Generator - independent - (d0.6 and alpha1.1)

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

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

[3]:
```
# 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"))

## for sigma = 1
# alpha = 1.1
vp_collection_d06 = pickle.load(open("vp_collection_d06", "rb"))
```

[4]:
```
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
```

[5]:
```
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]
```

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

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

[7]:
```
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
```

[7]:
```
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]])
```

[8]:
```
n_list = list(n_list) ; q_list = list(q_list)
```

[9]:
```
######################### Generating/reading eigenvalues and saving them in a vp␣
 ↪file ############################

valeur_propre_collection = vp_collection_d06
valeur_propre_collection[q_list.index(50)]
```

[9]:
```
array([0.97510923, 1.67328828, 0.89351659, 3.37591872, 1.72503673,
       1.36081186, 1.57742554, 0.66677266, 1.09063994, 0.8935847 ,
       0.54496284, 1.72118292, 1.0831433 , 0.24704241, 0.92338678,
       1.03489715, 0.49174069, 2.85322697, 1.74896556, 1.40564645,
       0.87047796, 0.565678  , 1.57107902, 3.06130656, 0.95970954,
       2.915226  , 0.86279639, 1.07099968, 0.55649651, 1.13476559,
       0.6212695 , 0.81358513, 0.60064821, 2.26296607, 0.85941744,
       1.07721453, 3.77528784, 0.83417751, 1.941779  , 0.51770486,
       0.69803273, 0.66628242, 1.34353837, 0.82043131, 1.61051701,
       6.29794491, 0.74396439, 1.09975627, 1.09331817, 0.55082575])
```

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

[11]:
```
########################## 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 = np.diag(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

[12]:
```
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')
```

```
[[39 12  1  1  0  2  7  0]
 [50 17  1  1  0  2  7  0]
 [39 14  1  1  1  2  7  0]
 [65 24  2  1  1  2  9  0]
 [67 27  1  1  1  1  8  0]
 [42 15  1  1  1  2  7  0]
 [48 17  1  1  1  2  8  0]
 [56 22  1  1  1  1  8  0]
 [66 23  1  1  0  2  7  0]
 [70 22  1  1  0  2  7  0]]
Finished in 1.2573942019998867 seconds
```

[13]:
```
len(dico.keys())
```

[13]: 7

[14]:
```
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')
```

```
[[50 18  1  1  1  2  7  0]
 [64 20  1  1  0  2  7  0]
 [35 14  1  1  1  1  8  0]
 [51 20  1  1  1  2  7  0]
 [39 16  1  1  1  2  8  0]
 [69 19  1  1  0  3  6  0]
 [30 14  1  1  1  1  9  0]
 [65 25  1  1  1  2  7  0]
 [40 17  2  1  1  1  9  0]
 [48 16  1  1  0  2  6  0]]
Finished in 1.5099213639996378 seconds
```

[15]:
```
data[list(data.keys())[1]]
```

[15]:
```
array([[114,  30,   3,   1,   1,   3,  10,   0],
       [ 82,  23,   2,   1,   1,   2,   9,   0],
       [118,  42,   3,   1,   1,   2,  11,   0],
       [119,  42,   3,   1,   1,   2,  12,   0],
       [ 97,  28,   3,   1,   1,   3,  11,   0],
       [ 70,  17,   2,   1,   0,   3,   9,   0],
       [110,  31,   2,   1,   1,   3,  10,   0],
       [ 97,  28,   2,   1,   1,   2,  10,   0],
       [ 82,  25,   3,   1,   1,   2,  11,   0],
       [ 84,  24,   2,   1,   1,   2,  10,   0]])
```

2 Generating step

[17]:
```
K = int(input("Number of Monte Carlo iterations is : "))
```

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

[18]:
```
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 0.0962 seconds
q = 100  and  n = 50 -->  Finished in 0.094 seconds
q = 150  and  n = 75 -->  Finished in 0.3375 seconds
q = 100  and  n = 100 -->  Finished in 0.1318 seconds
q = 200  and  n = 100 -->  Finished in 0.6727 seconds
q = 250  and  n = 125 -->  Finished in 1.0055 seconds
q = 150  and  n = 150 -->  Finished in 0.4139 seconds
q = 300  and  n = 150 -->  Finished in 2.1739 seconds
q = 350  and  n = 175 -->  Finished in 3.3477 seconds
q = 200  and  n = 200 -->  Finished in 0.9214 seconds
q = 400  and  n = 200 -->  Finished in 5.6364 seconds
[[50 16  2  1  0  3  7  0]
 [54 22  1  1  1  2  9  0]
 [38 15  1  1  1  2  7  0]
 [49 14  1  1  0  3  6  0]
 [37 15  2  1  1  2  8  0]
 [47 17  1  1  1  2  7  0]
 [37 14  1  1  1  2  8  0]
 [46 19  2  1  1  1  8  0]
 [48 16  1  1  1  2  7  0]
 [58 19  1  1  0  2  6  0]]

