Homework-2

1. Householder Reflector $F = I - \frac{2vv^*}{v^*v}$

where $v = \|x\|_2 e_1 - x$.

ⓐ $Fy = \lambda y$ if y is an eigenvector and λ is the associated eigenvalue.

$$y - \frac{2vv^*}{v^*v} y = \lambda y$$

$$(1-\lambda) y = \left(\frac{2vv^*}{v^*v}\right) y$$

$$(1-\lambda) y = \frac{2(v^*y)}{v^*v} v$$

So one possible solution is $\boxed{\lambda = 1}$ and the corresponding eigenvector y is perpendicular to v (or the RHS, $v^*y = 0$)

If $\lambda \neq 1$, we can see that y is a scalar multiple of v.

If $y = \alpha v$, $\alpha \in \mathbb{C}$, then $(1-\lambda)\alpha = 2\alpha$ or $\boxed{\lambda = -1}$

The corresponding set of eigenvectors are scalar multiples of v.

ⓑ We know $F^2 = I$

$$(I - 2q_1 q_1^*)(I - 2q_1 q_1^*)$$

$$= I - 4q_1 q_1^* + 4 q_1 (q_1^* q_1) q_1^* = I$$

So $\det(F) = \pm 1$

Now, consider $\det(I + xy^*)$ can be written as $\det\left(\begin{bmatrix} 1 & -y^* \\ x & I \end{bmatrix}\right)$. (where $xy^* = I$)

$= I - x^* y$ (using the determinant expression for a 2×2 matrix).

So $\det(F) = \det\left(I - 2\dfrac{vv^*}{v^*v}\right)$

If $\dfrac{v}{\sqrt{v^*v}} = q$, then

$$\boxed{\det(F) = \det(I - 2qq^*) = \det(I - 2q^*q) = -1}$$

(c) $F^* = \left(I - 2\dfrac{vv^*}{v^*v}\right)^* = I - 2\dfrac{vv^*}{v^*v} = F$

Since F is Hermitian, it has real eigenvalues and the eigendecomposition of $F = Q \Lambda Q^*$ yields a orthonormal matrix Q $(Q^* = Q^{-1})$

So, we can write $F = Q|\Lambda| \,\mathrm{sign}(\Lambda) Q^*$, $\mathrm{sign}(\Lambda)$ is a diagonal matrix with signs of each of the eigenvalues in Λ.

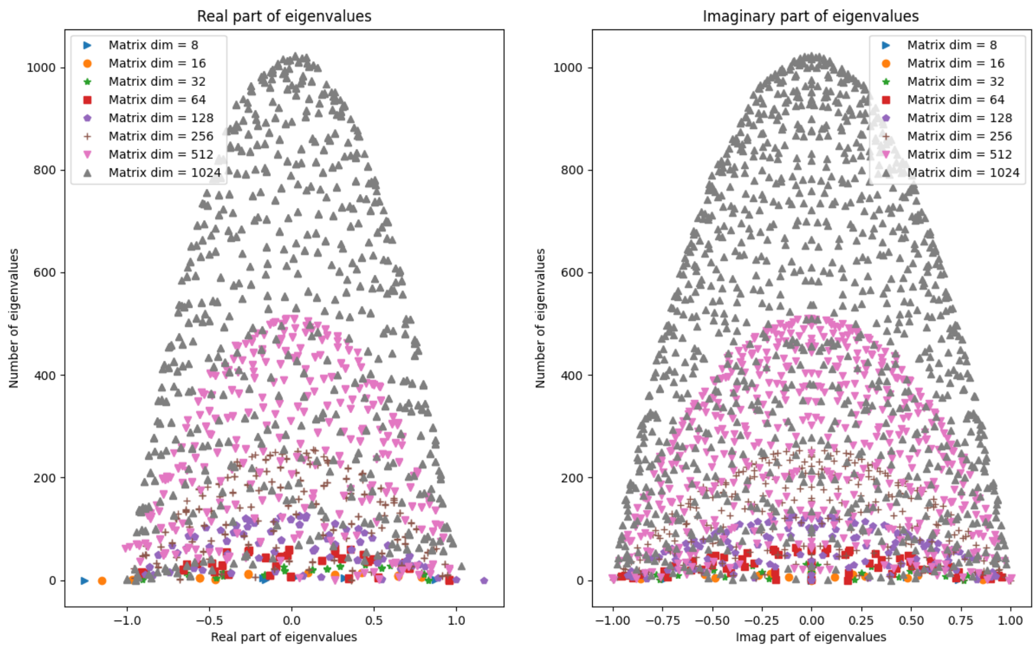
Since $\mathrm{Sign}(\Lambda) Q^*$ is unitary, we have ourselves a singular value decomposition for F.

$$\boxed{\Rightarrow \text{The singular values of } F = |\pm 1| = 1}$$

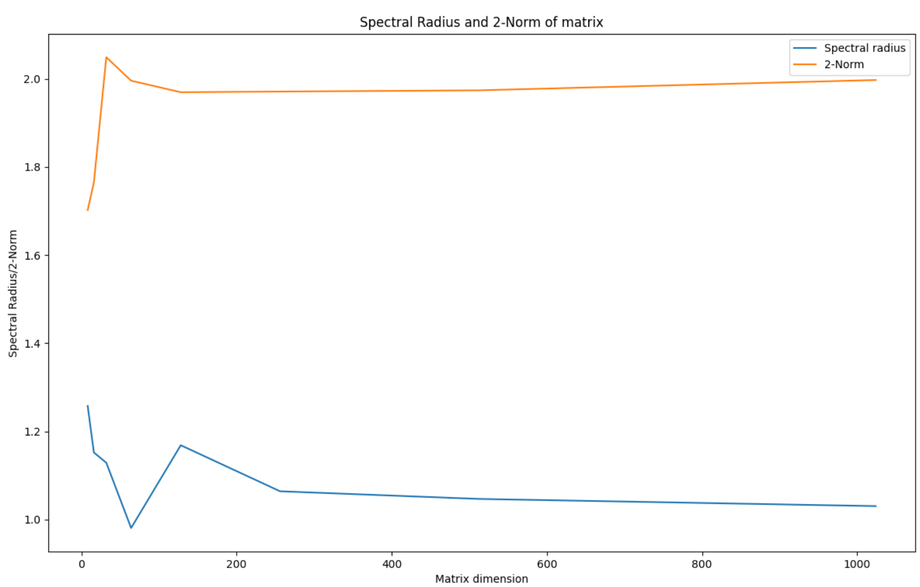
(since the eigenvalues of $F = \pm 1$).

Problem 2

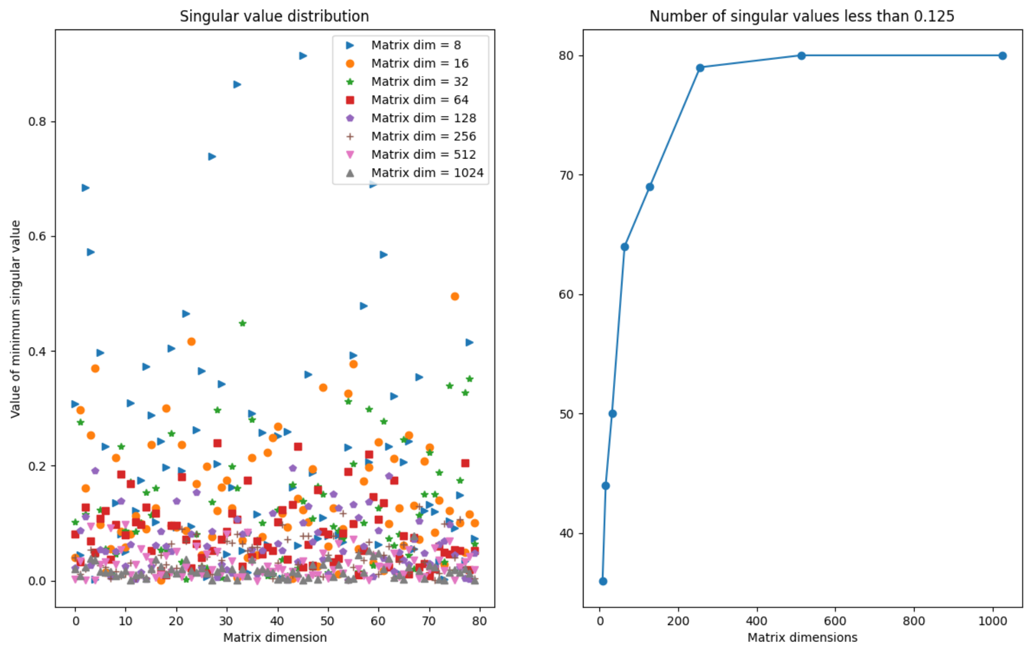
A. Eigenvalues superimposed in a single plot for different values of m (matrix dimension). The real and imaginary parts of the eigenvalues lie between -1.0 and 1.0 for all matrix dimensions.



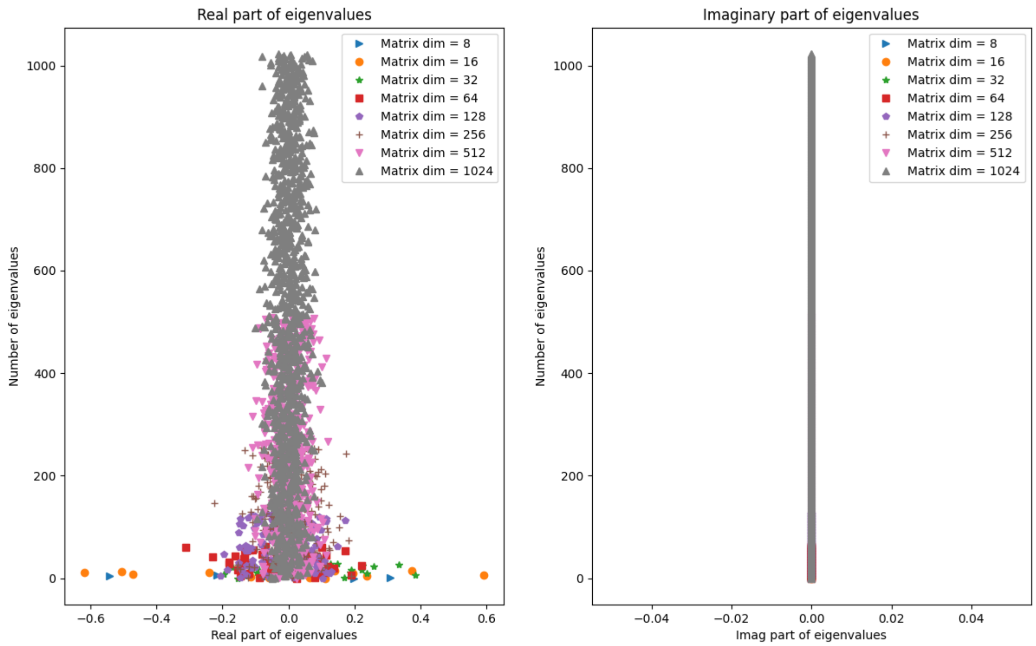
B. Spectral radius and 2-Norm of the matrix
The spectral radius tends to 2 as m tends to infinity. The 2-Norm of the matrix tends to 1 as m tends to infinity.



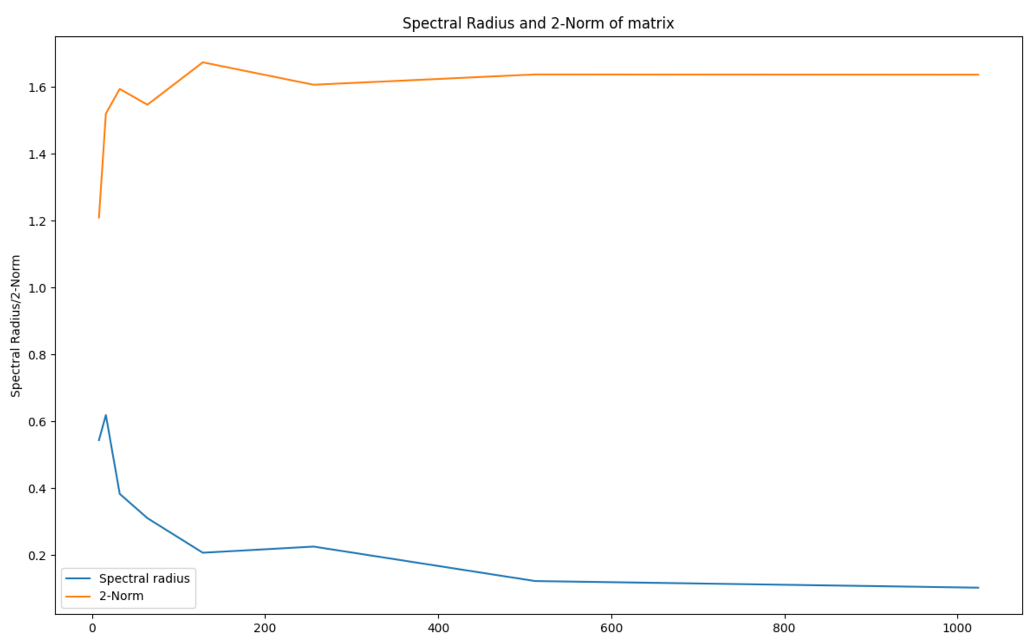
C. Smallest singular value of the matrix
   As the matrix dimension increases, the number of minimum singular values less than (½)^n where n = {-1, -2, …} increases. For example, m = 1024, the number of singular values less than ⅛ is higher than m = 16.



D. Upper Triangular Matrix with normally distributed entries
   a. Eigenvalue distribution
   We can see that the real part has a gaussian distribution and imaginary parts of the eigenvalues are 0 for all matrix dimensions.

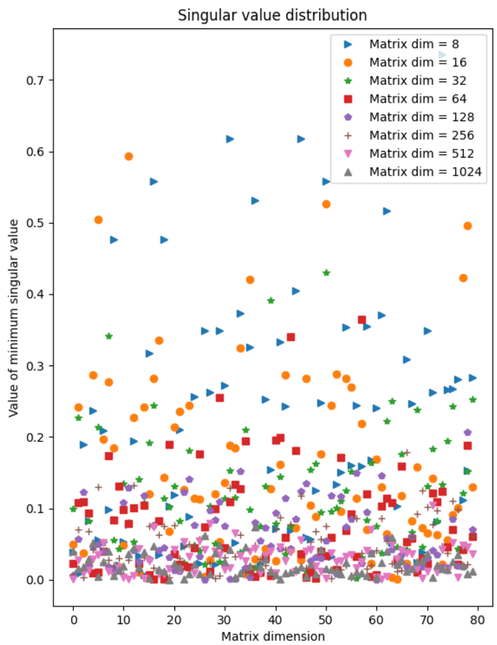


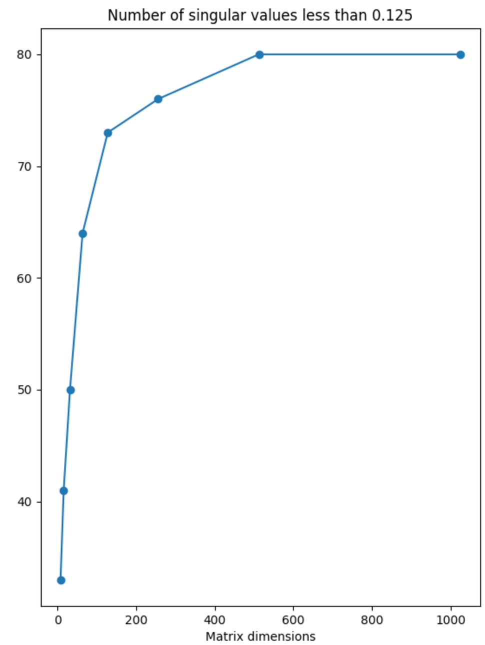
b. Spectral Radius and 2-Norm plot
The spectral radius tends to 1.6 and the 2-norm tends to 0 as m tends to infinity.



c. Smallest singular value of the matrix

As the matrix dimension increases, the number of minimum singular values less than $(\frac{1}{2})^n$ where $n = \{-1, -2, \ldots\}$ increases. For example, $m = 1024$, the number of singular values less than $\frac{1}{8}$ is higher than $m = 16$. However, the curve on the right is smoother than the curve for the above case.

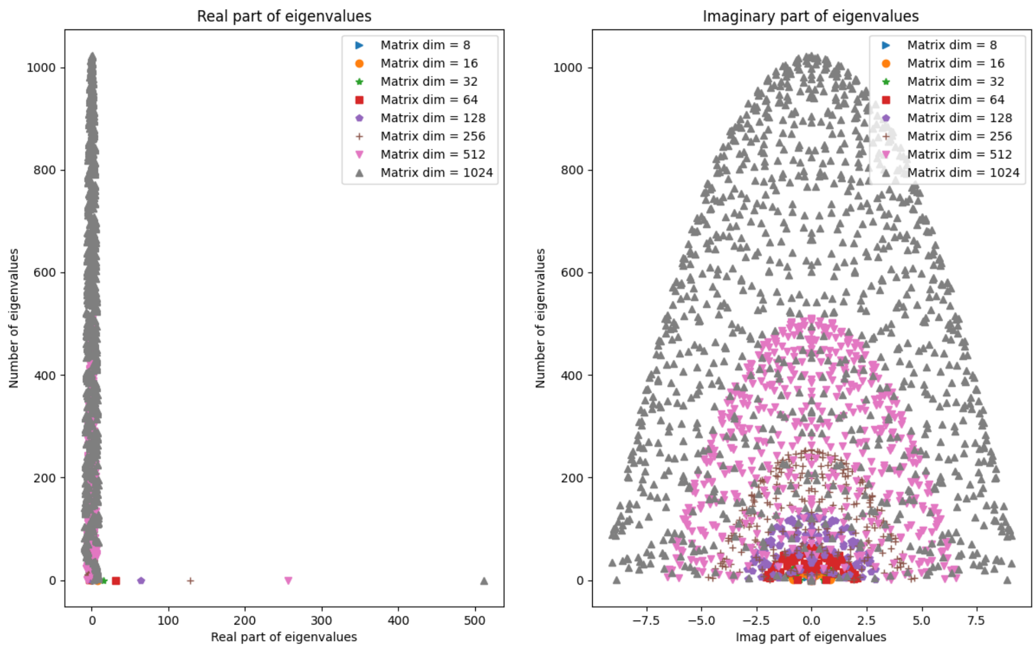




Problem 3

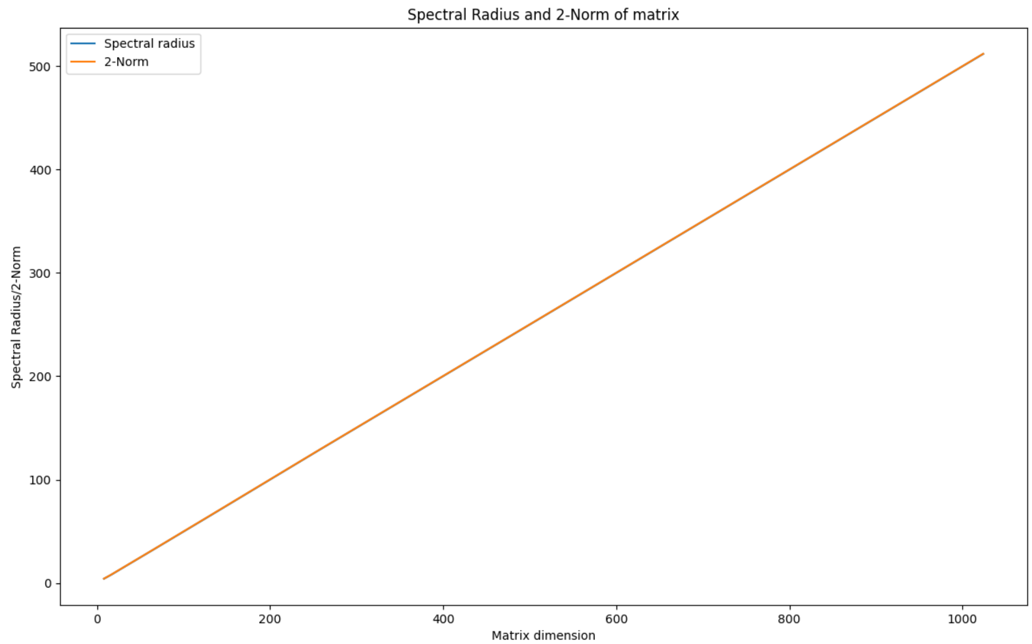
A. Distribution of Eigenvalues

The real part of the eigenvalues lie close to 0 and imaginary parts of the eigenvalues lie between -8.0 and 8.0 for all matrix dimensions.



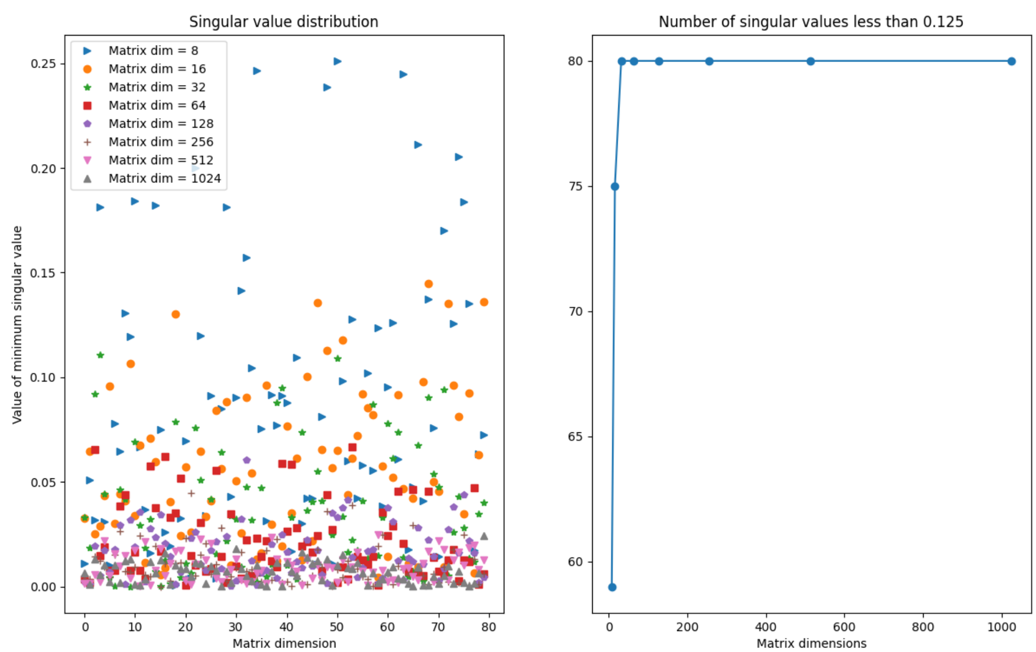
B. Spectral Radius and 2-Norm

The spectral radius and 2-norm curves overlap each other for all matrix dimensions and increase linearly with matrix dimension.



C. Distribution of minimum singular values

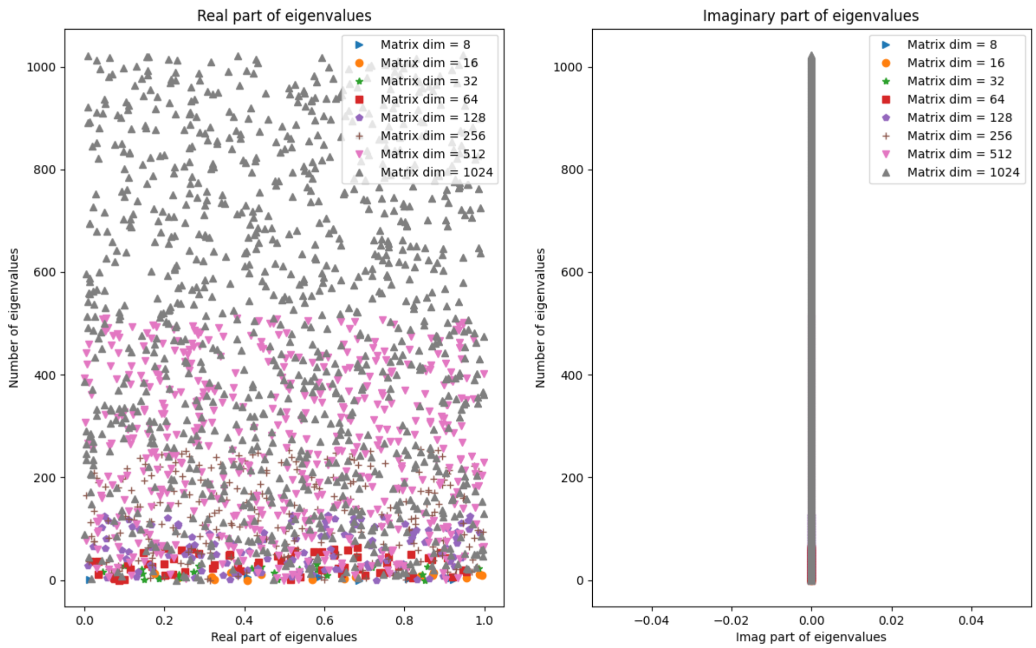
The minimum singular value distribution for matrices with uniformly distributed entries shows that higher numbers of singular values are close to 0 as m tends to infinity.



D. Upper triangular uniform distribution

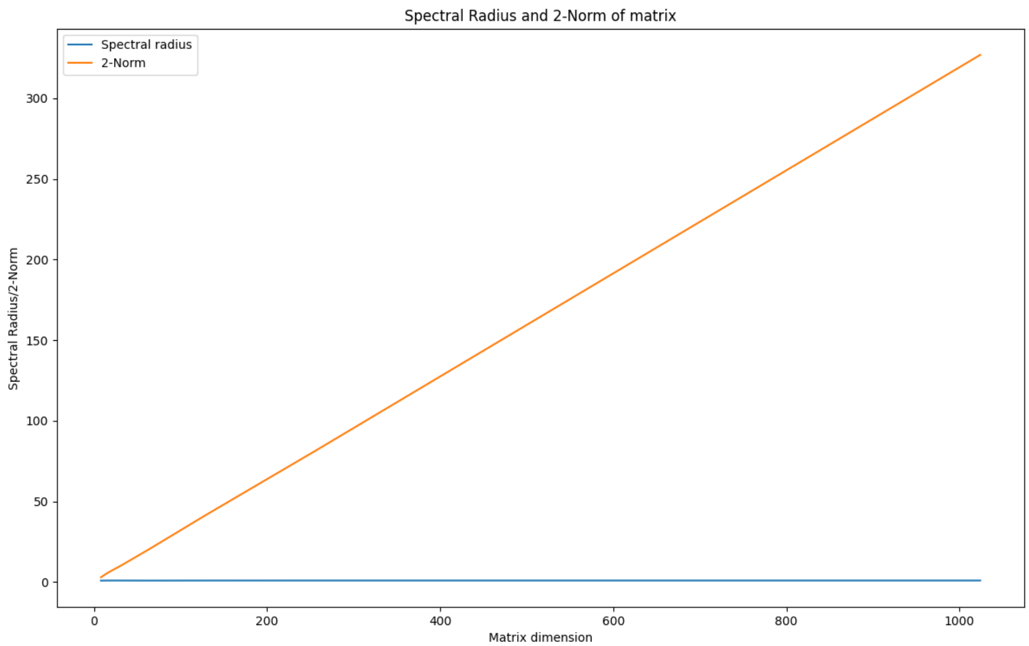
a. Eigenvalue distribution

The real part of eigenvalues lie between 0 and 1 and the imaginary part is 0.



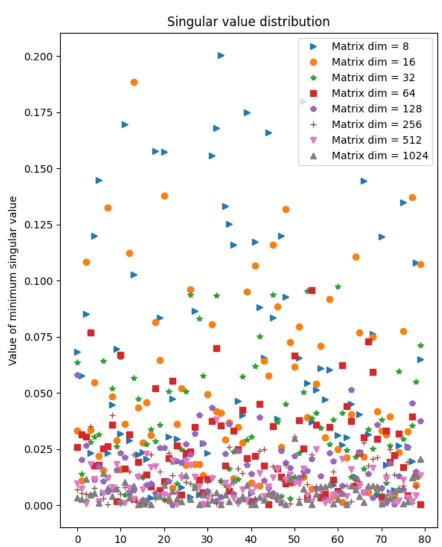
b. Spectral radius and 2-norm

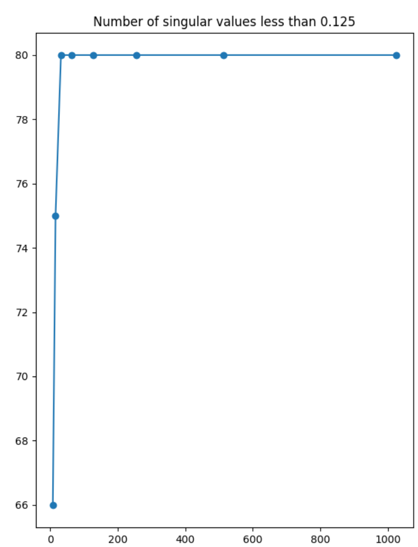
The 2-norm is similar to the curve in the case for the full matrix, but the spectral radius is 0.



c. Minimum singular value distribution

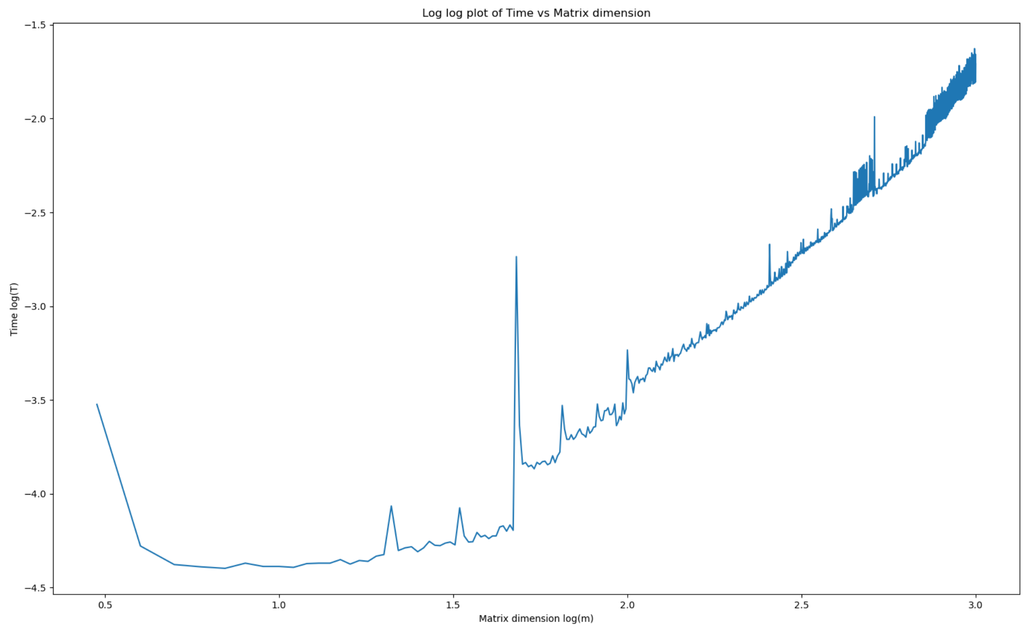
The minimum singular value distribution for matrices with uniformly distributed entries shows that higher numbers of singular values are close to 0 as m tends to infinity.





Problem 4

a.



The slope of this curve = 1.023844779280685 (end to end slope)
As the matrix dimension increases, the time taken first reduces and then starts increasing.

b. M = 1000

Spectral Radius = 3.9999901501133026
Max Singular Value = 3.999990150113325
Min Singular Value = 9.849886676613993e-06
Condition Number = 406095.04265772586
Frobenius Norm = 77.44675590365293
2 Norm = 77.44675590365293

g. Summarizing times for all methods

From the table, we can see that the SVD takes the longest time to decompose the matrix.
We also see that LU and Cholesky have the least residual for b-Ax, and so are most accurate for this matrix.
Also, np.linalg.solve has the same residual as LU, which means that the numpy library uses LU decomposition underneath.

Type of decomposition	Time taken	|b-Ax|
QR	2.823699951171875 s	5.3652346348568966e-05
LU	0.61257004737854 s	2.720789863768137e-05

SVD	18.694941997528076 s	0.0007673158470886349
Cholesky	0.49833083152770996 s	3.899493143684322e-05
np.linalg.solve	0.44628095626831055 s	2.720789863768137e-05

Code for Problem 2

```
from cProfile import label
import numpy as np
import matplotlib.pyplot as plt
from torch import norm, threshold

def generate_random_matrix(m):
    mat = np.random.randn(m, m)
    return mat

def make_upper_traingular(mat):
    return np.triu(mat)

def normalize_matrix(mat, m):
    mat = mat / np.sqrt(m)
    return mat

def get_eigen_values(mat):
    v, _ = np.linalg.eig(mat)
    return v

def get_spectral_radius(eig_vals):
    return np.max(np.absolute(eig_vals))

def get_norm_2(mat):
    return np.linalg.norm(mat, 2)

def get_smallest_sigma(mat):
    sg_min = np.min(np.linalg.svd(mat, compute_uv=False))
    return sg_min

def plot_eig_vals(eig_values, m_sizes, markers):
    plt.clf()
    plt.cla()
    fig, (ax1, ax2) = plt.subplots(1, 2)
```

```
    num_plots = len(m_sizes)
    for i in range(num_plots):
        ei = eig_values[i]
        ei_real = np.real(ei)
        ei_complex = np.imag(ei)
        m = m_sizes[i]
        y = np.arange(m)
        ax1.plot(ei_real, y, markers[i], label='Matrix dim = '+str(m))
        ax2.plot(ei_complex, y, markers[i], label='Matrix dim = '+str(m))
    ax1.legend()
    ax1.set_xlabel('Real part of eigenvalues')
    ax1.set_ylabel('Number of eigenvalues')
    ax1.set_title('Real part of eigenvalues')
    ax2.legend()
    ax2.set_xlabel('Imag part of eigenvalues')
    ax2.set_ylabel('Number of eigenvalues')
    ax2.set_title('Imaginary part of eigenvalues')
    # plt.legend()
    plt.show()

def plot_spectral_radius(sr, nm, m):
    plt.clf()
    plt.cla()
    plt.plot(m, sr, label='Spectral radius')
    plt.plot(m, nm, label='2-Norm')
    plt.title("Spectral Radius and 2-Norm of matrix")
    plt.xlabel("Matrix dimension")
    plt.ylabel("Spectral Radius/2-Norm")
    plt.legend()
    plt.show()

def plot_min_singular_distribution(m_list, threshold, markers):
    plt.clf()
    plt.cla()
    fig, (ax1, ax2) = plt.subplots(1, 2)
    sg_min_tail = []
    j = 0
    for m in m_list:
        sg_min_list = []
        count = 0
        for i in range(80):
```

```
            mat = generate_random_matrix(m)
            sg_min = get_smallest_sigma(mat)
            sg_min_list.append(sg_min)
            if (sg_min < threshold):
                count += 1
        sg_min_tail.append(count)
        ax1.plot(sg_min_list, markers[j], label='Matrix dim = {}'.format(m))
        j += 1
    ax1.set_title('Singular value distribution')
    ax1.legend()
    ax1.set_xlabel("Matrix dimension")
    ax1.set_ylabel("Value of minimum singular value")
    ax2.plot(m_list, sg_min_tail, 'o-')
    ax2.set_title('Number of singular values less than {}'.format(threshold))
    ax2.set_xlabel("Matrix dimensions")
    plt.show()
    return

def run_experiment(m_list):
    eig_values = []
    spectral_radius = []
    norm_2 = []
    threshold = 1./np.power(2, 3)
    for m in m_list:
        mat = generate_random_matrix(m)
        mat = make_upper_traingular(mat)
        mat = normalize_matrix(mat, m)
        ei = get_eigen_values(mat)
        eig_values.append(ei)
        sr = get_spectral_radius(ei)
        spectral_radius.append(sr)
        n2 = get_norm_2(mat)
        norm_2.append(n2)
    markers = ['>', 'o', '*', 's', "p", "+", "v", "^", "x"]
    plot_eig_vals(eig_values, m_list, markers)
    plot_spectral_radius(spectral_radius, norm_2, m_list)
    # m_copy = [m_list[4]]
    plot_min_singular_distribution(m_list, threshold, markers)

if __name__ == '__main__':
    m_list = [8, 16, 32, 64, 128, 256, 512, 1024]
```

```
    run_experiment(m_list)
```

Code for Problem 3

```
from cProfile import label
import numpy as np
import matplotlib.pyplot as plt
from torch import norm, threshold

def generate_random_matrix(m):
    mat = np.random.rand(m, m)
    return mat

def make_upper_traingular(mat):
    return np.triu(mat)

def normalize_matrix(mat, m):
    mat = mat / np.sqrt(m)
    return mat

def get_eigen_values(mat):
    v, _ = np.linalg.eig(mat)
    return v

def get_spectral_radius(eig_vals):
    return np.max(np.absolute(eig_vals))

def get_norm_2(mat):
    return np.linalg.norm(mat, 2)

def get_smallest_sigma(mat):
    sg_min = np.min(np.linalg.svd(mat, compute_uv=False))
    return sg_min

def plot_eig_vals(eig_values, m_sizes, markers):
    plt.clf()
    plt.cla()
    fig, (ax1, ax2) = plt.subplots(1, 2)
    num_plots = len(m_sizes)
    for i in range(num_plots):
```

```
        ei = eig_values[i]
        ei_real = np.real(ei)
        ei_complex = np.imag(ei)
        m = m_sizes[i]
        y = np.arange(m)
        ax1.plot(ei_real, y, markers[i], label='Matrix dim = '+str(m))
        ax2.plot(ei_complex, y, markers[i], label='Matrix dim = '+str(m))
    ax1.legend()
    ax1.set_xlabel('Real part of eigenvalues')
    ax1.set_ylabel('Number of eigenvalues')
    ax1.set_title('Real part of eigenvalues')
    ax2.legend()
    ax2.set_xlabel('Imag part of eigenvalues')
    ax2.set_ylabel('Number of eigenvalues')
    ax2.set_title('Imaginary part of eigenvalues')
    # plt.legend()
    plt.show()

def plot_spectral_radius(sr, nm, m):
    plt.clf()
    plt.cla()
    plt.plot(m, sr, label='Spectral radius')
    plt.plot(m, nm, label='2-Norm')
    plt.title("Spectral Radius and 2-Norm of matrix")
    plt.xlabel("Matrix dimension")
    plt.ylabel("Spectral Radius/2-Norm")
    plt.legend()
    plt.show()

def plot_min_singular_distribution(m_list, threshold, markers):
    plt.clf()
    plt.cla()
    fig, (ax1, ax2) = plt.subplots(1, 2)
    sg_min_tail = []
    j = 0
    for m in m_list:
        sg_min_list = []
        count = 0
        for i in range(80):
            mat = generate_random_matrix(m)
            sg_min = get_smallest_sigma(mat)
```

```
            sg_min_list.append(sg_min)
            if (sg_min < threshold):
                count += 1
        sg_min_tail.append(count)
        ax1.plot(sg_min_list, markers[j], label='Matrix dim = {}'.format(m))
        j += 1
    ax1.set_title('Singular value distribution')
    ax1.legend()
    ax1.set_xlabel("Matrix dimension")
    ax1.set_ylabel("Value of minimum singular value")
    ax2.plot(m_list, sg_min_tail, 'o-')
    ax2.set_title('Number of singular values less than {}'.format(threshold))
    ax2.set_xlabel("Matrix dimensions")
    plt.show()
    return

def run_experiment(m_list):
    eig_values = []
    spectral_radius = []
    norm_2 = []
    threshold = 1./np.power(2, 3)
    for m in m_list:
        mat = generate_random_matrix(m)
        mat = make_upper_traingular(mat)
        # mat = normalize_matrix(mat, m)
        ei = get_eigen_values(mat)
        eig_values.append(ei)
        sr = get_spectral_radius(ei)
        spectral_radius.append(sr)
        n2 = get_norm_2(mat)
        norm_2.append(n2)
    markers = ['>', 'o', '*', 's', "p", "+", "v", "^", "x"]
    plot_eig_vals(eig_values, m_list, markers)
    plot_spectral_radius(spectral_radius, norm_2, m_list)
    # m_copy = [m_list[4]]
    plot_min_singular_distribution(m_list, threshold, markers)

if __name__ == '__main__':
    m_list = [8, 16, 32, 64, 128, 256, 512, 1024]
    run_experiment(m_list)
```

Code for Problem 4

```
import numpy as np
import matplotlib.pyplot as plt
import scipy as scp
from scipy import linalg as scplinalg
from torch import permute
import time

def construct_1D_laplace(m):
   diagonal = -2. * np.ones(m)
   off_diag = np.ones(m-1)
   return np.diag(diagonal, 0) + np.diag(off_diag, 1) + np.diag(off_diag, -1)

def construct_rhs(m):
   b = np.arange(m)
   b = b.astype(np.float64)
   return b

def get_spectral_radius(mat):
   ei, _ = np.linalg.eig(mat)
   sr = np.abs(ei)
   sr = np.max(sr)
   return sr

def get_singular_value_min_max(mat):
   sg = np.linalg.svd(mat, compute_uv=False)
   return np.max(sg), np.min(sg)

def get_frobenius_norm(mat):
   return np.linalg.norm(mat, 'fro')

def get_2_norm(mat):
   return np.linalg.norm(mat, 2)

def compute_condition_number(mat):
   return np.linalg.cond(mat)

# Get Spectral radius, condition number, sigma_max/min, matrix norm fro, 2
def get_matrix_characteristics(mat):
   sr = get_spectral_radius(mat)
```

```python
    sg_max, sg_min = get_singular_value_min_max(mat)
    norm_f = get_frobenius_norm(mat)
    norm_2 = get_frobenius_norm(mat)
    cnum = compute_condition_number(mat)
    return sr, sg_max, sg_min, cnum, norm_f, norm_2

# QR Decomposition
def get_QR_decomposition(mat):
    q, r = np.linalg.qr(mat)
    return q, r

def check_qr_correctness(mat, q, r):
    diff = np.linalg.norm(mat - np.matmul(q, r))
    print("Diff in QR = {}".format(diff))

def solve_using_qr(q, r, b):
    start = time.time()
    y = np.matmul(np.transpose(q), b)
    end = time.time() - start
    x = scplinalg.solve_triangular(r, y)
    return x, end

# LU Decomposition
def get_lu_decomposition(mat):
    p, l, u = scp.linalg.lu(mat, permute_l=False)
    return p, l, u

def check_lu_correctness(mat, p, l, u):
    diff = np.linalg.norm(mat - p @ l @ u)
    print("Diff in LU = {}".format(diff))

def solve_using_lu(p, l, u, b):
    b1 = np.matmul(np.transpose(p), b)
    y = scplinalg.solve_triangular(l, b1, lower=True)
    x = scplinalg.solve_triangular(u, y, lower=False)
    return x

# SVD Decomposition
def get_svd_decomposition(mat):
    u, sigma, vh = np.linalg.svd(mat)
    return u, sigma, vh
```

```python
def check_svd_correctness(mat, u, sigma, vh):
    diff = np.linalg.norm(mat - u @ np.diag(sigma) @ vh)
    print("Diff in SVD = {}".format(diff))

def solve_using_svd(u, sigma, vh, b):
    c = np.dot(np.transpose(u),b)
    w = np.dot(np.diag(1./sigma),c)
    # Vh x = w <=> x = Vh.H w (where .H stands for hermitian = conjugate transpose)
    x = np.dot(np.transpose(vh),w)
    return x

# Cholesky Decomposition
def get_cholesky_decomposition(mat):
    L = np.linalg.cholesky(mat)
    return L

def check_cholesky_correctness(mat, L):
    diff = np.linalg.norm(mat - np.matmul(L, np.transpose(L)))
    print("Diff in Cholesky = {}".format(diff))

def solve_using_cholesky(L, b):
    y = scplinalg.solve_triangular(L, b, lower=True)
    x = scplinalg.solve_triangular(np.transpose(L), y, lower=False)
    return x

# Solve using numpy.linalg.solve
def solve_using_np_solve(mat, b):
    x = np.linalg.solve(mat, b)
    return x

# Verify norm of |b-Ax|
def verify_solution(mat, x, b):
    diff = np.linalg.norm(np.matmul(mat, x) - b)
    print("|b - Ax| = {}".format(diff))

#
def log_plot_qr_time():
    times = []
    m_list = []
    for m in range(3, 1001):
```

```
        mat = construct_1D_laplace(m)
        b = construct_rhs(m)
        time_qr = time.time()
        q, r = get_QR_decomposition(mat)
        y = np.matmul(np.transpose(q), b)
        time_qr = time.time() - time_qr
        y = y + 1
        times.append(np.log10(time_qr))
        m_list.append(np.log10(m))
    print(np.array(times[1:]) - np.array(times[0:-1])/(np.array(m_list[1:]) -
np.array(m_list[0:-1])))
    plt.clf()
    plt.cla()
    plt.plot(m_list, times)
    plt.title("Log log plot of Time vs Matrix dimension")
    plt.xlabel("Matrix dimension log(m)")
    plt.ylabel("Time log(T)")
    plt.show()

def run_experiment(m):
    mat = construct_1D_laplace(m)
    b = construct_rhs(m)

    sr, sg_max, sg_min, cnum, norm_f, norm_2 =
get_matrix_characteristics(construct_1D_laplace(1000))
    print("Spectral Radius     = {}".format(sr))
    print("Max Singular Value = {}".format(sg_max))
    print("Min Singular Value = {}".format(sg_min))
    print("Condition Number    = {}".format(cnum))
    print("Frobenius Norm      = {}".format(norm_f))
    print("2 Norm              = {}".format(norm_2))

    # Log time vs Log dimension
    log_plot_qr_time()

    # Solve using QR
    print("Solve using QR")
    time_qr = time.time()
    q, r = get_QR_decomposition(mat)
    time_qr = time.time() - time_qr
    check_qr_correctness(mat, q, r)
```

```
    x_qr, time_qr1 = solve_using_qr(q, r, b)
    time_qr += time_qr1
    print("Time taken for QR decomposition and y=Q*b = {} s".format(time_qr))
    verify_solution(mat, x_qr, b)

    # Solve using LU
    print("Solve using LU")
    time_lu = time.time()
    p, l, u = get_lu_decomposition(mat)
    time_lu = time.time() - time_lu
    check_lu_correctness(mat, p, l, u)
    time_lu_solve = time.time()
    x_lu = solve_using_lu(p, l, u, b)
    time_lu += time.time() - time_lu_solve
    print("Time taken for LU decomposition and solve = {} s".format(time_lu))
    verify_solution(mat, x_lu, b)

    # Solve using SVD
    print("Solve using SVDs")
    time_svd = time.time()
    u, sigma, vh = get_svd_decomposition(mat)
    time_svd = time.time() - time_svd
    check_svd_correctness(mat, u, sigma, vh)
    time_svd_solve = time.time()
    x_svd = solve_using_svd(u, sigma, vh, b)
    time_svd += time.time() - time_svd_solve
    print("Time taken for SVD decomposition and solve = {} s".format(time_svd))
    verify_solution(mat, x_svd, b)

    # Solve using Cholesky
    print("Solve using Cholesky")
    time_cholesky = time.time()
    L = get_cholesky_decomposition(-1.0 * mat)
    time_cholesky = time.time() - time_cholesky
    check_cholesky_correctness(-1.0 * mat, L)
    time_cholesky_solve = time.time()
    x_cholesky = solve_using_cholesky(L, -1.0 * b)
    time_cholesky += time.time() - time_cholesky_solve
    print("Time taken for Cholesky factorization and Solve = {}
s".format(time_cholesky))
    verify_solution(mat, x_cholesky, b)
```

```python
    # Solve using np.linalg.solve
    print("Solve using numpy.linalg.solve")
    time_np = time.time()
    x = solve_using_np_solve(mat, b)
    time_np = time.time() - time_np
    print("Time taken for Numpy factorization and Solve = {} s".format(time_np))
    verify_solution(mat, x, b)

if __name__ == '__main__':
    m = 4000
    run_experiment(m)
```