

# Numerical Linear Algebra Homework Project 3: Eigenvalues and Eigenvectors

Catalano Giuseppe, Cerrato Nunzia

## Problem 1

(1) We want to reduce the symmetric matrix  $A$

$$A = \begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix} \quad (1)$$

to a tridiagonal form using Housholder similarity transformations. The matrix  $A$  can be expressed in the following, compact way:

$$A = \left[ \begin{array}{c|c} a_{11} & \mathbf{x}^T \\ \hline \mathbf{x} & \hat{A}_1 \end{array} \right]. \quad (2)$$

Let us define the vector  $\mathbf{u}_1$

$$\mathbf{u}_1 = \mathbf{x} + \text{sgn}(x_1)\|\mathbf{x}\|_2 \mathbf{e}_1 \quad (3)$$

thanks to which we can define the matrix  $\hat{R}_1$

$$\hat{R}_1 = \mathbb{1}_3 - \frac{2}{\|\mathbf{u}_1\|_2^2} \mathbf{u}_1 \mathbf{u}_1^T \quad (4)$$

and a new matrix  $R_1$ , as follows:

$$R_1 = \left[ \begin{array}{c|c} 1 & \mathbf{0}^T \\ \hline \mathbf{0} & \hat{R}_1 \end{array} \right]. \quad (5)$$

We will not explicitly construct the matrix  $R_1$ , as we know that the product  $R_1 A$  will assume the following form:

$$R_1 A = \left[ \begin{array}{c|c} a_{11} & \mathbf{x}^T \\ \hline -\text{sgn}(x_1)\|\mathbf{x}\|_2 & \hat{R}_1 \hat{A}_1 \\ 0 & \\ 0 & \end{array} \right], \quad (6)$$

where the first column is obtained by construction and the matrix  $\hat{R}_1 \hat{A}_1$  can be obtained by considering the matrix  $\hat{A}_1$  written by columns as  $\hat{A}_1 = [(\hat{A}_1)_1, (\hat{A}_1)_2, (\hat{A}_1)_3]$ . Hence  $\hat{R}_1 \hat{A}_1 = [(\hat{R}_1 \hat{A}_1)_1, (\hat{R}_1 \hat{A}_1)_2, (\hat{R}_1 \hat{A}_1)_3]$ , where we know that a single column can be computed as:

$$(\hat{R}_1 \hat{A}_1)_i = (\hat{A}_1)_i - \frac{2}{\|\mathbf{u}_1\|_2^2} \mathbf{u}_1 \mathbf{u}_1^T (\hat{A}_1)_i. \quad (7)$$

By expliciting all the terms, we obtain:

$$R_1 A = \begin{bmatrix} 4 & -1 & -1 & 0 \\ \sqrt{2} & -2\sqrt{2} & -2\sqrt{2} & \sqrt{2} \\ 0 & -2\sqrt{2} & 2\sqrt{2} & 0 \\ 0 & -1 & -1 & 4 \end{bmatrix}. \quad (8)$$

Since we want to obtain a tridiagonal matrix, we have to compute the matrix  $R_1 A R_1$ , namely:

$$R_1 A R_1 = \left[ \begin{array}{c|ccc} a_{11} & \|\mathbf{x}\|_2 & 0 & 0 \\ \hline \|\mathbf{x}\|_2 & & & \\ 0 & & \hat{R}_1 \hat{A}_1 \hat{R}_1 & \\ 0 & & & \end{array} \right]. \quad (9)$$

Knowing that  $\hat{R}_1 \hat{A}_1 \hat{R}_1 = (\hat{R}_1 \hat{A}_1 \hat{R}_1)^T = (\hat{R}_1)^T (\hat{R}_1 \hat{A}_1)^T = \hat{R}_1 (\hat{R}_1 \hat{A}_1)^T$ , we can apply  $\hat{R}_1$  to  $(\hat{R}_1 \hat{A}_1)^T$  by columns:

$$\left[ \hat{R}_1 \hat{A}_1 \hat{R}_1 \right]_i = \left[ (\hat{R}_1 \hat{A}_1)^T \right]_i - \frac{2}{\|\mathbf{u}_1\|_2^2} \mathbf{u}_1 \mathbf{u}_1^T \left[ (\hat{R}_1 \hat{A}_1)^T \right]_i, \quad (10)$$

obtaining the following form for the matrix  $R_1 A R_1$ :

$$R_1 A R_1 = \begin{bmatrix} 4 & \sqrt{2} & 0 & 0 \\ \sqrt{2} & 4 & 0 & \sqrt{2} \\ 0 & 0 & 4 & 0 \\ 0 & \sqrt{2} & 0 & 4 \end{bmatrix}. \quad (11)$$

By identifying the submatrix  $\hat{R}_1 \hat{A}_1 \hat{R}_1$  as

$$\hat{R}_1 \hat{A}_1 \hat{R}_1 = \begin{bmatrix} 4 & 0 & \sqrt{2} \\ 0 & 4 & 0 \\ \sqrt{2} & 0 & 4 \end{bmatrix} = \begin{bmatrix} (\hat{R}_1 \hat{A}_1 \hat{R}_1)_{11} & \mathbf{y}^T \\ \mathbf{y} & \hat{A}_2 \end{bmatrix}, \quad (12)$$

we can build the vector  $\mathbf{u}_2$  as

$$\mathbf{u}_2 = \mathbf{y} + \text{sgn}(y_1) \|\mathbf{y}\|_2 \mathbf{e}_1^{(2)}, \quad (13)$$

with  $\mathbf{e}_1^{(2)} = (1, 0)^T$ , which will allow us to define the matrix  $\hat{R}_2$

$$\hat{R}_2 = \mathbb{1}_2 - \frac{2}{\|\mathbf{u}_2\|_2^2} \mathbf{u}_2 \mathbf{u}_2^T. \quad (14)$$

Using the same logic, we obtain the  $i$ -th column of the matrix  $\hat{R}_2 \hat{A}_2$ , namely:

$$(\hat{R}_2 \hat{A}_2)_i = (\hat{A}_2)_i - \frac{2}{\|\mathbf{u}_2\|_2^2} \mathbf{u}_2 \mathbf{u}_2^T (\hat{A}_2)_i, \quad (15)$$

and, by expliciting all the terms we obtain:

$$\hat{R}_2 \hat{A}_2 = \begin{bmatrix} 0 & -4 \\ -4 & 0 \end{bmatrix}. \quad (16)$$

As before, we can compute  $\hat{R}_2 \hat{A}_2 \hat{R}_2$  by columns, and the final form of the matrix will be:

$$R_2 R_1 A R_1 R_2 = \begin{bmatrix} 4 & \sqrt{2} & 0 & 0 \\ \sqrt{2} & 4 & -\sqrt{2} & 0 \\ 0 & -\sqrt{2} & 4 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}. \quad (17)$$

As we can see, we have reduced the initial matrix  $A$  to a tridiagonal form.

(2) In the following we report the script where we implement the QR diagonalization of a matrix  $A$ , defined as:

$$A = \begin{bmatrix} 4 & 3 & 2 & 1 \\ 3 & 4 & 3 & 2 \\ 2 & 3 & 4 & 3 \\ 1 & 2 & 3 & 4 \end{bmatrix}. \quad (18)$$

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # Set the initial values
5  tol = 1e-5
6  counter = 0
7  t = 1 # Initial exponent for the tolerance
8
9  # Set the number of significant digits
10 np.set_printoptions(precision=15, suppress=True)
11
12 # Construct the matrix A
13 A = np.array([[4,3,2,1],[3,4,3,2],[2,3,4,3],[1,2,3,4]])
14 # Compute the exact eigenvalues
15 exact_eigenvalues = np.linalg.eigvals(A)
16
17 # Initialize an empty list to store the number of iterations at each t
18 t_counter_list = []
19
20 # Cycle until the stopping criterion is not satisfied
21 while np.amax(np.abs(A - np.diag(np.diag(A)))) >= tol:
22     # Obtain the QR factorization of A and perform the QR iteration
23     Q, R = np.linalg.qr(A)
24     A = R@Q
25
26     # Obtain the current approximation of the eigenvalues
27     computed_eigvals = np.diag(A)
28
29     # Compute and store the maximum absolute error on the eigenvalues
30     abs_error = max(np.abs(exact_eigenvalues - computed_eigvals))
31
32     counter += 1
33
34     if abs_error < 10**(-t):
35         t_counter_list.append(counter)
36         t += 1
37
38     if counter in [1,5,10,15]:
39         print(f'k={counter}')
40         print(A)
41         print('-----')
42 print(f'Final k = {counter}')
43 print(A)

```

```

44 computed_eigvals = np.diag(A)
45
46 print('-----')
47 print(f'Exact eigenvalues = {format(exact_eigenvalues)}')
48 print(f'Computed eigenvalues = {format(computed_eigvals)}')
49 diff_eigvals = np.abs(exact_eigenvalues - computed_eigvals)
50 print(f'Absolute error = {format(diff_eigvals)}')

```

Here we report the intermediate  $A_k$  matrices for  $k = 1, 10, 15, 15$  and the final one with  $k = 21$  and the exact and computed eigenvalues, with the absolute errors.

```

k=1
[[ 9.733333333333334  2.834948829502994  0.878364340955178 -0.231869447880085]
 [ 2.834948829502991  3.783908045977011  1.539932953851869 -0.60279905751826 ]
 [ 0.878364340955178  1.539932953851869  1.515016685205784 -0.509164636937078]
 [-0.231869447880084 -0.602799057518259 -0.509164636937078  0.967741935483871]]
-----
k=5
[[11.098895703904493  0.030845371708295  0.000040119600318 -0.000003286456335]
 [ 0.030845371708295  3.414318666854397  0.006815571252927 -0.000792986818634]
 [ 0.000040119600319  0.006815571252927  0.884533164554776 -0.070136316449804]
 [-0.000003286456334 -0.000792986818633 -0.070136316449804  0.602252464686331]]
-----
k=10
[[11.099019512653443  0.000084962498578  0.00000000014034  0.00000000001383]
 [ 0.000084962498578  3.414213563282151  0.000008722995681  0.000000121056837]
 [ 0.00000000014034  0.000008722995681  0.900746172772326  0.008590655965716]
 [ 0.00000000001382  0.000000121056836  0.008590655965716  0.586020751292075]]
-----
k=15
[[11.099019513592774  0.000000234022501  0. -0.000000000000001]
 [ 0.000000234022501  3.414213562373102  0.000000011163285 -0.000000000018006]
 [ 0. 0.000000011163285  0.900977321539928 -0.000998767899781]
 [-0. -0.000000000018005 -0.000998767899781  0.58578602494192]]
-----
Final k = 26
[[11.099019513592783  0.0000000000000546 -0. 0.000000000000001]
 [ 0.0000000000000546  3.414213562373094  0.000000000000005 0.000000000000001]
 [ 0. 0.000000000000005  0.900980486163497 0.000008764600757]
 [ 0. 0. 0.000008764600758 0.585786437870623]]
-----
Exact eigenvalues =
[11.099019513592786  3.414213562373094  0.900980486407215  0.585786437626905]
Computed eigenvalues =
[11.099019513592783  3.414213562373094  0.900980486163497  0.585786437870623]
Absolute error =
[0.0000000000000004 0. 0.000000000243718 0.000000000243717]

```

We can observe the decreasing of the off-diagonal entries and the convergence of the diagonal terms to the eigenvalues computed using the function `np.linalg.eigvals`, which we consider as exact. The maximum absolute error on the eigenvalues is of the order of  $10^{-8}$ .

In the following script we implement the QR diagonalization of the matrix  $A$  using the Rayleigh quotient shift and deflation.

```

1  # Initialize values
2  t = 9
3  tol_list = 10.**(-np.array(range(1,t)))
4  counter_list = []
5
6  # Construct the matrix A
7  A = np.array([[4,3,2,1],[3,4,3,2],[2,3,4,3],[1,2,3,4]])
8  # Compute the exact eigenvalues
9  exact_eigenvalues = np.linalg.eigvals(A)
10 exact_eigenvalues_copy = exact_eigenvalues.copy()
11
12 # Cycle on the list of tolerances
13 for tol in tol_list:
14     counter = 0
15     A = np.array([[4,3,2,1],[3,4,3,2],[2,3,4,3],[1,2,3,4]])
16     exact_eigenvalues = exact_eigenvalues_copy.copy()
17     dim = np.shape(A)[0]
18
19     # Perform QR iteration until the matrix is reduced to a single value by
20     ↪ deflation
21     while dim > 0:
22         approx_lambda = A[dim-1,dim-1]
23         eig_error = abs(exact_eigenvalues-approx_lambda)
24         # If the error on an eigenvalue is less than the tolerance, perform
25         ↪ deflation
26         if min(eig_error) < tol:
27             index = np.argmin(eig_error)
28             exact_eigenvalues = np.delete(exact_eigenvalues,index)
29             A = A[:dim-1,:dim-1]
30             dim = dim - 1
31             continue
32
33     # Compute the shift and perform QR iteration
34     shift = A[dim-1,dim-1]
35     Q, R = np.linalg.qr(A - shift * np.eye(dim))
36     A = R@Q + shift * np.eye(dim)
37     counter = counter + 1
38
39 counter_list.append(counter)

```

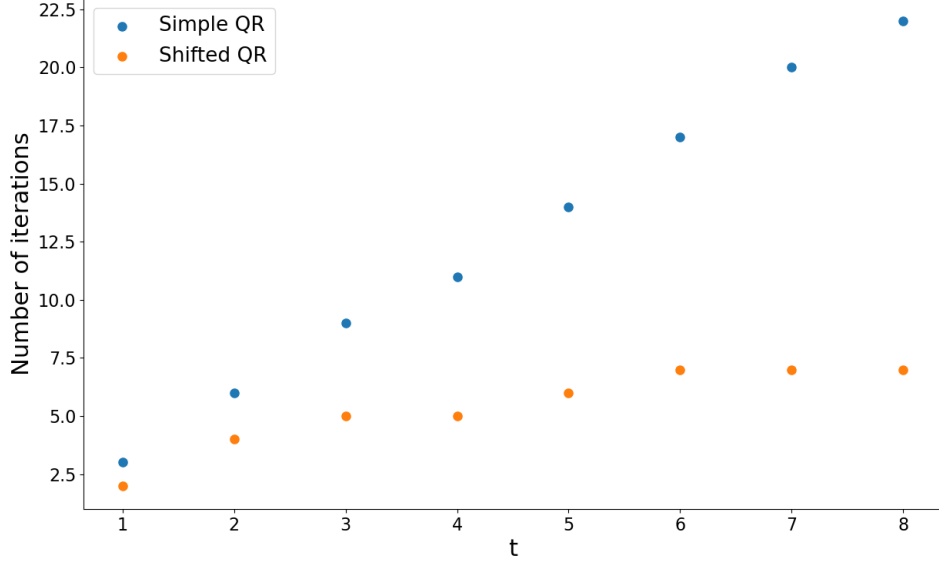


Figure 1: Number of iterations needed to achieve a maximum absolute error less than  $10^{-t}$  on the eigenvalues computed by using standard and shifted QR iteration algorithm.

As we see from Fig. 1 the number of iterations required to achieve a maximum absolute error less than  $10^{-t}$  on eigenvalues appears to grow linearly with  $t$  in the case of the simple QR iteration algorithm. On the other hand, in the case of the shifted QR iteration algorithm, implemented by using the Rayleigh shift, the same error appears to grow sublinearly with  $t$ , meaning that this last algorithm reaches the required tolerance with a smaller number of iterations, thus being faster.

## Problem 2

We want to find approximations to the eigenvalues and eigenfunctions of the one-dimensional Laplace operator  $L[u] := -\frac{d^2u}{dx^2}$  on the unit interval  $[0, 1]$  with boundary conditions  $u(0) = u(1) = 0$ . Let us define as  $\lambda_i$  an eigenvalue of  $L$  and as  $u_i$  the corresponding eigenfunction. Approximations to the eigenvalues and eigenfunctions can be obtained by discretizing the interval  $[0, 1]$  through  $N + 2$  evenly spaced points:  $x_i = ih$  where  $i = 0, 1, \dots, N + 1$  and  $h = 1/(N + 1)$ . The second derivative operator can then be approximated by centered finite differences:

$$-\frac{d^2u}{dx^2}(x_i) \approx \frac{-u(x_{i-1}) + 2u(x_i) - u(x_{i+1}))}{h^2} \quad (19)$$

and therefore the continuous (differential) eigenproblem can be approximated by the discrete (algebraic) eigenvalue problem

$$h^{-2}T_N \mathbf{u} = \lambda \mathbf{u}, \quad (20)$$

where we have set

$$T_N = \begin{bmatrix} 2 & -1 & & 0 \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ 0 & & -1 & 2 \end{bmatrix}, \text{ and } \mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \end{bmatrix}, \quad (21)$$

with  $u_i := u(x_i)$ . Recall that the eigenvalues and eigenfunctions of  $L$  are  $\lambda_j = j^2\pi^2$  and  $u_j(x) = \alpha \sin(j\pi x)$  for any nonzero constant  $\alpha$ , which we can take to be 1, and  $j$  positive integer. On the other hand, the  $N \times N$  matrix  $T_N$  has eigenvalues  $\mu_j = 2(1 - \cos \frac{\pi j}{N+1})$  for  $j = 1, \dots, N$ , corresponding to the eigenvectors  $\mathbf{u}_j$ , where  $\mathbf{u}_j(k) = \sqrt{\frac{2}{N+1}} \sin(\frac{j k \pi}{N+1})$  is the  $k$ th entry in  $\mathbf{u}_j$ .

(1) Since we are considering  $j \ll N$  and  $N \gg 1$  we can consider the Taylor expansion of  $\cos(\frac{\pi j}{N+1})$ , which leads us to approximate the smallest eigenvalues of  $h^{-2}T_N$  as follows:

$$2h^{-2} \left(1 - \cos \frac{\pi j}{N+1}\right) = 2h^{-2} \left(\frac{(\pi j h)^2}{2!} - \frac{(\pi j h)^4}{4!} + O(h^6)\right) = \pi^2 j^2 \left(1 - \frac{(\pi j h)^2}{12} + O(h^4)\right), \quad (22)$$

where we used that  $h = 1/N + 1$ .

For the largest eigenvalue of  $T_N$ , we have that  $j = N$ , therefore we can not truncate anymore the Taylor expansion of  $\cos \frac{\pi j}{N+1}$  if we want a good approximation. We can compute the  $N$ -th eigenvalue of  $T_N$  in the limit of  $N \gg 1$  (namely  $h \ll 1$ ) with the following manipulations:

$$\mu_N = 2 \left(1 - \cos \frac{\pi N}{N+1}\right) = 2(1 - \cos(\pi - \pi h)) = 2(1 + \cos \pi h) = 4 - \pi^2 h^2 + O(h^4). \quad (23)$$

Therefore, we have

$$h^{-2}\mu_N = \frac{4}{h^2} - \pi^2 + O(h^2) = 4(N+1)^2 - \pi^2 + O(N^{-2}), \quad (24)$$

which is not a good approximation of  $\lambda_N = \pi^2 N^2$ .

(2) We want to compare the eigenvectors  $\mathbf{u}_j$  of  $T_N$  with the eigenfunctions of  $L$ , up to the normalization constant, that we will set to 1 for both. If we recall that  $x_k = kh \forall k = 1, \dots, N$ , we can observe that the  $k$ -th component of the eigenvector  $\mathbf{u}_j$  is equal to the  $j$ -th eigenfunction  $u_j(x)$  computed in correspondence of the value  $x = x_k$ :

$$u_j(x_k) = \sin(j\pi x_k) = \sin(j\pi kh) = \sin\left(\frac{j\pi k}{N+1}\right) = \mathbf{u}_j(k). \quad (25)$$

(3) Now we compute the spectral condition number of  $T_N$ , defined as  $k_2(T_N) = \frac{h^{-2}\mu_N}{h^{-2}\mu_1}$ , in the limit of  $N \gg 1$ . We recall that the eigenvalues of  $T_N$  are

$$\mu_j = 2 \left(1 - \cos \frac{\pi j}{N+1}\right) = 2(1 - \cos \pi j h). \quad (26)$$

By considering the Taylor expansion of the cosine, we can compute the numerator and the denominator of  $k_2(T_N)$ . In particular, to compute the numerator we can use the expression reported in Eq. (22), where  $j = 1$ , while for the denominator we use the expression reported in Eq. (23).

$$k_2(T_N) = \frac{h^{-2}\mu_N}{h^{-2}\mu_1} = \frac{4h^{-2} - \pi^2 + \frac{1}{12}\pi^4 h^2 + O(h^4)}{\pi^2(1 - \frac{1}{12}\pi^2 h^2 + O(h^4))}. \quad (27)$$

Here, in the denominator, we can use the expansion  $(1+x)^\alpha = 1 + \alpha x + O(x^2)$ , finding  $(1 - \frac{1}{12}\pi^2 h^2 + O(h^4))^{-1} = 1 + \frac{1}{12}\pi^2 h^2 + O(h^4)$ . Therefore, we find

$$\begin{aligned} k_2(T_N) &= \left(\frac{4h^{-2}}{\pi^2} - 1 + \frac{1}{12}\pi^4 h^2 + O(h^4)\right) \left(1 + \frac{1}{12}\pi^2 h^2 + O(h^4)\right) \\ &= \frac{4h^{-2}}{\pi^2} - 1 + \frac{1}{12}\pi^4 h^2 + \frac{4h^{-2}}{\pi^2} \frac{1}{12}\pi^2 h^2 - \frac{1}{12}\pi^2 h^2 + O(h^4) \\ &= \frac{4h^{-2}}{\pi^2} - \frac{2}{3} + O(h^2) = \frac{4(N+1)^2}{\pi^2} - \frac{2}{3} + O(N^{-2}). \end{aligned} \quad (28)$$

(4-5) Here we report the plot of the eigenvalues and eigenvectors of  $T_N$ , with  $N = 21$ .

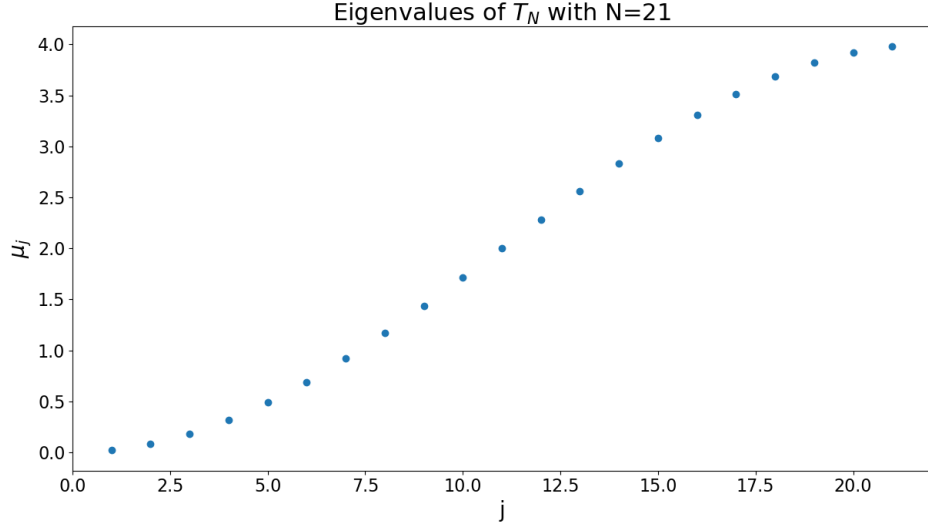


Figure 2: Eigenvalues of  $T_N$ , with  $N = 21$ .

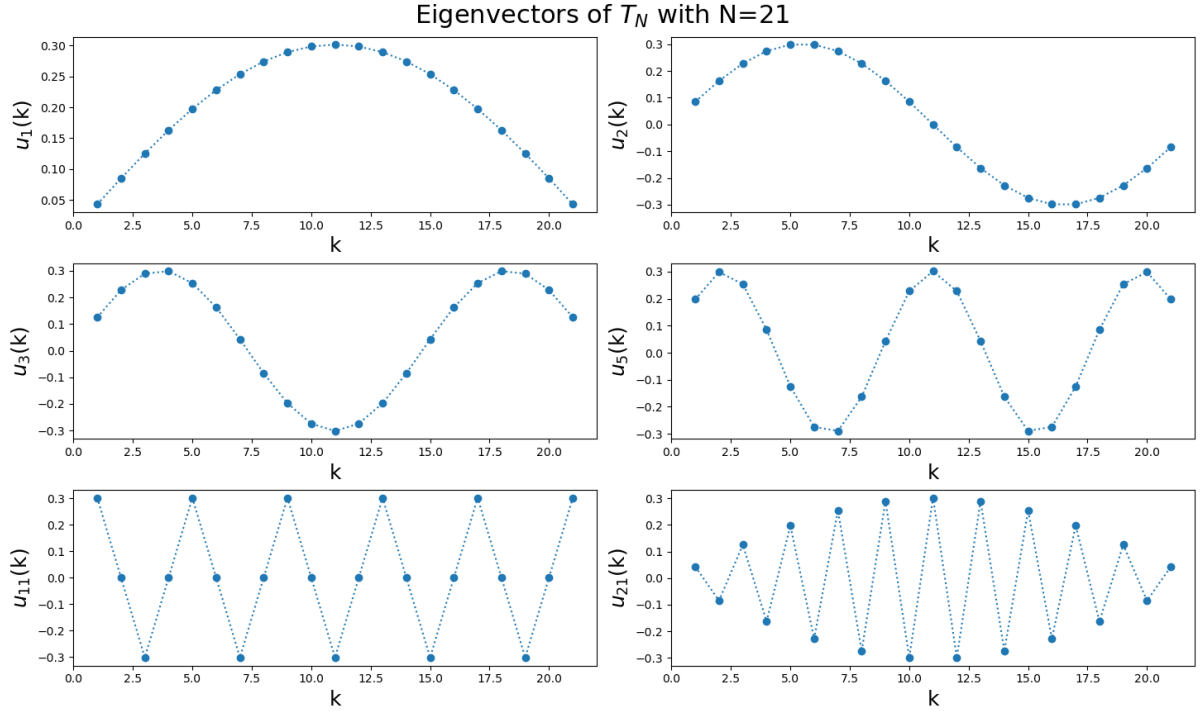


Figure 3: Eigenvectors of  $T_N$  for different values of  $j$ , with  $j$  being the number that identifies the  $j$ -th eigenvector.

(6) In the following we report the script implementing the inverse power method to find the smallest eigenvalue of  $h^{-2}T_N$ .

```

1 import numpy as np
2 import scipy
3
4 # Set initial parameters

```



```

5  N = 500
6  tol = 1e-8
7  initial_guess = np.random.random(N)
8  initial_guess = initial_guess/np.linalg.norm(initial_guess, ord=2)
9  count = 0
10 diff = 10 * tol
11
12 # Build T_N
13 T_N = scipy.sparse.diags([-1,2,-1],[-1,0,1],shape=(N,N)).toarray()
14
15 # Cholesky factorization of h-2T_N
16 L, low = scipy.linalg.cho_factor(T_N)
17 L = (N+1) * L
18
19 vect_old = initial_guess
20
21 # Cycle until the difference in the 2-norm between two successive
↪ (approximated) eigenvectors is less than the chosen tolerance
22 while diff >= tol:
23     # Compute and normalize the new vector by using the Choleski factorization
↪ of T_N
24     vect_new = scipy.linalg.cho_solve((L, low),vect_old)
25     vect_new = vect_new/np.linalg.norm(vect_new, ord=2)
26     diff = np.linalg.norm(vect_new - vect_old, ord=2)
27     count += 1
28     vect_old = vect_new
29
30 # Compute the approximated eigenvalue
31 approx_eig = vect_new @ T_N @ vect_new * (N+1)**2
32
33 print(f'Iterations performed = {count}')
34 exact_eigval_T_N = 2 * (N+1)**2 * (1-np.cos(np.pi/(N+1)))
35 print(f'Absolute error eigenvalue) = {abs(approx_eig - exact_eigval_T_N)}')
36 index = np.array(range(1,N+1))
37 exact_eigvect = np.sqrt(2/(N+1)) * np.sin(index*np.pi/(N+1))
38 print(f'2-norm error eigenvector = {np.linalg.norm(vect_new - exact_eigvect,
    ↪ ord=2)}')

```

The output of this code is the following:

```

Iterations performed = 10
Absolute error eigenvalue = 1.531041959879076e-11
2-norm error eigenvector = 3.2570943203952573e-09

```

As we can see, 10 iterations are required to obtain the chosen tolerance on the computed eigenvector. Finally, we report the plot of the computed eigenvector.

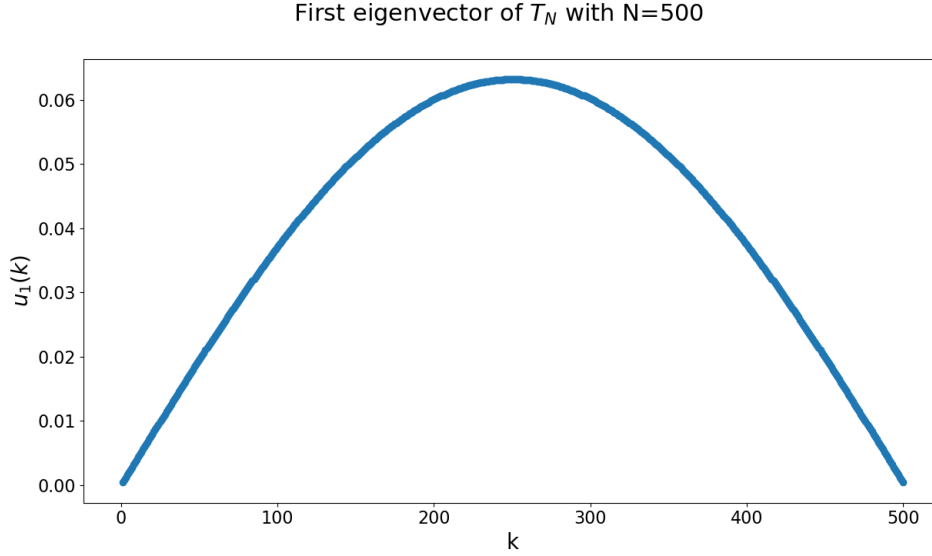


Figure 4: First eigenvector of  $T_N$ , with  $N = 500$  computed by using the inverse power method.

(7) Below we report the script implementing the shift-and-invert method to find the fifth eigenvalue of  $h^{-2}T_N$ . In this case, it is not possible to use the Cholesky factorization since the matrix  $h^{-2}T_N - 25\pi^2\mathbb{1}_N$  is not positive-definite.

```

1  import numpy as np
2  import scipy
3
4  # Set initial parameters
5  N = 500
6  lambda_5 = 25*np.pi**2
7  initial_guess = np.random.random(N)
8  initial_guess = initial_guess/np.linalg.norm(initial_guess, ord=2)
9  count = 0
10
11 # Build T_N
12 T_N = scipy.sparse.diags([-1,2,-1],[-1,0,1],shape=(N,N)).toarray()
13
14 tol = 1e-12 * np.linalg.norm((N+1)**2 * T_N, ord = np.inf)
15 diff = 10 * tol
16
17 # LU factorization of h^(-2)T_N
18 lu, piv = scipy.linalg.lu_factor(T_N*(N+1)**2 - lambda_5 * np.eye(N))
19
20 vect_old = initial_guess
21
22 # Cycle until the stopping criterion is satisfied - condition on the
23   ↪ residual
24 while diff >= tol:
25     # Compute and normalize the new vector by using the LU factorization to
26     ↪ solve a linear system
27     vect_new = scipy.linalg.lu_solve((lu, piv), vect_old)
28     vect_new = vect_new/np.linalg.norm(vect_new, ord=2)

```

```

27     # Compute the approximated eigenvalue
28     approx_eig = vect_new @ T_N @ vect_new * (N+1)**2
29     diff = np.linalg.norm(( T_N * (N+1)**2 - approx_eig * np.eye(N) ) @
    ↪ vect_new, ord=np.inf)
30     count += 1
31     vect_old = vect_new
32
33     print(f'Iterations performed = {count}')
34     exact_eigval_T_N = 2 * (N+1)**2 * (1-np.cos(5*np.pi/(N+1)))
35     print(f'Absolute error eigenvalue = {abs(approx_eig - exact_eigval_T_N)}')
36     index = np.array(range(1,N+1))
37     exact_eigvect = np.sqrt(2/(N+1)) * np.sin(index*np.pi*5/(N+1))
38     print(f'2-norm error eigenvector = {np.linalg.norm(vect_new - exact_eigvect,
    ↪ ord=2)}')

```

The output of this code is the following:

```

Iterations performed = 2
Absolute error eigenvalue = 1.0032863428932615e-11
2-norm error eigenvector = 5.021569614559994e-08

```

In this case, 2 iterations are required to obtain an accurate approximation of the fifth eigenvector. Finally, we report the plot of the computed eigenvector.

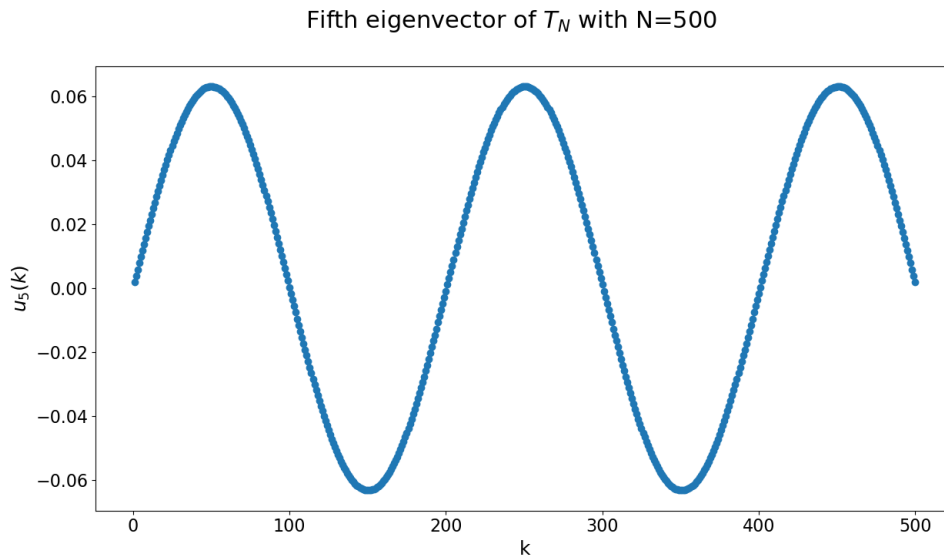


Figure 5: Fifth eigenvector of  $T_N$ , with  $N = 500$  computed by using the inverse shift-and-invert algorithm.