All loops finished in 73.732 seconds
```

[19]:
```
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])
```

[20]:
```
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 3.8543 seconds
q = 100  and  n = 50 -->  Finished in 8.282 seconds
q = 150  and  n = 75 -->  Finished in 29.8846 seconds
q = 100  and  n = 100 -->  Finished in 15.1116 seconds
q = 200  and  n = 100 -->  Finished in 64.8112 seconds
q = 250  and  n = 125 -->  Finished in 117.0531 seconds
q = 150  and  n = 150 -->  Finished in 51.9029 seconds
q = 300  and  n = 150 -->  Finished in 210.2379 seconds
q = 350  and  n = 175 -->  Finished in 336.2786 seconds
q = 200  and  n = 200 -->  Finished in 92.2538 seconds
q = 400  and  n = 200 -->  Finished in 563.8814 seconds
[[3.79724186e+01 1.54076924e+01 1.96657676e+00 1.39294368e+00
  1.34282197e+00 2.03998054e+00 9.68578962e+00 1.96238557e-04
  2.02677925e+00]
 [4.01586322e+01 1.61069034e+01 1.85730244e+00 1.36558750e+00
  1.24379379e+00 2.03917162e+00 9.14861789e+00 1.64805968e-03
  1.87008826e+00]
 [6.24771007e+01 2.44897918e+01 1.79343398e+00 1.33718710e+00
  1.15619732e+00 2.07417604e+00 8.34162664e+00 1.54302988e-03
  1.85776746e+00]
 [7.31951569e+01 2.42249092e+01 1.79539872e+00 1.32471474e+00
  8.88159097e-01 2.67788863e+00 6.89104085e+00 1.61708003e-05
  1.92070484e+00]
 [6.75129965e+01 2.20105737e+01 1.89176658e+00 1.35422067e+00
  9.15091220e-01 2.79957817e+00 6.99159832e+00 2.27571735e-03
  1.75878214e+00]
 [3.16100850e+01 1.25028408e+01 1.71908468e+00 1.33091459e+00
  1.12488446e+00 2.03394657e+00 8.08102828e+00 7.43257043e-04
  1.83017890e+00]
 [5.13527519e+01 2.47584965e+01 1.97168430e+00 1.40455795e+00
  1.83558208e+00 1.50870118e+00 1.05289980e+01 9.07745815e-04
  2.37189320e+00]
```

```
 [5.94068011e+01 2.28044192e+01 1.95360068e+00 1.37757938e+00
  1.21715381e+00 2.21109279e+00 9.57135650e+00 3.08854859e-04
  1.99628837e+00]
 [6.34808354e+01 2.54475506e+01 2.04383894e+00 1.42001908e+00
  1.36750468e+00 2.12232567e+00 8.78098337e+00 3.62174366e-04
  1.89071635e+00]
 [4.60418489e+01 1.46767979e+01 1.87652694e+00 1.35431543e+00
  8.78092198e-01 2.89423980e+00 6.92124743e+00 9.53817809e-04
  1.89867253e+00]]

All loops finished in 7826.661 seconds
```

[24]:
```
pickle.dump(dico, open("data_vp_collection_d06", "wb"))
```

[25]:
```
aaa = pickle.load(open("data_vp_collection_d06", "rb"))
len(aaa[list(aaa.keys())[0]])
```

[25]:
```
999
```

[26]:
```
aaa[list(aaa.keys())[0]][0]
```

[26]:
```
array([3.79724186e+01, 1.54076924e+01, 1.96657676e+00, 1.39294368e+00,
       1.34282197e+00, 2.03998054e+00, 9.68578962e+00, 1.96238557e-04,
       2.02677925e+00])
```

[ ]:

[ ]: