In [1]:

```
# getting the libraries that has some of the functions needed for simulations
import numpy as np
import matplotlib.pyplot as plt
from numba import jit
import time
from numba import njit, prange
import pandas as pd

%matplotlib inline
```

In [2]:

```
@jit(nopython=True)
def cdf_inv_fr(u, gamma):
  # function for getting inverse of CDF
  return ((pow(-np.log(u) , -gamma)))

@jit(nopython=True)
def cdf_inv_H(u , gamma, beta):
  #getting inverse of the function defined by H in the paper
  return (pow(-np.log(u)/(pow(beta , -1/gamma)-1) , -gamma) )

@jit(nopython=True)
def armax(beta , gamma, n):
  # simulating the armax distribution

  # declaring an array
  x = np.zeros(n)

  # declaring uniform distribution
  u = np.random.uniform(0,1,1)[0]
  # applying cdf_inv_fr
  x0 = cdf_inv_fr(u,gamma)
  xi_lag = x0
  x[0] = x0
  t = 1

  # getting the armax, ie, (max of Xi, Zi) * beta
  for i in range(n-1):
    u = np.random.uniform(0,1,1)[0]
    zi = cdf_inv_H(u,gamma,beta)
    xi = beta*max(xi_lag , zi)
    xi_lag = xi
    x[t] = xi
    t = t + 1
  return x
```

In [3]:

```
@jit(nopython=True)
def theta_n_k(X, n, k_=1):
  sum = 0
  k = int(k_)

  # getting the ascending n-k th order statistic
  X_k = np.partition(X, n-k-1)[n-k-1]

  for j in range(n-1):
    # k-th top order equals n-k low order
    if X[j] <= X_k and X[j+1] > X_k:
      sum += 1

  if k == 0:
    return 1
  return sum/k
```

```
@jit(nopython=True)
def theta_GJ_k(X, n, k, delta):

  numerator = (delta*delta + 1) * theta_n_k(X,n, int(np.floor(delta*k)) + 1) - delta*(th
eta_n_k(X, n, int(np.floor(delta*delta*k)) + 1) + theta_n_k(X, n, k))
  denominator = (1 - delta)**2

  if numerator < 0:
    return 0

  return numerator/denominator
```

In [4]:

```
@jit(nopython = True)
def np_apply_along_axis(func1d, axis, arr):
  # function as a workaround for a problem in calculating mean
  assert arr.ndim == 2
  assert axis in [0, 1]
  if axis == 0:
    result = np.empty(arr.shape[1])
    for i in range(len(result)):
      result[i] = func1d(arr[:, i])
  else:
    result = np.empty(arr.shape[0])
    for i in range(len(result)):
      result[i] = func1d(arr[i, :])
  return result
```

In [5]:

```
@jit(nopython=True)
def simulate_mean_mse(n, theta, runs = 15, replicates = 10):
  all_values_mean = np.zeros((runs, n))
  all_values_mse = np.zeros((runs, n))

  for run in range(runs):
    a = np.zeros((replicates, n))
    for i in range(replicates):
      a[i] = armax(1-theta, 1, n)

    path = np.zeros((replicates, n))
    path2 = np.zeros((replicates, n))

    k_range = list(range(n))

    for k in k_range:
      for j in range(replicates):
        path[j][k] = theta_n_k(a[j], n, k)
        path2[j][k] = (path[j][k] - theta)**2

    all_values_mean[run] = np_apply_along_axis(np.mean, 0, path)
    all_values_mse[run] = np_apply_along_axis(np.mean, 0, path2)

  return np_apply_along_axis(np.mean, 0, all_values_mean), np_apply_along_axis(np.mean,
0, all_values_mse)
```

In [6]:

```
@jit(nopython=True)
def simulate_mean_mse_GJ(n, theta, delta = 0.25, runs = 15, replicates = 10):
  all_values_mean = np.zeros((runs, n))
  all_values_mse = np.zeros((runs, n))

  for run in range(runs):
    a = np.zeros((replicates, n))
    for i in range(replicates):
      a[i] = armax(1-theta, 1, n)

    path = np.zeros((replicates, n))
```

```
    path2 = np.zeros((replicates, n))

    k_range = list(range(n))

    for k in k_range:
      for j in range(replicates):
        path[j][k] = theta_GJ_k(a[j], n, k, delta)
        path2[j][k] = (path[j][k] - theta)**2


    all_values_mean[run] = np_apply_along_axis(np.mean, 0, path)
    all_values_mse[run] = np_apply_along_axis(np.mean, 0, path2)

  return np_apply_along_axis(np.mean, 0, all_values_mean), np_apply_along_axis(np.mean,
0, all_values_mse)
```

In [7]:

```
@njit
def subsampling_GJ(X, n, theta, T = 2, delta = 0.25):
  r = int(np.floor(n/T))

  theta_Vi = np.zeros((T, r))

  #X = armax(1- theta, 1, n)

  for i in range(T):
    V = X[i:(r-1)*T+i+1:T]

    for j in range(1, r):
      theta_Vi[i][j] = theta_GJ_k(V, V.shape[0], j, delta)

  theta_sub = -1 * np.ones(n)

  for j in range(1, r):
    #temp = 1 - np.power(abs(1 - theta_Vi[:,j]), 1/T).mean(axis = 0)
    theta_sub[(j - 1)*T + 1] = 1 - ((np.power(1 - theta_Vi[:,j], 1/T)).mean())
    theta_sub[np.isnan(theta_sub)] = 0

  for k in range(1, n):
    if theta_sub[k] == -1:
      theta_sub[k] = theta_sub[k-1]

  return theta_sub
```

In [8]:

```
@njit
def simulate_mean_mse_GJ_sub(n, T = 2, theta = 0.2, delta = 0.25, runs = 15, replicates
= 10):
  all_values_mean = np.zeros((runs, n))
  all_values_mse = np.zeros((runs, n))

  for run in range(runs):
    #print("\nrun = ", run + 1, "/", runs ,"...", sep = "", end = " ")
    a = np.zeros((replicates, n))
    for i in range(replicates):
      a[i] = armax(1-theta, 1, n)

    path = np.zeros((replicates, n))
    path2 = np.zeros((replicates, n))

    k_range = list(range(n))


    for j in range(replicates):
      path[j] = subsampling_GJ(a[j], n, theta, T, delta)
      path2[j] = np.power((path[j] - theta), 2)

    #one_run_mean = np.mean(path, axis = 0)
```

```
    all_values_mean[run] = np.apply_along_axis(np.mean, 0, path)
    all_values_mse[run] = np.apply_along_axis(np.mean, 0, path2)

  return np.apply_along_axis(np.mean, 0, all_values_mean), np.apply_along_axis(np.mean,
0, all_values_mse)
```

In [9]:

```
@njit
def reff(mse_n, mse_gj):
  return np.sqrt(mse_n/mse_gj)

@njit
def bri(theta_n, theta_gj, theta):
  return np.abs((theta_n - theta)/(theta_gj - theta))

@njit
def sti(e_n, e_gj, theta, n, senstivity = 0.01):
  numerator = 0
  denominator = 0
  for i in range(n - 1):
    if np.abs(e_n[i] - theta) <= senstivity:
      denominator += 1
    if np.abs(e_gj[i] - theta) <= senstivity:
      numerator += 1
  return numerator/denominator
```

In [10]:

```
%%time
ns = [1000, 2000]
Ts = [2, 3, 4]

runs = 5000
replicates = 10
delta = 0.25
theta = 0.2

kns = np.zeros((len(Ts), len(ns)))
kgjs = np.zeros((len(Ts), len(ns)))
ens = np.zeros((len(Ts), len(ns)))
egjs = np.zeros((len(Ts), len(ns)))
mses = np.zeros((len(Ts), len(ns)))
msegjs = np.zeros((len(Ts), len(ns)))
reffs = np.zeros((len(Ts), len(ns)))
bris = np.zeros((len(Ts), len(ns)))
stis = np.zeros((len(Ts), len(ns)))

start = time.time()
i = 0
for T in Ts:
  j = 0
  for n in ns:
    print("T = {0} and n = {1}".format(T, n))

    e_n, mse_n = simulate_mean_mse(n, theta, runs, replicates)
    e_gj, mse_gj = simulate_mean_mse_GJ_sub(n, T, theta, delta, runs, replicates)

    k_n = np.where(mse_n == mse_n.min())[0][0]
    k_gj = np.where(mse_gj == mse_gj.min())[0][0]

    print("k/n\t{0:.4f}".format((k_n + 1)/n) )
    print("k_gj/n\t{0:.4f}".format((k_gj+1)/n))

    kns[i][j] = (k_n + 1)/n
    kgjs[i][j] = (k_gj + 1)/n

    print("E_N\t{0:.4f}".format(e_n[k_n]))
    print("E_GJ\t{0:.4f}".format(e_gj[k_gj]))

    ens[i][j] = e_n[k_n]
```

```
    egjs[i][j] = e_gj[k_gj]

    print("MSE_N\t{0:.4f}".format(mse_n[k_n]))
    print("MSE_GJ\t{0:.4f}".format(mse_gj[k_gj]))

    mses[i][j] = mse_n[k_n]
    msegjs[i][j] = mse_gj[k_gj]

    print("REFF\t{0:.4f}".format(reff(mse_n[k_n], mse_gj[k_gj])))
    print("BRI\t{0:.4f}".format(bri(e_n[k_n], e_gj[k_gj], theta)))
    print("STI\t{0:.4f}".format(sti(e_n, e_gj, theta, n)))

    reffs[i][j] = reff(mse_n[k_n], mse_gj[k_gj])
    bris[i][j] = bri(e_n[k_n], e_gj[k_gj], theta)
    stis[i][j] = sti(e_n, e_gj, theta, n)
    end = time.time()
    print("Time =", end - start, "\n")

    j += 1
  print("\n\n")
  i += 1
```

```
T = 2 and n = 1000
k/n 0.1700
k_gj/n 0.8720
E_N 0.1827
E_GJ 0.2164
MSE_N 0.0011
MSE_GJ 0.0020
REFF 0.7348
BRI 1.0539
STI 7.6667
Time = 1595.3145959377289

T = 2 and n = 2000
k/n 0.1355
k_gj/n 0.8000
E_N 0.1855
E_GJ 0.2125
MSE_N 0.0007
MSE_GJ 0.0011
REFF 0.8089
BRI 1.1589
STI 8.6026
Time = 7409.737357378006




T = 3 and n = 1000
k/n 0.1680
k_gj/n 0.8750
E_N 0.1834
E_GJ 0.2161
MSE_N 0.0010
MSE_GJ 0.0021
REFF 0.7031
BRI 1.0313
STI 6.8462
Time = 8807.554994344711

T = 3 and n = 2000
k/n 0.1380
k_gj/n 0.8155
E_N 0.1854
E_GJ 0.2131
MSE_N 0.0007
MSE_GJ 0.0012
REFF 0.7729
BRI 1.1105
STI 7.9108
Time = 13815.05311369896
```

```
T = 4 and n = 1000
k/n 0.1710
k_gj/n 0.8940
E_N 0.1827
E_GJ 0.2190
MSE_N 0.0011
MSE_GJ 0.0025
REFF 0.6437
BRI 0.9122
STI 3.8462
Time = 15124.925018072128

T = 4 and n = 2000
k/n 0.1355
k_gj/n 0.8310
E_N 0.1859
E_GJ 0.2139
MSE_N 0.0007
MSE_GJ 0.0013
REFF 0.7263
BRI 1.0140
STI 7.8500
Time = 19753.08597445488




CPU times: user 5h 29min 13s, sys: 32.9 s, total: 5h 29min 46s
Wall time: 5h 29min 13s
```

In [11]:

```
np.save("kn1.npy", kns)
np.save("k_gj1.npy", kgjs)
np.save("e_n1.py", ens)
np.save("e_gj1.npy", egjs)
np.save("mse_n1.npy", mses)
np.save("mse_gj1.npy", msegjs)
np.save("reff1.npy", reffs)
np.save("bri1.npy", bris)
np.save("sti1.npy", stis)
```

In [ ]:

```

```