. As it turns out, there are multiple ways of doing this. This particular method is known as the "Box-Muller transform".

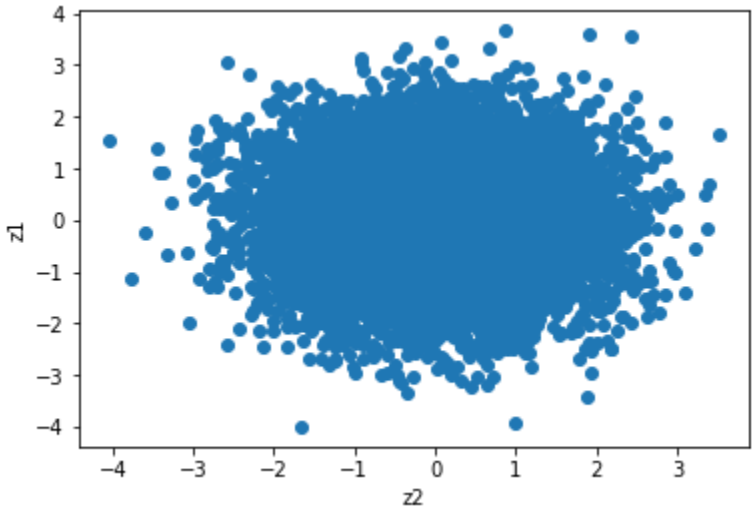
1. Starting from a uniform random number distribution (`numpy.random.rand()`), generate 10,000 Gaussian-distributed random numbers using inverse transform method:
   A. Generate a pair of uniform-distributed numbers $u_1 \in [0..1]$ and $u_2 \in [0..1]$
   B. Compute $z_1 = \sin(2\pi u_1)\sqrt{-2\ln u_2}$ and $z_2 = \cos(2\pi u_1)\sqrt{-2\ln u_2}$
2. Make histograms of z_1 and z_2.

3. Make a scatter plot of z_2 vs z_1.
4. Do the random variables z_2 and z_1 appear to follow a Gaussian distribution?

In [25]:
```
import numpy as np
import matplotlib.pyplot as plt

u1 = np.random.rand(10000)
u2 = np.random.rand(10000)

z1 = []
z2 = []

for i in range(10000):
    z1calc = (np.sin(2 * np.pi * u1[i])) * (np.sqrt(-2 * np.log(u2[i])))
    z1.append(z1calc)
    z2calc = (np.cos(2 * np.pi * u1[i])) * (np.sqrt(-2 * np.log(u2[i])))
    z2.append(z2calc)

plt.hist(z1, bins=19)
plt.xlabel('z1')
plt.ylabel('counts')
plt.xlim([-4, 4])
plt.show()
plt.hist(z2, bins=19)
plt.xlabel('z2')
plt.ylabel('counts')
plt.xlim([-4, 4])
plt.show()

plt.scatter(z2, z1)
plt.xlabel('z2')
plt.ylabel('z1')
plt.show()

# Remember numpy has all the math function you need: np.sin, np.cos, np.sqrt, np.log
```



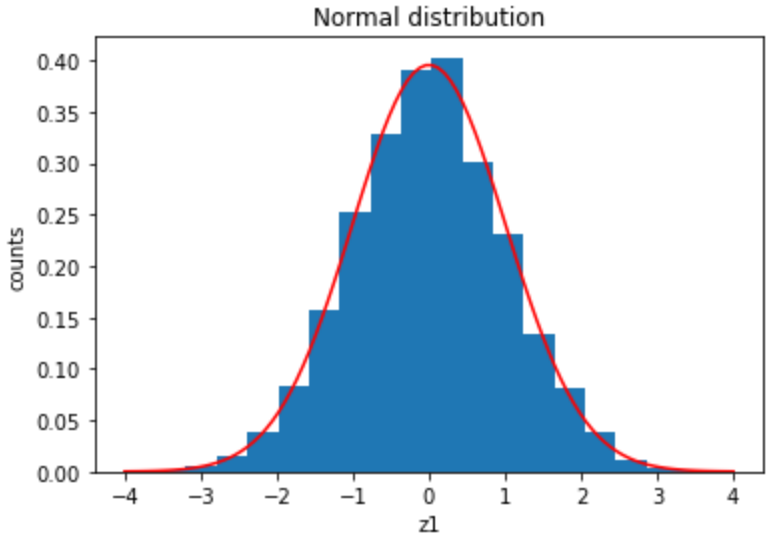




(Optional) If you want to get ahead for the next homework assignment, try using `scipy.stats.norm.fit()` to fit the z_1 and/or z_2 distributions. This function will return the best-fit values for μ and σ (which should be about 0 and 1, respectively).

In [28]:
```
from scipy.stats import norm

par = norm.fit(z1)

print ('mean  = {0:.8f}'.format(par[0]))
print ('sigma = {0:.8f}'.format(par[1]))


x = np.linspace(-4,4,100)

# fitted distribution
pdf_fitted = norm.pdf(x,loc=par[0],scale=par[1])

plt.title('Normal distribution')
plt.plot(x,pdf_fitted,'r-')
plt.hist(z1,bins=19, density=True)
plt.xlabel('z1')
plt.ylabel('counts')
plt.show()

# If you'd like, fit the z1 and/or z2 distribution using norm.fit()
```

```
mean  = -0.00087088
sigma = 1.00802940
```



Exercise 2: Integration by accept-reject Monte Carlo method

The "accept-reject" method is another way of simulating drawing from one probability distribution starting from sampling a uniform distribution. Here we explore this technique for finding the area under curves using random numbers. Although in this exercise we will only use it to estimate the value of π, you can use it for more sophisticated problems.

Before we start coding, let us just look at a quarter circle inscribed inside a square of dimensions 1 by 1.

If we were to throw a dart randomly, with uniform probability, into the square, what is the probability it would land in the shaded region under the curve? It should be the ratio of the shaded area to the area of the entire square.

$$P(\text{dart under curve}) = \frac{A_{\text{curve}}}{A_{\text{square}}}$$

$$A_{\text{curve}} = \frac{1}{4}\pi \times 1^2$$

$$A_{\text{square}} = 1 \times 1$$

$$P(\text{dart under curve}) = \frac{\pi}{4}$$

We also know that if we were to throw a very large number of darts, the fraction of them that land in the shaded region would approach the value of $P(\text{dart under curve})$ found above. Therefore, if we can simulate throwing these darts and checking whether they landed inside the shaded region or outside the shaded region, we can estimate the area of the shaded region. In this case, the area of the shaded region is proportional to π, so we can then use that area to estimate π.

This technique of randomly generating points and checking whether they are under the curve or above the curve is referred to as $Monte\ Carlo\ integration$. Compute the value of π using Monte Carlo method.

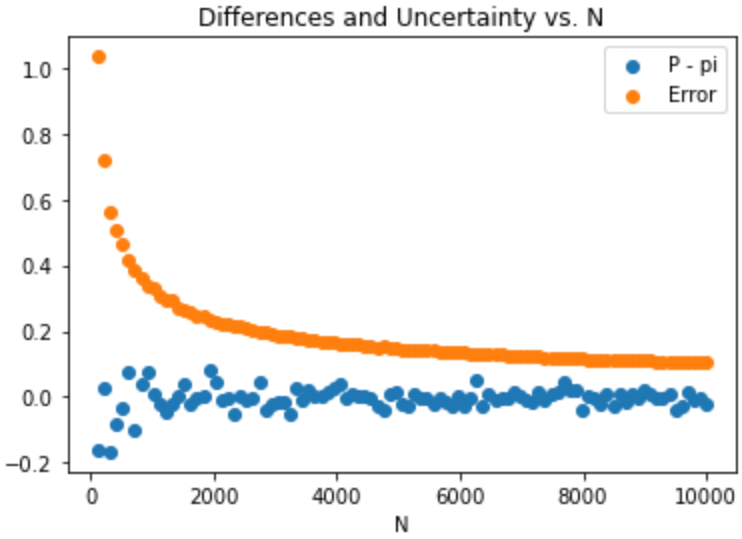
1. Implement the Monte Carlo accept-reject method for computing π
2. For a given number of events N you use in the calculation, compute
    A. The estimate of π

B. The estimated precision of the value π

3. Plot the difference between estimated and true value of π as a function of the number of events N and compare that difference to the uncertainty you estimated

In [78]:
```
import numpy as np
import matplotlib.pyplot as plt

def predictFunc(N):

    x = np.random.rand(N)
    y = np.random.rand(N)

    pIN = 0
    pOUT = 0

    for i in range(N):
        calc = (x[i]**2) + (y[i]**2)
        if (calc < 1 or calc == 1):
            pIN += 1
        else:
            pOUT += 1

    return (pIN/(pIN + pOUT))*4


def calc_error(N):
    variance = 16 * 1/N * predictFunc(N)*(1.0-predictFunc(N))
    error = np.sqrt(np.sqrt(variance**2))
    return error


value = predictFunc(10000)
error = calc_error(10000)

print('{0:.3} +/- {1:.3}'.format(value, error))

# Don't use np.pi until you are comparing your estimate to the true value

Ns = np.linspace(10, 10000, 100)
vals = []
errors = []

for value in Ns:
    vals.append(    (predictFunc(int(value)) - np.pi)   )
    errors.append(calc_error(int(value)))


Ns = np.delete(Ns, 0)
vals.pop(0)
errors.pop(0)


plt.scatter(Ns, vals)
plt.scatter(Ns, errors)
plt.legend(['P - pi', 'Error'])
plt.title('Differences and Uncertainty vs. N')
plt.xlabel('N')
plt.show()
```

```
3.18 +/- 0.104
```


Differences and Uncertainty vs. N
P - pi
Error
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
2000
4000
6000
8000
10000
N

In [ ]: