Generator - independent (d0.99)

June 14, 2023

[2]:
```
import warnings
warnings.filterwarnings('ignore')
```

[3]:
```
############################## 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

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 ! ********************
```

[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 = pickle.load(open("q_list", "rb"))
n_list = pickle.load(open("n_list", "rb"))

vp_collection_d099 = pickle.load(open("vp_collection_d099", "rb"))
```

[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 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]:
```
print("q_list: ", q_list)
print("n_list: ",n_list)
```

```
q_list:  [ 50 100 150 200 250 300 350 400]
n_list:  [ 50  75 100 125 150 175 200]
```

[8]:
```
list(product([1,2],[3,5,6]))
```

[8]:
```
[(1, 3), (1, 5), (1, 6), (2, 3), (2, 5), (2, 6)]
```

[9]:
```
for i in range(5):
    if i == 0 :
        array = np.array(i*np.ones(5, dtype=np.int))
    else :
        array = np.vstack((array,np.array(i*np.ones(5, dtype=np.int))))

array
```

[9]:
```
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3],
       [4, 4, 4, 4, 4]])
```

[10]:
```
n_list = list(n_list) ; q_list = list(q_list)
```

[11]:
```
######################## Generating/reading eigenvalues and saving them in a vp␣
 ↪file ##########################

valeur_propre_collection = vp_collection_d099
valeur_propre_collection[q_list.index(50)]
```

[11]:
```
array([ 1.06966779,  0.546609  ,  0.30378641,  0.15724564,  0.2847767 ,
        0.39310208,  0.5543666 ,  2.00091877,  1.5075462 ,  0.54438166,
        0.61244331,  1.9512356 ,  6.43578564,  0.88522971,  1.1319392 ,
        1.54630815,  0.53659325,  1.7144116 ,  0.91957655,  2.92735842,
        2.73877276,  1.30040558,  0.33558894,  0.29997956,  0.20296237,
        0.4946676 ,  3.37935214,  0.38785038,  0.28324284,  4.18337946,
        0.33396126,  2.16623162,  1.01510435,  1.13914008,  0.06280495,
        0.42595863,  1.22650489,  0.45700073,  2.1736622 ,  2.84315448,
        0.97452588,  0.67214135,  0.36562842,  4.20642738,  0.1345158 ,
        3.42401627,  0.2376995 , 43.02066943,  2.6754375 ,  0.1397993 ])
```

[16]:
```
print([alphaaa2(i) for i in vp_collection_d099])
```

```
[35.81322639270601, 55.5057610930806, 67.16015515618975, 74.03142454338604,
78.71857289577437, 82.06276498069886, 84.38759586673962, 86.29469286163241]
```

[ ]:
```

```

[10]:
```
# list(product(n_list, range(K)))[:12]
```

[19]:
```
########################## 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 = np.diag(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

[20]:
```
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], 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')
```

```
[[ 39  33   5   2  35   0]
 [ 42  34   4   2  21   0]
 [121 109   5   2  51   0]
 [ 59  43   3   1   9   0]
 [ 75  65   5   2  36   0]
 [ 50  41   4   1  20   0]
 [ 55  48   5   2  39   0]
 [ 50  39   4   1  15   0]
 [103  90   4   2  34   0]
 [ 76  66   5   2  38   0]]
Finished in 0.7875641440000436 seconds
```

[21]:
```
len(dico.keys())
```

[21]: 7

[22]:
```
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')
```

```
[[83 70  4  1 23  0]
 [58 48  5  2 29  0]
 [95 81  5  2 31  0]
 [61 50  5  2 29  0]
 [64 53  4  1 20  0]
 [68 58  5  2 33  0]
 [68 58  4  2 28  0]
 [72 59  3  1 16  0]
 [74 59  4  2 18  0]
 [58 49  5  2 34  0]]
Finished in 1.091563658999803 seconds
```

[23]:
```
data[list(data.keys())[1]]
```

[23]:
```
array([[ 95,  67,  30,   3,  72,   1],
       [111,  76,  27,   3,  61,   1],
       [ 85,  54,  26,   3,  46,   1],
       [ 76,  50,  26,   3,  50,   1],
       [140,  89,  26,   3,  45,   2],
       [ 98,  69,  28,   3,  66,   1],
       [ 87,  57,  23,   3,  46,   1],
       [148, 103,  28,   3,  67,   1],
       [127,  83,  26,   3,  50,   1],
       [144, 109,  31,   3,  98,   1]])
```

2 Generating step

[24]:
```
K = int(input("Number of Monte Carlo iterations is : "))
```

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

[25]:
```
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 0.0477 seconds
q = 100  and  n = 50 -->  Finished in 0.0887 seconds
q = 150  and  n = 75 -->  Finished in 0.1633 seconds
q = 100  and  n = 100 -->  Finished in 0.0978 seconds
q = 200  and  n = 100 -->  Finished in 0.3645 seconds
q = 250  and  n = 125 -->  Finished in 0.6751 seconds
q = 150  and  n = 150 -->  Finished in 0.2705 seconds
q = 300  and  n = 150 -->  Finished in 1.89 seconds
q = 350  and  n = 175 -->  Finished in 2.8796 seconds
q = 200  and  n = 200 -->  Finished in 0.7012 seconds
q = 400  and  n = 200 -->  Finished in 5.1091 seconds
[[ 55  46   4   2  23   0]
 [ 58  53   5   2  53   0]
 [ 57  53   8   2 108   0]
 [ 54  49   6   2  59   0]
 [ 70  60   5   2  33   0]
 [ 87  75   5   2  32   0]
 [ 79  69   5   2  36   0]
 [ 50  44   4   2  32   0]
 [ 78  67   5   2  32   0]
 [ 60  54   7   2  62   0]]

All loops finished in 55.378 seconds
```

[26]:
```
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 = np.diag(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])
```

[27]:
```
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 7.0502 seconds
q = 100  and  n = 50 -->  Finished in 15.5203 seconds
q = 150  and  n = 75 -->  Finished in 51.3596 seconds
q = 100  and  n = 100 -->  Finished in 27.4664 seconds
q = 200  and  n = 100 -->  Finished in 133.0195 seconds
q = 250  and  n = 125 -->  Finished in 224.4931 seconds
q = 150  and  n = 150 -->  Finished in 76.7582 seconds
q = 300  and  n = 150 -->  Finished in 406.2834 seconds
q = 350  and  n = 175 -->  Finished in 634.1729 seconds
q = 200  and  n = 200 -->  Finished in 186.1759 seconds
q = 400  and  n = 200 -->  Finished in 1056.4459 seconds
[[7.31952278e+01 6.53952608e+01 6.34760333e+00 2.39993680e+00
  5.32185807e+01 2.86250528e-01 5.47418072e+01 2.35624133e-04
  6.51793418e+00]
 [6.28752292e+01 5.38609811e+01 5.04958751e+00 2.11452755e+00
  3.01717608e+01 3.53890247e-01 4.16047591e+01 2.39199464e-06
  3.84272763e+00]
 [5.43374320e+01 4.82045771e+01 5.80392654e+00 2.21649116e+00
  4.56191820e+01 2.81994356e-01 5.11015260e+01 2.21992171e-04
  4.90899521e+00]
 [5.08257705e+01 4.51972363e+01 6.98306312e+00 2.43557494e+00
  5.60741288e+01 3.03308744e-01 5.62206077e+01 1.02465853e-04
  8.62140774e+00]
 [6.68024017e+01 5.57334966e+01 3.98261358e+00 1.90854980e+00
  2.00530203e+01 3.79045960e-01 3.38113967e+01 1.10521688e-04
  6.36897484e+00]
 [5.30859115e+01 4.55998116e+01 4.93886739e+00 2.14851897e+00
```

```
 3.00839458e+01 3.52721360e-01 4.13158015e+01 9.63342046e-04
 4.23435831e+00]
[5.09937035e+01 4.50455238e+01 4.67812639e+00 2.19684509e+00
 3.54274189e+01 2.90089408e-01 4.44596507e+01 9.10919543e-04
 2.89772409e+00]
[4.07925553e+01 3.53907975e+01 6.31959260e+00 2.28720773e+00
 4.14041934e+01 3.49100414e-01 4.84571800e+01 2.11441495e-04
 4.07873093e+00]
[4.52789257e+01 3.53402610e+01 4.52225843e+00 1.88506751e+00
 1.60804089e+01 5.30133438e-01 3.11038306e+01 3.00213246e-04
 6.08733295e+00]
[8.98087758e+01 7.77395171e+01 5.26911456e+00 2.14634960e+00
 3.39389875e+01 3.33226260e-01 4.40415948e+01 9.51370488e-04
 5.04926442e+00]]

All loops finished in 13597.529 seconds
```

[28]:
```
pickle.dump(dico, open("data_vp_collection_d099", "wb"))
```

[29]:
```
aaa = pickle.load(open("data_vp_collection_d099", "rb"))
len(aaa[list(aaa.keys())[0]])
```

[29]:
```
999
```

[30]:
```
aaa[list(aaa.keys())[0]][0]
```

[30]:
```
array([7.31952278e+01, 6.53952608e+01, 6.34760333e+00, 2.39993680e+00,
       5.32185807e+01, 2.86250528e-01, 5.47418072e+01, 2.35624133e-04,
       6.51793418e+00])
```

[ ]:

[ ]: