Monte Carlo Methods

Metropolis Monte Carlo for Bayesian Inference

Anand Kamble
amk23j@fsu.edu

7th November 2023

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import Markdown, display, Latex
```

$$p_{skew} = \frac{p_{true}}{2} + \frac{1}{4}$$

In [2]:
```
def pskew(ptrue):
    return (ptrue/2)+0.25
```

$$log\pi(p_{true}|X) = Xlogp_{skew} + (N - X)log(1 - p_{skew}) + constant$$

In [3]:
```
def dist(ptrue, X = 35, N = 100, c = 0):
    a = np.log(pskew(ptrue))*(N - X)*np.log(1. - pskew(ptrue))
    return a if 0 <= ptrue <= 1 else 0
```

In [4]:
```
def proposal(oldx, delta):
    newx = oldx + np.random.uniform(-delta, delta)
    return newx
```

In [5]:
```
def metropolis_accept(newx, oldVal, func):
    newVal     = func(newx)
    ratio = np.exp(newVal - oldVal+ 1.0e-21)
    accept = ratio > 1. or np.random.rand() < ratio
    return accept, newVal
```

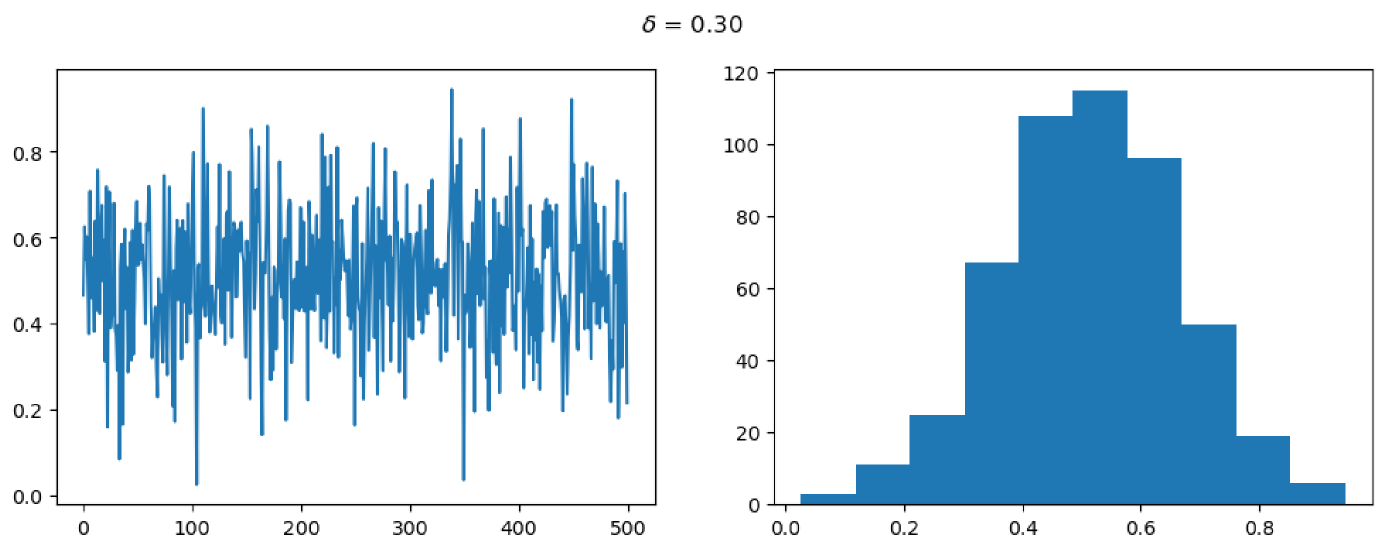
In [6]:
```
def driver(delta, nsteps=10000, thin=10):
    x         = 0.35
    f         = dist(x)
    AccRatio  = 0.0
    NumSucc   = 0
    recz      = np.zeros((int(nsteps/thin)))
    for iMCS in range(nsteps):

        newx = proposal(x, delta)
        accept, newf = metropolis_accept(newx, f, dist)

        if accept:
            NumSucc += 1
            x = newx
            f = newf

        if (iMCS % thin) == 0:
            recz[int(iMCS/thin)] = x
```

```python
    AccRatio = float(NumSucc)/float(nsteps)

    return recz, AccRatio
```

In [7]:
```python
def plotChainDist(delta, nsteps=5000):
    recz, ar = driver(delta, nsteps)
    fig, axs = plt.subplots(1, 2, figsize=(12,4))

    axs[0].plot(recz,'-')
    axs[1].hist(recz)

    fig.suptitle('$\delta$ = {0:0.2f}'.format(delta))
    print("axRatio\t{0:0.4f}".format(ar))
```

In [8]:
```python
plotChainDist(0.3)
```

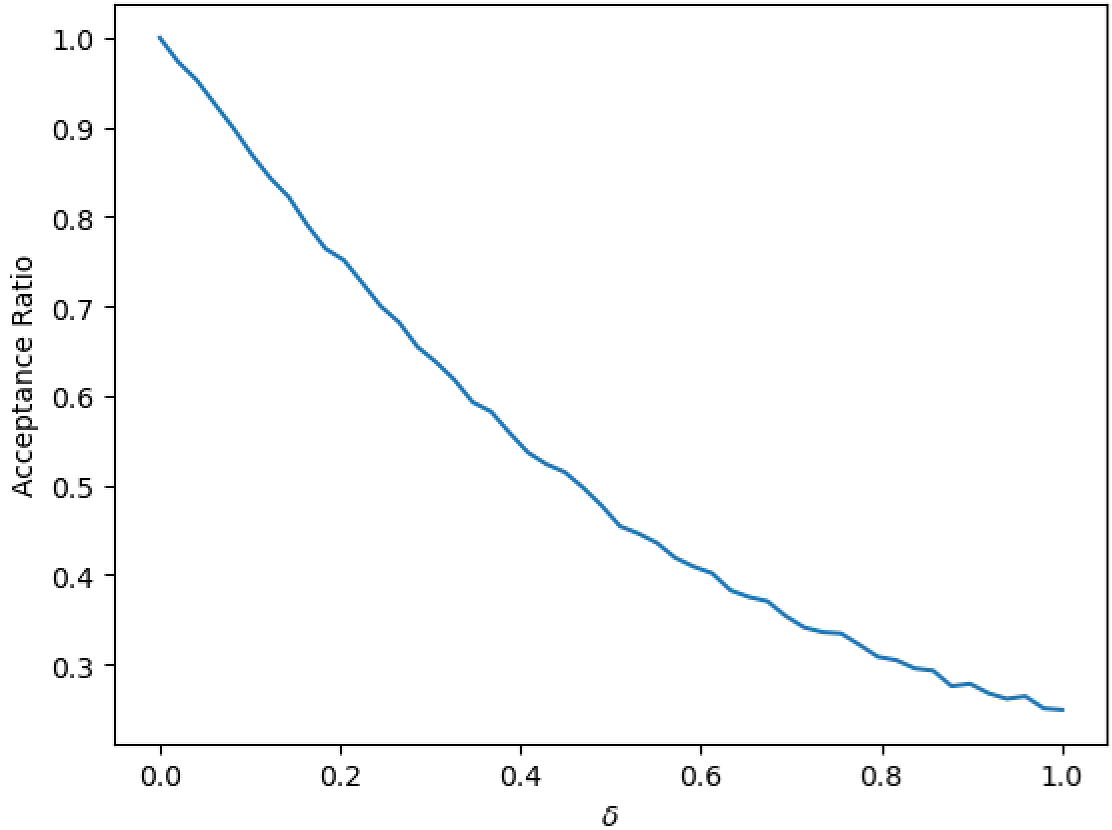
```
axRatio 0.6486
```



Varying Step Size (δ), and observing the changes in acceptance ratio.

In [9]:
```python
x = np.linspace(0,1,50)
ars = list()
for xi in x:
    recz, ar = driver(xi)
    ars.append(ar)

plt.xlabel("$\delta$")
plt.ylabel("Acceptance Ratio")
plt.plot(x,ars);
```

```
C:\Users\91911\AppData\Local\Temp\ipykernel_17192\461230281.py:2: RuntimeWarning: invalid value encountered in log
  a = np.log(pskew(ptrue))*(N - X)*np.log(1. - pskew(ptrue))
```

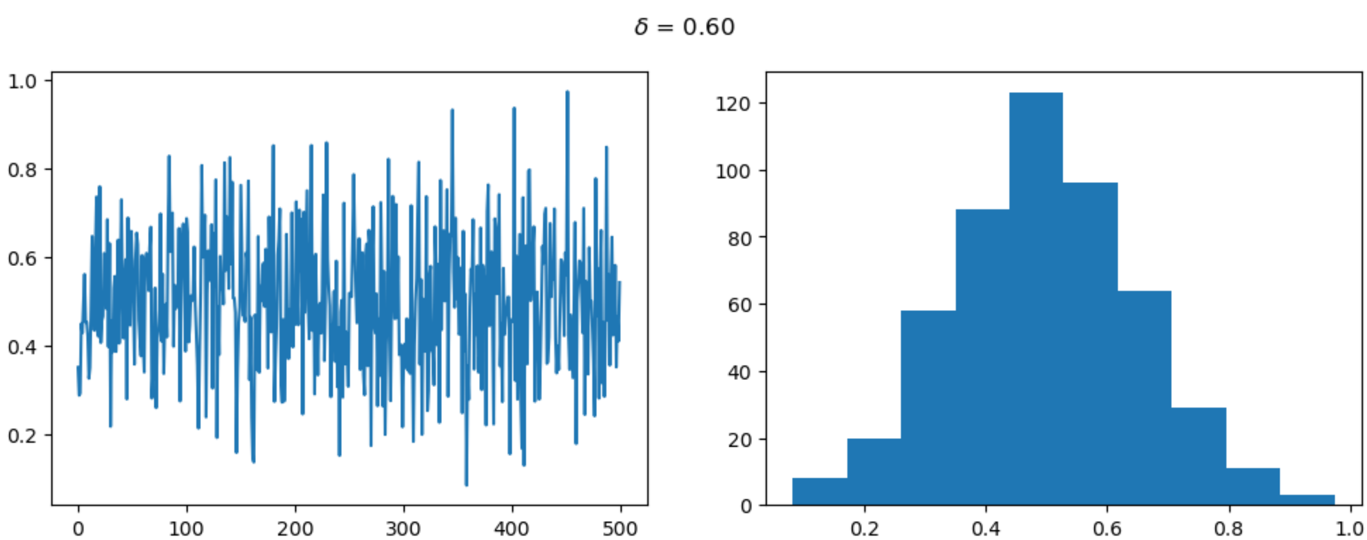


In the diagram, we can see that as the delta increases from 0 to 1, the acceptance ratio decreases.

δ that yields an acceptance ratio of 0.4 ± 0.1.

In [10]:
```
delta = 0.6
plotChainDist(delta)
```

```
C:\Users\91911\AppData\Local\Temp\ipykernel_17192\461230281.py:2: RuntimeWarning: invali
d value encountered in log
  a = np.log(pskew(ptrue))*(N - X)*np.log(1. - pskew(ptrue))
axRatio 0.3986
```



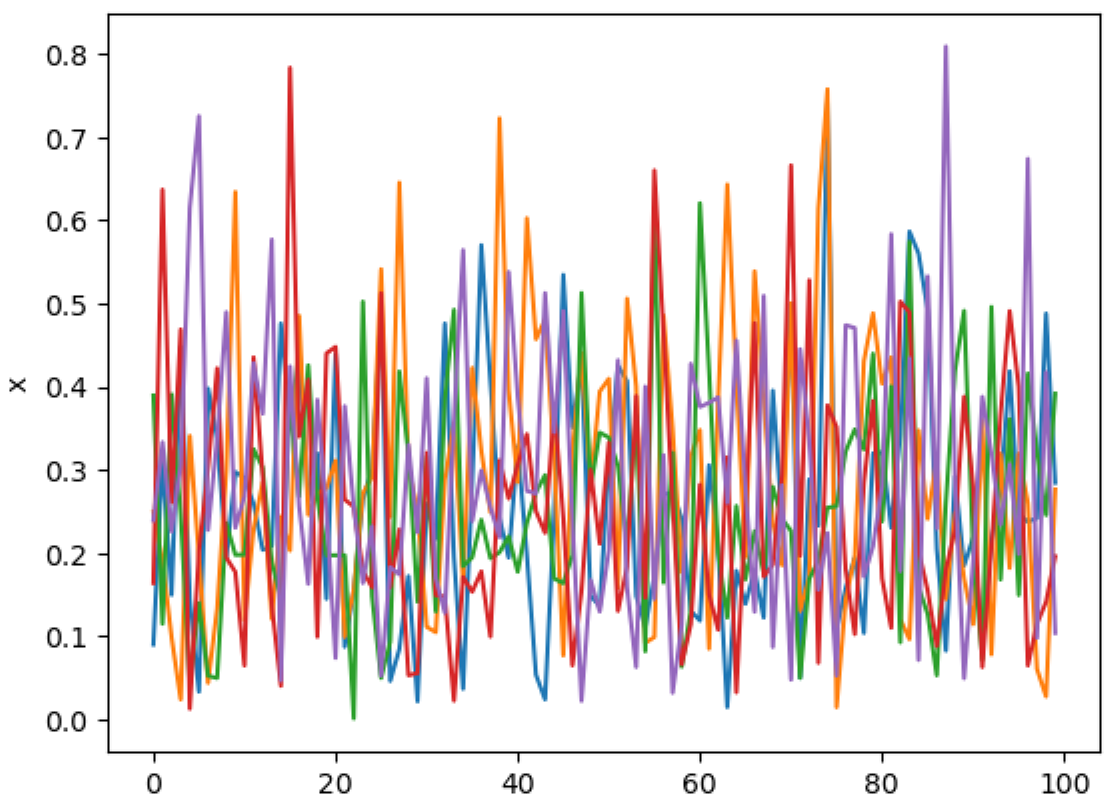
Running the MCMC simulation long enough to ensure convergence

In [11]:
```
def GelmanRubin(A, M, n):
```

```
    sj2 = np.zeros(n); aj  = np.zeros(n)
    for j in range(n):
        sj2[j] = np.var(A[:,j])
        aj[j]  = np.mean(A[:,j])

    W  = np.mean(sj2) # within-chain
    B  = M * np.var(aj)
    s2 = (1. - 1./M)*W + 1./M * B # inter-chain
    R  = np.sqrt(s2/W)

    return R, s2, W, B

n     = 5
thin  = 10
M     = 100
thin  = 10
delta = 0.5

X = np.zeros((M, n))

for j in range(n):
    c, ar = driver(delta, 2*M*thin, thin)
    X[:,j] = c[M:2*M]**2 # + c[M:2*M, 1]**2     # after burnin

    plt.plot(X[:,j])

plt.ylabel('x')
R,s2, W, B = GelmanRubin(X, M, n)
display(Latex("$\hat{R} = {" + str(R) + "}$"))
display(Latex("This ensures that the we h`ave ran the simulation long enough to converge
```

$\hat{R} = 1.001837975384077$

This ensures that the we h`ave ran the simulation long enough to converge. $(\hat{R} < 1.1)$



Finding the mean and standard deviation.

In [12]:

```
recz, ar = driver(0.5, 5000)

mean_estimate = np.mean(recz)
std_dev_estimate = np.std(recz, ddof=1)

display(Markdown(f"#### Mean: {mean_estimate}"))
display(Markdown(f"#### Standard Deviation: {std_dev_estimate}"))
```

Mean: 0.4973080126237827

Standard Deviation: 0.15764264799387026