Generator (dependent case - s0.6)

June 14, 2023

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
############################## Importing packages␣
↪####################################

import numpy as np                      # to handle arrays and matrices
import pickle

from scipy.linalg import toeplitz       # to generate toeplitz matrix
from scipy.stats import chi2            # to have chi2 quantiles
from scipy.special import chdtri

import matplotlib.pyplot as plt         # to plot histograms ...
import pandas as pd                     # to handle and create dataframes

from scipy.linalg import toeplitz       # to generate toeplitz matrix
from scipy.stats import chi2            # to have chi2 quantiles
from scipy.special import chdtri

import time
import concurrent.futures
import random
import os

from itertools import product


######################## For printing with colors ##########################
class color:
   PURPLE = '\033[95m'
   BLACK = '\033[1;90m'
   CYAN = '\033[96m'
   DARKCYAN = '\033[36m'
   BLUE = '\033[94m'
   GREEN = '\033[1;92m'
   YELLOW = '\033[93m'
```

```
    RED = '\033[1;91m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'
    END = '\033[0m'
    BBCKGRND = '\033[0;100m'
    RBCKGRND = '\033[0;101m'


print(color.BLUE + color.BOLD + '******************** Starting the program !␣
 ↪********************' + color.END )
```

```
******************** Starting the program ! ********************
```

0.1 2.1 Toeplitz Matrix

Creating a function that gets two parameters s and q such that : - s (dependency threshold or intraparameter of dispersion) - and q (the dimension)

then returns a symetric toeplitz matrix S^2 that can be used as a covariance matrix for generating normal vectors

[3]:
```
def cov_toep(s, q):
    """A function that takes the dependency thresholds $s$
    and the dimension $q$ and returns a qxq-toeplitz matrix.
    """
    row = np.array([])
    for k in range(q):
        row = np.append(row, float(s**k))
    return toeplitz(row, row)
```

[4]:
```
print(cov_toep(0.1, 4))
print()
print(cov_toep(0.99, 4))
```

```
[[1.    0.1   0.01  0.001]
 [0.1   1.    0.1   0.01 ]
 [0.01  0.1   1.    0.1  ]
 [0.001 0.01  0.1   1.   ]]

[[1.       0.99     0.9801   0.970299]
 [0.99     1.       0.99     0.9801  ]
 [0.9801   0.99     1.       0.99    ]
 [0.970299 0.9801   0.99     1.      ]]
```

[5]:
```
# importing q_list, n_list, S2 diagonals, for different sigma (sigma means here␣
 ↪the std of underluying normal
# distribution that generates lognormal vectors), having the same alpha
q_list = list(pickle.load(open("q_list", "rb")))
```

```
n_list = list(pickle.load(open("n_list", "rb")))
print(q_list)
```

```
[50, 100, 150, 200, 250, 300, 350, 400]
```

[6]:
```
def scalar(A,B):
    """Takes two symmetric matrices A and B of sizes q
    and returns the modified frobenius scalar of A and B
    """
    return(np.trace(A.dot(np.transpose(B)))/A.shape[0])


def norm(A):
    """Takes a symmetric matrix A of sizes q
    and returns the norm of A
    This norm is associated to the modified frobenius scalar
    """
    return np.sqrt(scalar(A,A))

def alphaaa1(S_2):
    q = len(S_2)
    I_q = np.diag(np.ones(q))
    sigma2 = scalar(S_2,I_q)
    alpha2 = norm(S_2 - sigma2*I_q)**2
    return alpha2

def alphaaa2(vec):
    q = len(vec)
    I_q = np.diag(np.ones(q))
    S_2 = np.diag(vec)
    sigma2 = scalar(S_2,I_q)
    alpha2 = norm(S_2 - sigma2*I_q)**2
    return alpha2
```

[7]:
```
######################### Preparing true covariance matrices for different␣
 ↪values of q ##########################

valeur_propre_collection = [cov_toep(0.6,q) for q in q_list]
print(*valeur_propre_collection[q_list.index(50)][0][:10])
print([alphaaa1(cov_toep(0.6,q)) for q in q_list])
```

```
1.0 0.6 0.36 0.21599999999999997 0.1296 0.07775999999999998 0.04665599999999999
0.027993599999999993 0.016796159999999994 0.010077695999999997
[1.0898437500000002, 1.107421875, 1.1132812499999998, 1.1162109374999998,
1.1179687499999995, 1.119140625, 1.1199776785714288, 1.1206054687499998]
```

[8]:
```
# list(product(n_list, range(K)))[:12]
```

[9]:
```
########################## Choosing dimension␣
 ↪#######################################

# K = int(input("Number of Monte Carlo iterations is : "))
print()
print("list of q values : ", q_list, "\n")
print("list of n values : ", n_list, "\n")


######################### Generator function ################################

def generator(n, q):
    """
    Creating a function that gets into paramaters :
       the number of observations
       the dimension
       dependency threshold
     ²  covariance parameter
    and returns a matrix of n observations (n rows), where each row represents
    a q-vector normally distributed with a mean 0 and covariance matrix S²␣
 ↪defined
    by a toeplitz matrix with a threshold s
    """

    # defining eigenvalues, lognormal variables
    valeur_propre = valeur_propre_collection[q_list.index(q)]

    # defining a mean vector and a covariance matrix
    mean = np.zeros(q) ; cov = valeur_propre

    # z_intermediate = q rows and n columns we still need to transpose
    z_int = np.random.multivariate_normal(mean, cov, n).T

    # z sample matrix, having n rows and q columns
    z = np.transpose(z_int)

    # Returning the matrix of n-observations of dimension q
    return z


############################ Statistics functions ######################

def sn2(z):
    return np.cov(np.transpose(z))

def zbar(z):
    """takes a n x q sample matrix and return the vector mean of each column
```

```
    """
    return z.mean(axis=0)

def max_p(M):
    """Largest eigenvalue of a given matrix M"""
    val_p = np.linalg.eigvals(M)
    return max(val_p.real)

def min_p(M):
    """Smallest eigenvalue of a given matrix M that is not null"""
    val_p = np.linalg.eigvals(M)
    val_p = val_p[val_p>=10**-6]
    return min(val_p.real)


def product_vect(z, i):
    return np.matmul(z[i].reshape(z[i].shape[0], 1), np.transpose(z[i].
 ↪reshape(z[i].shape[0], 1)))


################################ Algebra functions␣
 ↪#####################################
def scalar(A,B):
    """Takes two symmetric matrices A and B of sizes q
    and returns the modified frobenius scalar of A and B
    """
    return(np.trace(A.dot(np.transpose(B)))/A.shape[0])


def norm(A):
    """Takes a symmetric matrix A of sizes q
    and returns the norm of A
    This norm is associated to the modified frobenius scalar
    """
    return np.sqrt(scalar(A,A))
```

```
list of q values :  [50, 100, 150, 200, 250, 300, 350, 400]

list of n values :  [50, 75, 100, 125, 150, 175, 200]
```

1 Time comparision

[10]:
```
def monte_carlo(k):
    # random.seed(k)
    np.random.seed(int(os.getpid() * time.time()) % 123456789)
    z = generator(n,q)
    sn_2 = sn2(z)
    # beta = (norm(sn_2 - s_2))**2
    # calculate empirical mean
    z_bar = zbar(z)
    #empirical covariance matrix
    sn_2 = sn2(z)

    # Calculating empirical eigenvalues and eigenvectors (of Sn²)
    emp_val_p, emp_vec_p = np.linalg.eigh(sn_2)

    # Calculating true (theoretical) eigenvalues and eigenvectors (of S²)
    # val_p, vec_p = np.linalg.eigh(s_2)

    # Identity matrix of size q
    I_q = np.diag(np.ones(q))

    # sigma_n ( ² )
    sigma_n = scalar(sn_2, I_q)

    # delta_n ( ²)
    delta_n = norm(sn_2 - sigma_n*I_q)**2

    # intermediate beta_n
    beta_bar_n = (1/n**2)*0
    for i in range(n):
        beta_bar_n += (1/n**2)*norm(product_vect(z, i) - sn_2)**2

    # beta_n ( ²)
    beta_n = min(beta_bar_n, delta_n)

    # alpha_n ( ² )
    alpha_n = delta_n - beta_n

    # rho_n ( *² )
    rho_n = (beta_n/alpha_n)*sigma_n

    rho_1_n = (beta_n/delta_n)*sigma_n

    rho_2_n = alpha_n/delta_n

    Sigma_n_hat_ast = sn_2 + rho_n*I_q
```

```
    Sigma_n_hat = rho_1_n*I_q + rho_2_n*sn_2

    self_norm_sum = n*z_bar.dot(np.linalg.inv(Sigma_n_hat).dot(np.
 ↪transpose(z_bar)))
    self_norm_sum_ast = n*z_bar.dot(np.linalg.inv(Sigma_n_hat_ast).dot(np.
 ↪transpose(z_bar)))
    return np.array([self_norm_sum, self_norm_sum_ast, beta_n, sigma_n,␣
 ↪alpha_n, rho_n, max_p(sn_2),
                     min_p(sn_2)], dtype=np.int)



t1 = time.perf_counter()
dico = {}
for n in n_list[:3]:
    for q in q_list[:3]:
        if n <= q :
            dico[(n,q)] = np.stack(list(map(monte_carlo, range(10))))

print(dico[list(dico.keys())[0]])

t2 = time.perf_counter()

print(f'Finished in {t2-t1} seconds')
```

```
[[49 26  0  0  1  0  6  0]
 [62 32  1  1  1  0  5  0]
 [74 41  1  1  1  0  7  0]
 [43 23  0  0  1  0  6  0]
 [62 31  1  1  1  0  5  0]
 [58 30  1  1  1  0  6  0]
 [40 21  1  1  1  0  6  0]
 [48 26  0  0  1  0  7  0]
 [48 29  1  1  1  0  8  0]
 [43 22  0  0  0  0  5  0]]
Finished in 2.470908208997571 seconds
```

[11]:
```
len(dico.keys())
```

[11]: 7

[12]:
```
t1 = time.perf_counter()
data = {}
for q in q_list[:3]:
    for n in n_list[:3]:
        if n <= q :
            with concurrent.futures.ProcessPoolExecutor() as executor:
```

```
                f1 = np.stack(list(executor.map(monte_carlo, range(10))))
                #f2 = f1.result()
                data[(n,q)] = f1
                # if q/n%1==0 : print(str(i)+","+str(j)+")",
                                #color.BLUE + color.BOLD + f"for n = %d \t␣
 ↪and q = %d"%(n,q) + color.END,"\n",
                                #f1, "\n\n")

print(data[list(data.keys())[0]])

t2 = time.perf_counter()

print(f'Finished in {t2-t1} seconds')
```

```
[[48 26  1  1  1  0  6  0]
 [37 18  0  0  0  1  5  0]
 [74 37  1  0  1  0  6  0]
 [45 22  0  0  0  1  5  0]
 [63 31  0  0  0  0  6  0]
 [61 33  1  1  1  0  6  0]
 [62 34  1  1  1  0  7  0]
 [43 22  0  0  0  0  6  0]
 [67 35  1  1  1  0  7  0]
 [45 22  1  0  1  0  6  0]]
Finished in 3.6140988240003935 seconds
```

[13]:
```
data[list(data.keys())[1]]
```

[13]:
```
array([[102,  38,   2,   0,   1,   1,   9,   0],
       [ 88,  30,   1,   0,   1,   1,   7,   0],
       [ 74,  29,   2,   1,   1,   1,   8,   0],
       [101,  37,   1,   0,   1,   1,   8,   0],
       [ 99,  36,   2,   1,   1,   1,   8,   0],
       [115,  41,   2,   1,   1,   1,   8,   0],
       [104,  43,   2,   1,   1,   1,   9,   0],
       [ 90,  35,   2,   1,   1,   1,   8,   0],
       [117,  39,   2,   1,   1,   1,   7,   0],
       [ 96,  37,   1,   0,   1,   1,   8,   0]])
```

2 Generating step

[14]:
```
K = int(input("Number of Monte Carlo iterations is : "))
```

```
Number of Monte Carlo iterations is : 999
```

[15]:
```
t_init = time.perf_counter()
dico = {}
for n in n_list:
    for q in q_list:
        t1 = time.perf_counter()
        if n <= q :
            dico[(n,q)] = np.stack(list(map(monte_carlo, range(10))))
            t2 = time.perf_counter()
            if q==n or q==2*n :
                print(f"q = {q}  and  n = {n} --> ",f'Finished in {round(t2-t1,␣
 ↪4)} seconds')

print(dico[list(dico.keys())[0]][:10])

t_fin = time.perf_counter()
print()
print(f'All loops finished in {round(t_fin-t_init, 3)} seconds')
```

