Eindhoven University of Technology
Department of Mathematics & Computer Science

Estimating free spectral norms of random graphs

Thomas van Kaathoven Waseem Khan David Meijll

Contents

1 Introduction

An important field within probability theory regards the study of random variables and random graphs. In particular, probability theorists are interested in explaining the behaviour of random variables from their expectation or variance. To do this, they aim to find inequalities that make calculations easier. An example of such an inequality is the Law of Large Numbers, which states that the sum of a number of independent, identically distributed random variables tends to the expectation.
In recent times, such an inequality was found that relates to the spectral norm of a sum of dependent matrices. Using free probability theory, a so-called free model can be calculated that yields a strict bound for the spectral norm.
In this report, we will explore methods to estimate this free model using simulation techniques. To do this, we will use a certain type of random graph, the Erdős–Rényi graph, and build a number of experiments to estimate the free model quantity.

2 Mathematical description

Although we use simulation to get our estimations, we first lay down a mathematical base that introduces all of the relevant concepts.

Firstly, the spectral norm is given by

$$||S_n||_2 := \max_{||v||_2 \neq 0} \frac{||S_n v||_2}{||v||_2} = \lambda_{\max}(S_n),$$

for

$$S_n := \sum_{i=0}^{n} X_i,$$

which is the sum of n dependent matrices. The inequality that was found relating to the spectral norm now implies that

$$\mathbb{E}\left[||S_n||_2\right] \leq ||S_{n,\text{free}}|| + \epsilon_d,$$

for some small ϵ_d. Note that $||S_{n,\text{free}}||$ is the free model we wish to estimate. To help with this, we are given an inequality that describes the free model, based on positive definite matrices. This inequality states that

$$||S_{n,\text{free}}|| \leq \max_{\eta \in \{-1,+1\}} \min\{\lambda_{\max}\left(Z_1^{-1} + \eta\mathbb{E}[S_n] + \mathbb{E}[(S_n - \mathbb{E}[S_n])^T Z_1 (S_n - \mathbb{E}[S_n])]\right), \ldots, \quad (1)$$

$$\lambda_{\max}\left(Z_m^{-1} + \eta\mathbb{E}[S_n] + \mathbb{E}[(S_n - \mathbb{E}[S_n])^T Z_m (S_n - \mathbb{E}[S_n])]\right)\}. \quad (2)$$

In order to estimate the quantity by simulation, we now need methods to generate S_n, Z, and estimate $\mathbb{E}[S_n]$. As mentioned earlier, we will use Erdős–Rényi graphs for this. An undirected Erdős–Rényi graph $G(d, p(d))$, with $d \in \mathbb{N}_+$ and $p : \mathbb{N}_+ \to \{0, 1\}$, is a graph with adjacency matrix $A \in \{0, 1\}^{d \times d}$ such that

$$A_{i,j} = \begin{cases} A_{j,i} = \text{Bernoulli}(p(d)) & \text{if } i \neq j, \\ 0 & \text{if } i = j, \end{cases}$$

for $i = \{1, \ldots, d\}$. The adjacency matrix A can then be translated to the graph G. Note that, in this report, we will often use q in the place of $p(d)$.

3 Implementation

In the following segment, we will discuss the different experiments we performed. We do this by explaining the code that we implemented, either using words or using pseudo-code. Our results will then be discussed in section 4. The experiments consist of two groups: the first group requires us to find a general solution or method to solve, and the second group takes specific parameters as input. Hence, most of the implementations and algorithms will already be set up in the first group of experiments, while the second group will contain the most interesting results.

First, we were asked to visualize the eigenvalues of an Erdős–Rényi adjacency matrix in a histogram. For this, we need a function that generates such a matrix, calculates its eigenvalues, and prints them in a histogram. This function takes dimension d and probability p as an input. Based on p, we first generate a d-dimensional matrix A such that $a_{ij} = 1$ with probability p, if $i > j$. This is done by sampling from a Bernoulli distribution with parameter p. We then ensure that the diagonal contains only zeroes, and add the matrix to its transpose, to make it symmetric.
Now that we have generated an adjacency matrix, we use `numpy` to find all eigenvalues of the matrix, and `pyplot` to generate a histogram.

Next, we estimate the upper bound for $||S_{n,\text{free}}||$ described in Equation 1 using two different methods. In order to do this, we first need to be able to generate S_n and positive definite matrices Z, and some useful estimates need to be made. In the following algorithms, we use $O_{d\times d}$ to denote a square 0-matrix of dimension d, and `erdosRenyiGen` to denote the function that generates adjacency matrices from before.

Algorithm 1 mu_nEstimator: Generate S_n and use it as an estimate μ_n for $\mathbb{E}[S_n]$

Input: d, n, p.
Output: $S_n = \sum_{i=1}^{n} X_i = \mu_n$.

$X \leftarrow n \cdot O_{d\times d}$ (a collection of n $d \times d$-matrices)
for $i \in \{1, \dots n\}$ **do**
  $X_i \leftarrow \text{erdosRenyiGen}(d, p)$
$S_n \leftarrow \sum_{i=1}^{n} X_i$
return S_n

Algorithm 2 zi6Gen: Generate positive definite Z-matrix

Input: d, Σ.
Output: positive definite Z.

$G \leftarrow O_{d\times d}$
for $i \in \{1, \dots d\}$ **do**
  $G_i \leftarrow \text{MultivariateNormal}_d(0, \Sigma)$
$Z \leftarrow O_{d\times d}$
for $i \in \{1, \dots d\}$ **do**
  $Z \leftarrow Z + G_i \cdot G_i^T$
return Z

Algorithm 3 ups_nEstimator: Estimate $\mathbb{E}[(S_n - \mathbb{E}[S_n])^T Z(S_n - \mathbb{E}[S_n])]$

Input: d, n, p, μ_n, Z.
Output: estimate ν_n.

$S \leftarrow p \cdot (J_d - I_d)$
$\nu_n \leftarrow (S \cdot n - \mu_n)^T Z(S \cdot n - \mu_n)$

Using these three algorithms, we can now estimate the spectral norm using the upper bound from Equation 1. Our results will be discussed in section 4.

Following this estimation, we use two different methods of generating positive definite matrices to find another upper bound for the spectral norm. In these versions, we use the fact that for any nonsingular matrix A, we know that AA^T is a positive definite matrix. Hence, if we generate any matrix, we only have to compute the determinant to know whether we can generate a positive definite matrix.

The previous part concludes the general experiments, meaning that the next experiments take a specific set of input parameters. For each of them, we will quickly describe the experiment, and the method we used to perform it.

Experiment **a** regards the visualisation of an Erdős–Rényi random graph, given the adjacency matrix A. To do this, we need a function that converts a matrix into a graph. Using `networkx`, we first generate an empty graph and fill it with d nodes, and for each 1 that appears in the adjacency matrix, we add an edge. In other words, if $A_{ij} = 1$, then we add an edge from i to j. Since the matrix is symmetric, we only need to iterate through the upper triangular values. Using the values d = 20 and q =0.3, we visualise an Erdős–Rényi graph where p(d) = q.

For our next experiment, **b**, we are once again asked to visualize the eigenvalues of an adjacency matrix, this time with different values $d \in \{100, 1000\}$ and $p \in \{0.2, 0.5, 0.8\}$. Since we have already determined a general method of doing this earlier, this experiment does not require any additional algorithms.

Next we conduct experiment **c** to choose a suitable sample size m for estimating the upper bound for $||S_{n,\text{free}}||$, with parameters $d = 10$, $n = 1$, $q = p(d) = 0.7$, and $\Sigma = I_d$. To tackle this problem, we compare the 3 methods that were discussed earlier, `zi6Gen`, `zi7AGen` and `zi7BGen`. For each method, we then visualise a plot to show the various standard deviations of $||S_{n,\text{free}}||$ for the different values of m. This experiment is performed in this way, so that we can find out which implementation of generating Z-matrices is more efficient.

In experiment **d**, a 3×9 table is generated, showing upper bounds for $||S_{n,\text{free}}||$ for different values of n and q, when $d = 10$, $m = 100$, $\Sigma = I_d$. Using `numpy`, we then find confidence intervals for each of the entries of the table. In the extension of this experiment, **e**, our aim is to increase the accuracy of the confidence intervals to two digits. We do this by first obtaining the decimal places based on the log 10 of our initial estimate of the mean of x on 100 repetitions for the desired significant figures. We then calculate how many more repetitions are necessary to achieve the desired accuracy based on the formula n $> (\frac{z_{\alpha/2} * \sigma}{\varepsilon})^2$, thus obtaining a (1 - α) confidence interval for the mean of Z of half-width $\varepsilon > 0$ over repetitions n.

After this, we perform experiment **f**. In this experiment, we are asked to estimate a different spectral norm, namely the norm of the centered adjacency matrix $(A - p(d)(J_d - I_d))$, where in this case, $p(d)$ depends on some α.

First, we generate such a matrix using `erdosRenyiGen`. Then, based on its definition, we calculate the spectral norm as the maximum absolute eigenvalue of $(A - p(d)(J_d - I_d))^T(A - p(d)(J_d - I_d))$. We call this particular estimate ξ_n. Subsequently, we use a similar method as before to find an estimate ζ_n for $||(A - p(d)(J_d - I_d))_{\text{free}}||$. In this case, we can use `zi6Gen`, `zi7AGen`, or `zi7BGen` to generate Z-matrices. Note that this quantity does not depend on any n, so we only have to generate one adjacency matrix, instead of n.
We then plot ξ_n and ζ_n against α, and once again include confidence intervals for the estimates.

Finally, for experiment **g**, we investigate the eigenvalue distribution of centered Erdős–Rényi random graphs. We do this by plotting the eigenvalues in a histogram, with the functions described earlier.

4 Results

In this segment, the results of the experiments following the implementation detailed in section 3. For each experiment the results will be listed separately.

1. After running the code to generate a histogram with arbitrarily chosen starting variables being $d = 20$ and $p = 0.3$, Figure 1 is the result. A noticeable feature of this histogram is the presence of some degree of symmetry of the eigenvalues around the 0 point with the presence of an outlier in the positive direction.

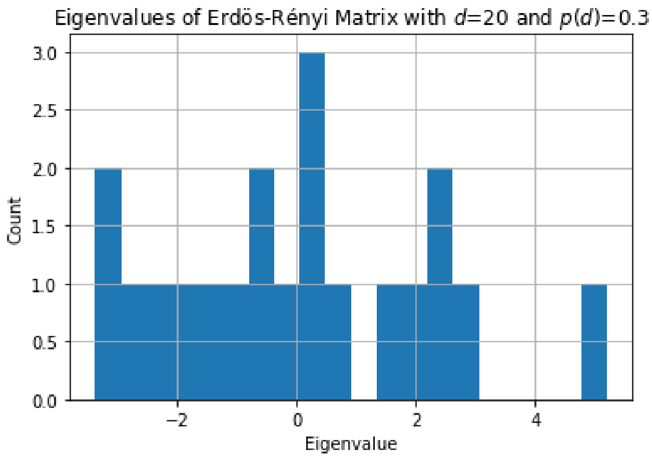


Figure 1: Eigenvalues of Erdös-Rényi adjacency matrix with $d = 20$ and $p(d) = 0.3$.

2. Using arbitrary values $d = 20$, $m = 10$, $n = 100$, $p = 0.3$ and $\Sigma = \text{Identity}(d, d)$, method 'exercise 6' gave $||S_{n,free}|| = 12073$.

3. Using the same arbitrary values from section 4 item 2., method 'exercise 7A' gave $||S_{n,free}|| = 8378$ and method 'exercise 7B' gave $||S_{n,free}|| = 7197$. These are both significantly lower as the estimate for $||S_{n,free}||$ using method 'exercise 6'. A probable reason for this is the difference in distribution of the random values of the positive definite matrix used.

a. Running the code constructing a Erdős–Rényi random graph adjacency matrix with $d = 20$ and $p(d) = 0.3$ and visualising it using a network graph results in Figure 2.

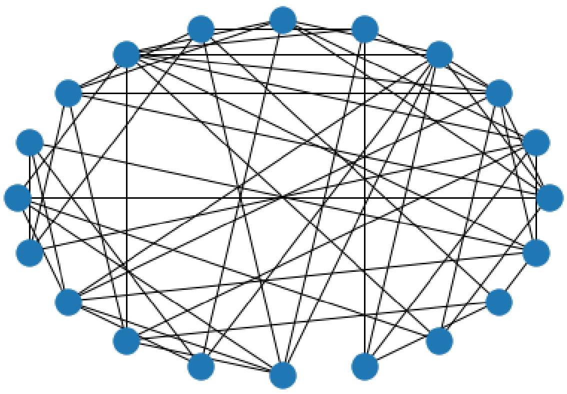


Figure 2: Erdös-Rényi random network graph with $d = 20$ and $p(d) = 0.3$.

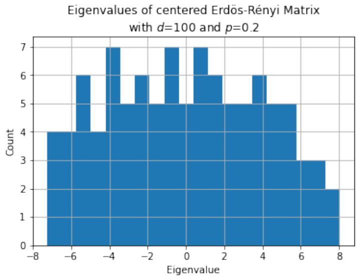
b. Setting $d \in \{100, 1000\}$ and $p(d) \in \{0.2, 0.5, 0.8\}$, generating histographs of the eigenvalues of $A - p(d)(j_d - I_d)$ with A being adjacency matrices of Erdős–Rényi random graphs results in Figure 3. From this figure, two observations can be made.

- Each histogram is roughly symmetric around zero, with the histograms generated using $d = 1000$ also trending towards forming semicircles. This is further investigated in item g.
- The spread of the histograms generated using $p = 0.2$ and $p = 0.8$ seems to be overlapping exactly. The $p = 0.5$ histogram seems to have a wider spread for both d-values.

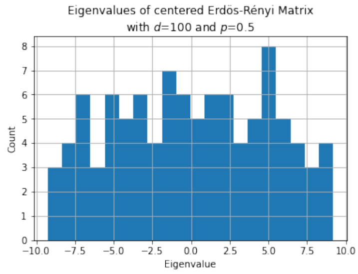
c. With $d = 10$, $n = 1$, $p(d) = 0.7$ and $\Sigma = identity(d, d)$, a graph was generated for the relative standard deviation for a range of values for m using method 'exercise 6', 'exercise 7A' and 'exercise 7B'. This Figure 4 was generated using 1000 iterations per data point to get an accurate relative standard deviation.
As is clearly visible, method 'exercise 6' is a lot faster with converging towards a low value compared to method 'exercise 7A' and 'exercise 7B', which trend roughly equally. The latter two methods do trend towards the former, however at such large values for m, the calculation time increases so quickly it becomes unfeasible to continue. If a 'sufficiently large' m had to be chosen for each method before which increasing it would be a waste of time unless strictly necessary, it would be $m = 64$ for every method.

d & e. When generating a table for $n \in \{1, 10, 100\}$ and $p(d) \in \{0.05, 0.15, \ldots, 0.95\}$ with $d = 10$, $m = 100$ and $\Sigma = identity(d, d)$, an repetition count of 100 was used to determine the confidence interval for each value. This resulted in a table that contained less than 10 entries without the desired 3 or more significant digits. This resulted in the direct inclusion of code to get each entry to the desired significant digits. Running this updated code with the aforementioned variables resulted in Table 1.
From Table 1, two very clear observations can be made.

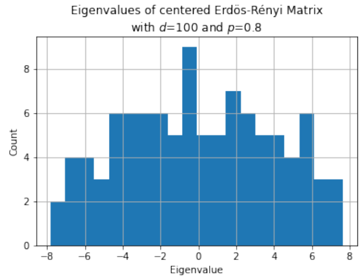
- There is symmetry between each pair that can be made using $p(d)_a = 1 - p(d)_b$. This further puts some weight behind the second observation made in section 4 item b.
- Though there is a clear correlation, multiplying n with a factor gives a roughly but not exactly equal increase in $||S_{n,free}||$.



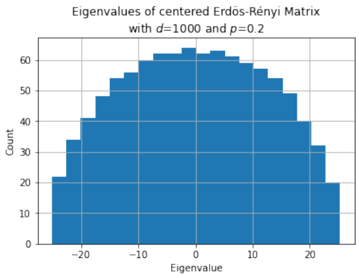
(a) $d = 100$, $p(d) = 0.2$



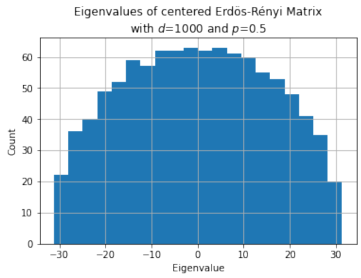
(b) $d = 100$, $p(d) = 0.5$



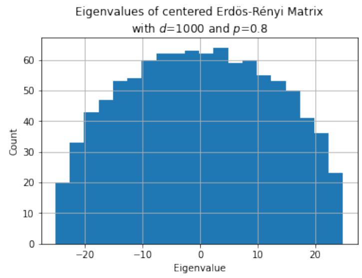
(c) $d = 100$, $p(d) = 0.8$



(d) $d = 1000$, $p(d) = 0.2$



(e) $d = 1000$, $p(d) = 0.5$



(f) $d = 1000$, $p(d) = 0.8$

Figure 3: Eigenvalue histograms for centered Erdös-Rényi random adjacency matrices for $d \in \{100, 1000\}$ and for $p(d) \in \{0.2, 0.5, 0.8\}$.

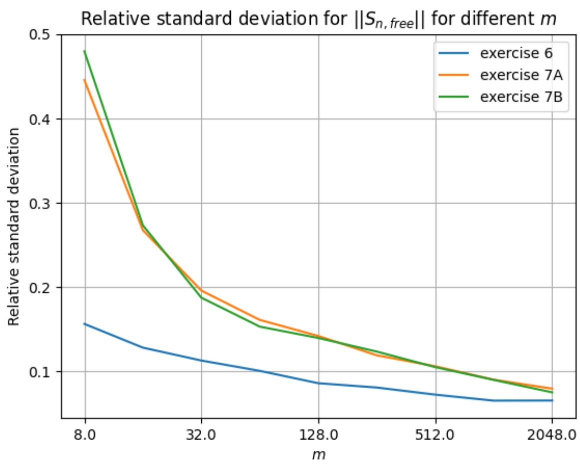
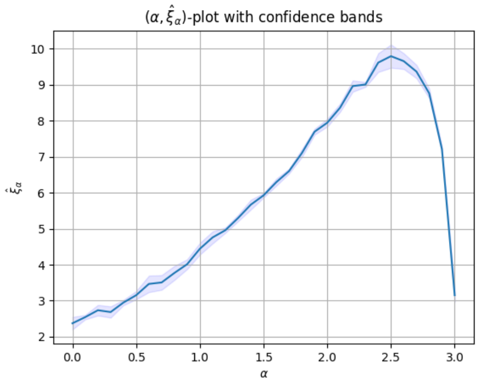
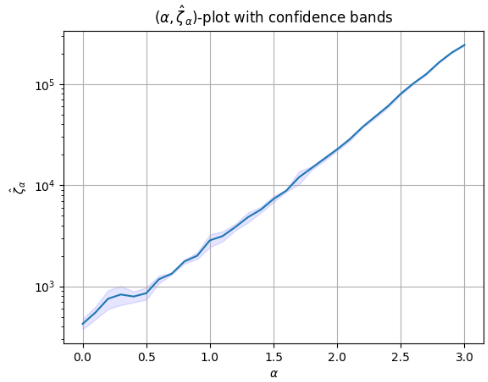


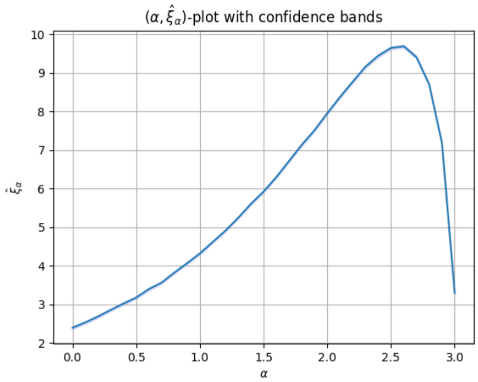
Figure 4: Relative standard deviation in $||S_{n,free}||$ for multiple methods of generating positive definite matrices for different m values.



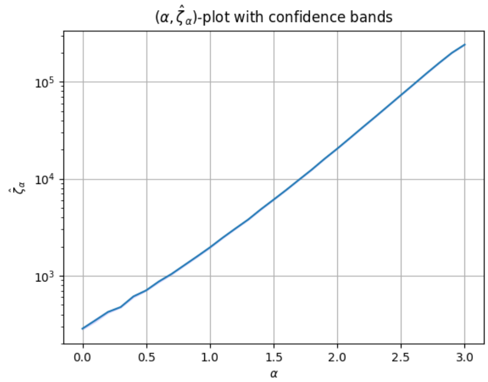
(a) ξ_α for *iterations* $= 5$ and $m = 5$



(b) ζ_α for *iterations* $= 5$ and $m = 5$



(c) ξ_α for *iterations* $= 100$ and $m = 50$



(d) ζ_α for *iterations* $= 100$ and $m = 50$

Figure 5: ξ_α and ζ_α graphs for $\alpha \in (0, 3)$ and for *iterations* $= 5$ and $m = 5$ as well as *iterations* $= 100$ and $m = 50$.

f. Setting $d = 100$ and $p(d) = (\ln d)^\alpha / d$ with $\alpha \in (0, 4)$, the code to generate both graphs was run twice. The first time with the repetition per interval at 5 and $m = 5$ and once with the repetitions at 100 and $m = 50$. For the first set, the lines are still jagged with visible confidence bands, but for the second set, the confidence bands are sufficiently small to not be visible with a smoothed out line as seen in Figure 5.

g. As referenced in section 4 item b., it was observed eigenvalue histograms of centered Erdös-Rényi random graph adjacency matrices with sufficiently large values for d tend to form a semicircle. Generating the same type of histogram as in item b. with $d = 10000$, $p = 0.5$ and 50 bars, the result can be seen in Figure 6 which even more so tends towards a semicircle. This semicircle distribution of eigenvalues for a zero-mean, symmetric random matrix is known as Wigner's semicircle law.[1]

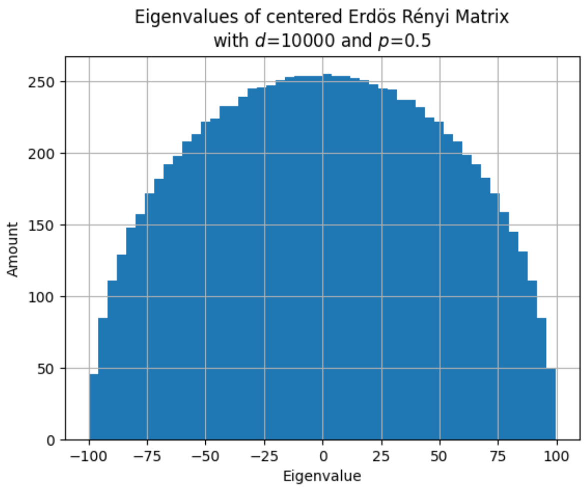


Figure 6: Centered Erdös-Rényi random adjacency graph eigenvalues for $d = 10000$ and $p = 0.5$.

Table 1: $||S_{n,free}||$ for $n \in \{1, 10, 100\}$ and $p(d) =\in \{0.05, 0.15, \ldots, 0.95\}$.

$p(d)$	$n = 1$	$n = 10$	$n = 100$
0.05	3.03±0.08	39.6±0.9	387±7
0.15	13.4±0.4	111±2	1040±20
0.25	19.3±0.4	164±3	1540±30
0.35	23.3±0.4	195±4	1910±40
0.45	25.3±0.4	216±4	2010±40
0.55	25.3±0.3	211±4	2010±40
0.65	23.4±0.4	199±4	1830±40
0.75	19.2±0.4	162±3	1510±30
0.85	13.7±0.3	110±2	1030±20
0.95	3.19±0.09	39.5±0.9	390±9

A Python Code

A.1 General code

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd

rng = np.random.default_rng()

#functions used in deliverables
def erdosRenyiGen(d, p):        #generate symmetrical binomial_p(0, 1) matrices
    matrix = rng.binomial(1, p, [d, d]) #with 0 on the diagonal, dimension dxd
    for j in range(1, d):
        for i in range(j):
            matrix[i][j] = 0
    i_d = np.identity(d)
    j_d = np.ones(d)
    matrix = matrix * (j_d - i_d)
    matrix += np.transpose(matrix)
    return matrix

def graphFromNetwork(a, d, title):  #generate a network graph from dxd matrix A
    network = nx.Graph()
    for j in range(d):
        network.add_node(j+1)
        for i in range(j):
            if i < j:
                if a[i][j] == 1:
                    network.add_edge(i+1,j+1)
    nx.draw_circular(network)
    plt.title(title)
    plt.show()


def printHist(x, title, xlabel, ylabel):  #generate histogram of x with title,
    plt.hist(x, bins=20)                   #xlabel and ylabel
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.grid(True)
    plt.show()


def zi6Gen(d, sig):                       #generate dxd positive generate matrix
    g = np.zeros((d, d))                   #using multivariate normal
    for i in range(d):
        g[i] = rng.multivariate_normal([0] * d, sig)
    zi = np.zeros((d, d))
    for i in range(d):
        zi += g[i] * g[i][:,np.newaxis]
    return zi


def zi7AGen(d):                           #generate dxd positive definite matrix
    diagMatrix = np.identity(d)            #using congruency with diagonal matrices
    congruencyMatrix = np.zeros([d, d])
    for i in range(d):
        for j in range(d):
            congruencyMatrix[i][j] = rng.uniform(0, 1)
```

```
    zi = np.dot(np.dot(np.transpose(congruencyMatrix), diagMatrix), congruencyMatrix)
    return zi


def zi7BGen(d):                             #generate dxd positive definite matrix
    Matrix = rng.uniform(0, 1, [d, d])       #using the properties of its eigenvalues
    while np.linalg.det(Matrix) == 0:
        Matrix = rng.uniform(0, 1, [d, d])
    zi = np.dot(Matrix, Matrix.transpose())
    return zi


def mu_nEstimator(d, n, p):                 #calculate average entries of n dxd matrices
    x = np.zeros((n, d, d))                  #with each non-diagonal entry binomial_p(0, 1)
    for i in range(n):                       #and the diagonal entries 0's
        x[i] = erdosRenyiGen(d, p)
    mu_n = np.sum(x, axis=0)
    return mu_n


def esn_nCalculator(d, p):                  #generate average of dxd matrix with
    j_d = np.ones(d)                         #each non-diagonal entry binomal_p(0, 1)
    i_d = np.identity(d)                     #and the diagonal entries 0's
    esn_n = p * (j_d - i_d)
    return esn_n


def ups_nEstimator(d, n, p, mu_n, zi):    #calculate upsilon_n for dxd matrix mu_n
    esn_n = esn_nCalculator(d, p)            #generated with p, n using identity matrix zi
    ups_n = np.dot(np.dot(np.transpose(esn_n * n - mu_n), zi), (esn_n * n - mu_n))
    return ups_n


def snFree6Estimator(d, m, n, p, sig):    #estimate the free spectral norm of S_n
    snFree6 = np.zeros(2)                    #using zi6Gen
    for eta in (-1, 1):
        eigenValues = np.zeros(m)
        for i in range(m):
            zi = zi6Gen(d, sig)
            mu_n = mu_nEstimator(d, n, p)
            ups_n = ups_nEstimator(d, n, p, mu_n, zi)
            summation = np.linalg.inv(zi) + eta * mu_n + ups_n
            eigenValues[i] = np.amax(summation)
        j = int((eta + 1) / 2)
        snFree6[j] = np.amin(eigenValues)
    return np.amax(snFree6)


def snFree7AEstimator(d, m, n, p):          #estimate the free spectral norm of S_n
    snFree7 = np.zeros(2)                     #using zi7AGen
    for eta in (-1, 1):
        eigenValues = np.zeros(m)
        for i in range(m):
```

```
            zi = zi7AGen(d)
            mu_n = mu_nEstimator(d, n, p)
            ups_n = ups_nEstimator(d, n, p, mu_n, zi)
            summation = np.linalg.inv(zi) + eta * mu_n + ups_n
            eigenValues[i] = np.amax(summation)
        j = int((eta + 1) / 2)
        snFree7[j] = np.amin(eigenValues)
    return np.amax(snFree7)


def snFree7BEstimator(d, m, n, p):        #estimate the free spectral norm of
    S_n
    snFree7 = np.zeros(2)                   #using zi7BGen
    for eta in (-1, 1):
        eigenValues = np.zeros(m)
        for i in range(m):
            zi = zi7BGen(d)
            mu_n = mu_nEstimator(d, n, p)
            ups_n = ups_nEstimator(d, n, p, mu_n, zi)
            summation = np.linalg.inv(zi) + eta * mu_n + ups_n
            eigenValues[i] = np.amax(summation)
        j = int((eta + 1) / 2)
        snFree7[j] = np.amin(eigenValues)
    return np.amax(snFree7)

# Validity / Accuracy tests
import pytest as pytest

#3
def test_exp_est():
    d = 20
    n = 100
    p = 0.3
    estimate_expectation = mu_nEstimator(d, n, p) / n
    test_quantity = esn_nCalculator(d, p)
    assert np.allclose(estimate_expectation, test_quantity, atol = 0.1), "test
    failed"

#4&5: assertion already done by functions
def estimateUps_n(z, mu_n, es_n_n, n): #exercise 4
    assert np.all(np.linalg.eigvals(Z) > 0)
    ups_n = np.dot(np.dot(np.transpose(es_n_n * n - mu_n), z), (es_n_n * n -
    mu_n))
    return ups_n

def exercise5(z, eta, mu_n, ups_n): #exercise 5
    assert np.all(np.linalg.eigvals(Z) > 0)
    matrix = np.linalg.inv(z) + eta * mu_n + ups_n
    eigen = np.linalg.eig(matrix)[0]
    return np.max(eigen)

#6
def test_zi6Gen_pos_def():
    d = 4
    sig = [[2,0,0,0],[0,2,0,0],[0,0,5,0],[0,0,0,8]]
    z_i = zi6Gen(d, sig)
    assert np.all(np.linalg.eigvals(z_i) > 0), "test failed"

#7
def test_zi7AGen_pos_def():
    d = 20
    z_i = zi7AGen(d)
    assert np.all(np.linalg.eigvals(z_i) > 0), "test failed"
```

```
def test_zi7BGen_pos_def():
    d = 20
    z_i = zi7BGen(d)
    assert np.all(np.linalg.eigvals(z_i) > 0), "test failed"
```

A.2 Deliverables

```
#Deliverable 1
erdosRenyiMatrix = erdosRenyiGen(d, p)
erdosRenyiEigVals = np.linalg.eigvals(erdosRenyiMatrix)
title = 'Eigenvalues of Erd s - R nyi Matrix with $d$=' + str(d) + ' and $p(d)$='  + str(p)
xlabel = 'Eigenvalue'
ylabel = 'Count'
printHist(erdosRenyiEigVals, title, xlabel, ylabel)

#Deliverable 2
print('Estimate for ||Sn,free|| using the method from exercise 6: ' + str(snFree6Estimator(d, m, n, p, sig)))

#Deliverable 3
print('Estimate for ||Sn,free|| using the method A from exercise 7: ' + str(snFree7AEstimator(d, m, n, p)))
print('Estimate for ||Sn,free|| using the method B from exercise 7: ' + str(snFree7BEstimator(d, m, n, p)))

#Deliverable A
#set specific starting variables
d = 20
q = 0.3

ERMatrix = erdosRenyiGen(d, q)    #generate random network matrix
title = 'Erd s - R nyi random network graph with $d$=' + str(d) + ' and $p(d)$=' + str(q)
graphFromNetwork(ERMatrix, d, title)     #convert network matrix to graph

#Deliverable B
#set specific starting variables
d = ([100, 1000])
p = ([0.2, 0.5, 0.8])

for di in d:
    for pi in p:
        #for each d and p, generate, normalise and get eigenvalues of network matrix
        erdosRenyiMatrix = erdosRenyiGen(di, pi)
        j_d = np.ones((di, di))
        i_d = np.identity(di)
        centeredErdosRenyi = erdosRenyiMatrix - pi * (j_d - i_d)
        centeredErdosRenyiEigVals = np.linalg.eigvals(centeredErdosRenyi)

        #generate histogram of eigenvalues of network matrix
        title = 'Eigenvalues of centered Erd s - R nyi Matrix\nwith $d$=' + str(di) + ' and $p$=' + str(pi)
        xlabel = 'Eigenvalue'
        ylabel = 'Count'
        printHist(centeredErdosRenyiEigVals, title, xlabel, ylabel)

#Deliverable C
#set specific starting variables
d = 10
n = 1
p = 0.7
```

```
sig = np.identity(d)

#variables specific for checking accuracy of ||S_n,Free|| estimate
count = 1000        #runs for calculation variance
min = 3             #actual min and max are 2^min and 2^max
max = 11            #higher value increases chance to find asymptote

exercise = ['6', '7A', '7B']
methods = np.alen(exercise)
err = np.zeros((methods, max + 1 - min))
xvalues = 2**np.linspace(min, max, num=max + 1 - min)   #only check powers of 2
    for m
results = np.zeros(count)

for method in range(methods):
    for i in range(min, max + 1):
        m = 2 ** i
        if method == 0:
            for j in range(count):    #inside method selector to increase speed
                results[j] = snFree6Estimator(d, m, n, p, sig)
        elif method == 1:
            for j in range(count):
                results[j] = snFree7AEstimator(d, m, n, p)
        elif method == 2:
            for j in range(count):
                results[j] = snFree7BEstimator(d, m, n, p)
        #use relative standard deviation to determine the error for each m
    value
        err[method][i - min] = np.std(results) / np.mean(results)

    #generate line for each method for esimating ||S_n,Free||
    label = 'exercise ' + str(exercise[method])
    plt.plot(xvalues, err[method], label=label)

title = r'Relative standard deviation for $||S_{n,free}||$ for different $m$'
plt.title(title)
xlabel = r'$m$'
plt.xlabel(xlabel)
ylabel = 'Relative standard deviation'
plt.ylabel(ylabel)
plt.xscale('log')
plt.xticks(ticks=xvalues[0::2], labels=xvalues[0::2])
plt.minorticks_off()
plt.grid(True)
plt.legend()
plt.show()

#Deliverable D&E
#set specific starting variables
d = 10
m = 100
sig = np.identity(d)
n = [1, 10, 100]
q = np.arange(0.05, 1, 0.1)

table = pd.DataFrame(columns=n, index=q)    #create empty table
repetitions = 100        #small nr of reps to estimate how many more to repeat
safetyFactor = 1.25      #used to make sure the required significant digits don'
    t undershoot
reqSigDig = 3            #desired significant digits, >2 greatly increases
    runtime
for ni in n:
    upperBoundLst = []
    for qi in q:
```

```
        sim = np.zeros(repetitions)
        for run in range(repetitions):      #run standard number of simulations
            sim[run] = snFree6Estimator(d, m, ni, qi, sig)

        avgX = np.mean(sim)                 #calculate E[X]
        sd = np.std(sim)                    #estimated standard deviation of E[X]
        ci = 1.96 * sd / np.sqrt(repetitions)

        #determine extra repetitions required
        sigDigAvgX = int(np.floor(np.log10(avgX)))
        sigDigCi = int(np.floor(np.log10(ci)))
        if (sigDigAvgX - sigDigCi) < (reqSigDig - 1):
            repScalar = (ci / (10 ** (sigDigAvgX - reqSigDig + 2)) * safetyFactor) ** 2
            extraReps = int(repetitions * (repScalar - 1))
            sim = np.append(sim, np.zeros(extraReps))
            for run in range(extraReps):    #execute extra simulation steps
                sim[run+repetitions] = snFree6Estimator(d, m, ni, qi, sig)

            avgX = np.mean(sim)             #update E[X]
            sd = np.std(sim)                #update standard deviation of E[X]
            ci = 1.96 * sd / np.sqrt(repetitions + extraReps)
            sigDigAvgX = int(np.floor(np.log10(avgX)))
            sigDigCi = int(np.floor(np.log10(ci)))
        upperBound = str(round(avgX, -sigDigCi)) + '  ' + str(round(ci, -sigDigCi))
        upperBoundLst.append(upperBound)

    table[ni] = upperBoundLst
print(table)

#Deliverable F
def p_d(d, alpha):      #convert alpha value to chance
    p_d = (np.log(d) ** alpha) / d
    return p_d


def zetaEstimator(d, m, p):   #estimate free spectral norm zeta using zi7AGen
    zeta = np.zeros(2)
    for eta in (-1, 1):
        eigenValues = np.zeros(m)
        for i in range(m):
            zi = zi7AGen(d)
            mu_n = erdosRenyiGen(d, p)
            esn_n = np.zeros((d, d))
            ups_n = np.dot(np.dot(np.transpose(esn_n - mu_n), zi), (esn_n - mu_n))
            summation = np.linalg.inv(zi) + eta * mu_n + ups_n
            eigenValues[i] = np.amax(summation)
        j = int((eta + 1) / 2)
        zeta[j] = np.amin(eigenValues)
    return np.amax(zeta)


#set variables for the graph
repetitions = 5
steps = 31

#set starting variables
d = 100
m = 5
alphaRange = np.linspace(0, 3, steps, endpoint=True)

#prepare data structures
```

```
xiMeans = np.zeros(steps)
xiStds = np.zeros(steps)
xiCis = np.zeros(steps)
zetaMeans = np.zeros(steps)
zetaStds = np.zeros(steps)
zetaCis = np.zeros(steps)
xi = np.zeros(repetitions)
zeta = np.zeros(repetitions)

for i in range(steps):
    p = p_d(d, alphaRange[i])
    for n in range(repetitions):
        #calculate xi
        A = erdosRenyiGen(d, p)
        esn = esn_nCalculator(d, p)
        A_centered = A - esn
        xi[n] = np.sqrt(np.max(np.absolute(np.linalg.eigvals(np.dot(np.transpose(A_centered), A_centered)))))

        #calculate zeta
        zeta[n] = zetaEstimator(d, m, p)
    #get estimates and standard deviations
    xiMeans[i] = np.mean(xi)
    xiStds[i] = np.std(xi)
    zetaMeans[i] = np.mean(zeta)
    zetaStds[i] = np.std(zeta)

#calculate confidence intervals
xiCis = 1.96 * xiStds / np.sqrt(repetitions)
zetaCis = 1.96 * zetaStds / np.sqrt(repetitions)

#generate plots
plt.plot(alphaRange, xiMeans)
title = r'($\alpha, \hat\xi_\alpha$)-plot with confidence bands'
plt.title(title)
xlabel = r'$\alpha$'
plt.xlabel(xlabel)
ylabel = r'$\hat\xi_\alpha$'
plt.ylabel(ylabel)
plt.grid(True)
plt.fill_between(alphaRange, (xiMeans - xiCis), (xiMeans + xiCis), color='b', alpha=.1)
plt.show()

plt.plot(alphaRange, zetaMeans)
title = r'($\alpha, \hat\zeta_\alpha$)-plot with confidence bands'
plt.title(title)
xlabel = r'$\alpha$'
plt.xlabel(xlabel)
ylabel = r'$\hat\zeta_\alpha$'
plt.ylabel(ylabel)
plt.yscale('log')
plt.grid(True)
plt.fill_between(alphaRange, (zetaMeans - zetaCis), (zetaMeans + zetaCis), color='b', alpha=.1)
plt.show()

#Deliverable G
p = 0.5
d = 10000
erdosRenyiMatrix = erdosRenyiGen(d, p)
j_d = np.ones((d, d))
i_d = np.identity(d)
centeredErdosRenyi = erdosRenyiMatrix - p * (j_d - i_d)
```

```
centeredErdosRenyiEigVals = np.linalg.eigvals(centeredErdosRenyi)
title = 'Eigenvalues of centered Erd s R nyi Matrix\nwith $d$=' + str(d) + ' and $p$=' + str(p)
xlabel = 'Eigenvalue'
ylabel = 'Amount'
plt.hist(centeredErdosRenyiEigVals, bins=50)
plt.title(title)
plt.xlabel(xlabel)
plt.ylabel(ylabel)
plt.grid(True)
plt.show()
```

B Bibliography

1 Alon, N.; Krivelevich, M.; and Vu, V. H. "On the Concentration of Eigenvalues of Random Symmetric Matrices." Israel J. Math. 131, 259-267, 2002.