```
q = 50  and  n = 50 -->  Finished in 0.077 seconds
q = 100  and  n = 50 -->  Finished in 0.1482 seconds
q = 150  and  n = 75 -->  Finished in 0.5359 seconds
q = 100  and  n = 100 -->  Finished in 0.2312 seconds
q = 200  and  n = 100 -->  Finished in 1.1857 seconds
q = 250  and  n = 125 -->  Finished in 1.9085 seconds
q = 150  and  n = 150 -->  Finished in 0.6856 seconds
q = 300  and  n = 150 -->  Finished in 3.45 seconds
q = 350  and  n = 175 -->  Finished in 6.3775 seconds
q = 200  and  n = 200 -->  Finished in 2.2651 seconds
q = 400  and  n = 200 -->  Finished in 10.6307 seconds
[[63 35  0  0  1  0  6  0]
 [63 33  0  0  1  0  6  0]
 [34 19  1  1  1  0  6  0]
 [45 22  1  1  1  1  5  0]
 [59 32  1  1  1  0  6  0]
 [32 18  1  1  1  0  7  0]
 [43 24  0  0  1  0  6  0]
 [29 16  1  1  1  0  7  0]
 [45 23  1  1  1  1  6  0]
 [45 23  1  1  1  0  6  0]]

All loops finished in 133.603 seconds
```

[16]:
```
def monte_carlo(k):
    # random.seed(k)
    np.random.seed(int(os.getpid() * time.time()) % 123456789)
    z = generator(n,q)
    sn_2 = sn2(z)
```

```
    s_2 = valeur_propre_collection[q_list.index(q)]
    beta = (norm(sn_2 - s_2))**2
    # calculate empirical mean
    z_bar = zbar(z)
    #empirical covariance matrix
    sn_2 = sn2(z)

    # Calculating empirical eigenvalues and eigenvectors (of Sn²)
    emp_val_p, emp_vec_p = np.linalg.eigh(sn_2)

    # Calculating true (theoretical) eigenvalues and eigenvectors (of S²)
    # val_p, vec_p = np.linalg.eigh(s_2)

    # Identity matrix of size q
    I_q = np.diag(np.ones(q))

    # sigma_n ( ² )
    sigma_n = scalar(sn_2, I_q)

    # delta_n ( ²)
    delta_n = norm(sn_2 - sigma_n*I_q)**2

    # intermediate beta_n
    beta_bar_n = (1/n**2)*0
    for i in range(n):
        beta_bar_n += (1/n**2)*norm(product_vect(z, i) - sn_2)**2

    # beta_n ( ²)
    beta_n = min(beta_bar_n, delta_n)

    # alpha_n ( ² )
    alpha_n = delta_n - beta_n

    # rho_n ( *² )
    rho_n = (beta_n/alpha_n)*sigma_n

    rho_1_n = (beta_n/delta_n)*sigma_n

    rho_2_n = alpha_n/delta_n

    Sigma_n_hat_ast = sn_2 + rho_n*I_q
    Sigma_n_hat = rho_1_n*I_q + rho_2_n*sn_2

    self_norm_sum = n*z_bar.dot(np.linalg.inv(Sigma_n_hat).dot(np.
↪transpose(z_bar)))
    self_norm_sum_ast = n*z_bar.dot(np.linalg.inv(Sigma_n_hat_ast).dot(np.
↪transpose(z_bar)))
```

```
    return np.array([self_norm_sum, self_norm_sum_ast, beta_n, sigma_n,␣
 ↪alpha_n, rho_n, max_p(sn_2),
                     min_p(sn_2), beta])
```

[17]:
```
print(color.BLUE + color.BOLD + '******************** Starting the extraction !␣
 ↪********************' + color.END )

K = int(input("Number of Monte Carlo iterations is : "))
t_init = time.perf_counter()
dico = {}
for n in n_list:
    for q in q_list:
        t1 = time.perf_counter()
        if n <= q :
            dico[(n,q)] = np.stack(list(map(monte_carlo, range(K))))
            t2 = time.perf_counter()
            if q==n or q==2*n :
                print(f"q = {q}  and  n = {n} --> ",f'Finished in {round(t2-t1,␣
 ↪4)} seconds')

print(dico[list(dico.keys())[0]][:10])

t_fin = time.perf_counter()
print()
print(f'All loops finished in {round(t_fin-t_init, 3)} seconds')
```

```
******************** Starting the extraction ! ********************
Number of Monte Carlo iterations is : 999
q = 50  and  n = 50 -->  Finished in 6.9363 seconds
q = 100  and  n = 50 -->  Finished in 20.505 seconds
q = 150  and  n = 75 -->  Finished in 53.5549 seconds
q = 100  and  n = 100 -->  Finished in 22.8282 seconds
q = 200  and  n = 100 -->  Finished in 108.3973 seconds
q = 250  and  n = 125 -->  Finished in 212.258 seconds
q = 150  and  n = 150 -->  Finished in 74.594 seconds
q = 300  and  n = 150 -->  Finished in 387.8014 seconds
q = 350  and  n = 175 -->  Finished in 664.2058 seconds
q = 200  and  n = 200 -->  Finished in 90.4089 seconds
q = 400  and  n = 200 -->  Finished in 539.5729 seconds
[[4.36163387e+01 2.29947615e+01 1.01058375e+00 9.94405566e-01
  1.12688434e+00 8.91777509e-01 6.76539144e+00 2.92378122e-04
  1.05684057e+00]
 [3.90776396e+01 2.43712288e+01 1.19192070e+00 1.07589405e+00
  1.97523192e+00 6.49230289e-01 9.01775325e+00 9.11361451e-04
  1.53953565e+00]
 [3.77803838e+01 1.97392666e+01 8.70796720e-01 9.31511522e-01
  9.52761872e-01 8.51374517e-01 5.92505744e+00 1.12011788e-03
```

```
 9.65588366e-01]
[4.76576721e+01 2.49843607e+01 9.04682944e-01 9.46184550e-01
 9.96895627e-01 8.58662634e-01 6.09685173e+00 1.85657343e-04
 9.67960342e-01]
[5.28018508e+01 3.10856052e+01 1.07685739e+00 1.03343395e+00
 1.54146183e+00 7.21951696e-01 7.40632959e+00 4.79023767e-04
 1.32548358e+00]
[6.35447898e+01 3.37727223e+01 1.13486920e+00 1.02885141e+00
 1.28736851e+00 9.06975560e-01 7.12971728e+00 4.34679267e-04
 1.26854087e+00]
[4.08157855e+01 2.27036168e+01 1.09457021e+00 1.03256946e+00
 1.37204456e+00 8.23748584e-01 7.81942168e+00 9.61351707e-04
 1.29813534e+00]
[6.00682046e+01 3.04724475e+01 9.66956786e-01 9.69569352e-01
 9.95600139e-01 9.41674903e-01 5.70593205e+00 9.60919817e-04
 9.34965244e-01]
[3.09277276e+01 1.64594267e+01 9.82699628e-01 9.66607448e-01
 1.11793863e+00 8.49675248e-01 6.23098512e+00 9.45408080e-05
 1.01211637e+00]
[5.44032587e+01 2.83334117e+01 1.00524975e+00 9.93486050e-01
 1.09253249e+00 9.14116156e-01 5.80753461e+00 1.61521465e-03
 1.15697731e+00]]

All loops finished in 12750.635 seconds
```

[18]:
```
pickle.dump(dico, open("data_d_s0.6", "wb"))
```

[19]:
```
teest = pickle.load(open("data_d_s0.6", "rb"))
len(teest[list(teest.keys())[0]])
```

[19]: 999

[20]:
```
teest[list(teest.keys())[0]][:100,0]
```

[20]:
```
array([43.61633868, 39.07763963, 37.78038384, 47.65767212, 52.80185077,
       63.54478979, 40.81578545, 60.06820461, 30.9277276 , 54.40325868,
       53.92542133, 40.7806492 , 61.91762462, 45.53339006, 46.01145098,
       57.9904726 , 45.8014757 , 51.65765489, 51.99714112, 42.08810106,
       48.90455907, 48.35879879, 66.13698697, 34.69292988, 36.92586344,
       67.40620951, 40.74739076, 61.96337079, 37.28401014, 68.90944204,
       63.65948672, 44.47166404, 36.93888047, 48.2314378 , 51.49386381,
       51.70780367, 48.0522432 , 57.07099258, 54.70883515, 43.95208731,
       41.61446553, 43.5558393 , 36.47337442, 64.30745563, 38.39860469,
       41.46017687, 36.47325054, 52.44642022, 51.68395228, 34.84017003,
       45.35290214, 30.11200769, 50.83476017, 56.97882203, 83.37324586,
       35.64845542, 55.78649682, 44.74674965, 67.7756861 , 62.00105905,
       49.7587842 , 57.20680248, 33.8051873 , 77.41768783, 44.98385818,
```

```
56.61212951, 56.45081119, 46.01350391, 33.94984957, 33.58601856,
52.43267599, 59.94202846, 42.56985293, 43.92285685, 42.10736153,
40.94954339, 31.93847436, 49.96644167, 68.7511026 , 42.08953012,
39.00069594, 65.85563002, 20.24246909, 55.51626525, 48.55358617,
65.19782297, 49.9842238 , 23.73902728, 61.15717356, 44.24313986,
38.03946275, 41.01256628, 55.3776742 , 74.00487526, 66.20808614,
26.72228924, 79.64053046, 39.51449616, 80.02408484, 59.61516822])
```

[ ]: