Numerical Analysis and Optimization Project 3

```julia
using LinearAlgebra, Plots
```

Problem 1

Task 1

The first Householder transformation needed to reduce the matrix

$$A = \begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix} \equiv [\mathbf{a}^{(1)}, \mathbf{a}^{(2)}, \mathbf{a}^{(3)}, \mathbf{a}^{(4)}]$$

to its triagonal form has to set to zero the term a_{13}.

Thus, we form the first Householder vector in the usual way ($\mathbf{x} \equiv \mathbf{a}^{(1)}_{2:4}$):

$$\hat{\mathbf{u}}_1 = \mathbf{x} + \mathrm{sgn}(x_1)||\mathbf{x}||_2\, e_1 = \begin{bmatrix} -1 \\ -1 \\ 0 \end{bmatrix} - \sqrt{2} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Then, as usual, we can construct the first Householder reflection $\hat{R}_1$ as the operation on a vector $\mathbf{v}$ such that

$$\hat{R}_1 \mathbf{v} = \mathbf{v} - 2\frac{(\hat{\mathbf{u}}_1^T \mathbf{v})}{||\hat{\mathbf{u}}_1||_2^2}\hat{\mathbf{u}}_1$$

Thus we can compute

$$\hat{R}_1 \mathbf{a}^{(1)}_{2:4} = \begin{bmatrix} \sqrt{2} \\ 0 \\ 0 \end{bmatrix}$$

$$\hat{R}_1 \mathbf{a}_{2:4}^{(2)} = \begin{bmatrix} -2\sqrt{2} \\ -2\sqrt{2} \\ -1 \end{bmatrix}$$

$$\hat{R}_1 \mathbf{a}_{2:4}^{(3)} = \begin{bmatrix} -2\sqrt{2} \\ 2\sqrt{2} \\ -1 \end{bmatrix}$$

$$\hat{R}_1 \mathbf{a}_{2:4}^{(4)} = \begin{bmatrix} \sqrt{2} \\ 0 \\ 4 \end{bmatrix}$$

Then

$$R_1 A = \begin{bmatrix} 1 & 0 \\ 0 & \hat{R}_1 \end{bmatrix} A = \begin{bmatrix} 4 & -1 & -1 & 0 \\ \sqrt{2} & -2\sqrt{2} & -2\sqrt{2} & \sqrt{2} \\ 0 & -2\sqrt{2} & 2\sqrt{2} & 0 \\ 0 & -1 & -1 & 4 \end{bmatrix}$$

Now we have to apply R_1 from the right. Since the Householder transformations are symmetric matrices, we note that

$$(R_1 A R_1)^T = R_1^T (R_1 A)^T = R_1 (R_1 A)^T$$

So to right-apply R_1 is equivalent to apply R_1 (and thus $\hat{R}_1$) to the rows of $R_1 A$, which is more efficient.

$$\hat{R}_1 \begin{bmatrix} -1 \\ -1 \\ 0 \end{bmatrix} = \begin{bmatrix} \sqrt{2} \\ 0 \\ 0 \end{bmatrix}$$

$$\hat{R}_1 \begin{bmatrix} -2\sqrt{2} \\ -2\sqrt{2} \\ \sqrt{2} \end{bmatrix} = \begin{bmatrix} 4 \\ 0 \\ \sqrt{2} \end{bmatrix}$$

$$\hat{R}_1 \begin{bmatrix} -2\sqrt{2} \\ 2\sqrt{2} \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \\ 0 \end{bmatrix}$$

$$\hat{R}_1 \begin{bmatrix} -1 \\ -1 \\ 4 \end{bmatrix} = \begin{bmatrix} \sqrt{2} \\ 0 \\ 4 \end{bmatrix}$$

So, at the end of the first Householder similarity transformation, we get

$$R_1 A R_1 = \begin{bmatrix} 4 & \sqrt{2} & 0 & 0 \\ \sqrt{2} & 4 & 0 & \sqrt{2} \\ 0 & 0 & 4 & 0 \\ 0 & \sqrt{2} & 0 & 4 \end{bmatrix} \equiv A_1$$

We proceed with the second transformation to set to zero the element $4, 2$ of A_1. In this case the Householder vector is

$$\hat{\mathbf{u}}_2 = \begin{bmatrix} 0 \\ \sqrt{2} \end{bmatrix} + \sqrt{2} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \Longrightarrow \mathbf{u}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

and the tranformation applied to a vector $\mathbf{v}$ is

$$\hat{R}_2 \mathbf{v} = \mathbf{v} - 2(\mathbf{u}_2^T \mathbf{v})\mathbf{u}_2 = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} - (v_1 + v_2) \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -v_2 \\ -v_1 \end{bmatrix}$$

So the transformation is a swap of elements of the vector plus a change of sign.

We get then

$$R_2 A_1 = \begin{bmatrix} I_{2\times 2} & 0 \\ 0 & \hat{R}_2 \end{bmatrix} A_1 = \begin{bmatrix} 4 & \sqrt{2} & 0 & 0 \\ \sqrt{2} & 4 & 0 & \sqrt{2} \\ 0 & -\sqrt{2} & 0 & -4 \\ 0 & 0 & -4 & 0 \end{bmatrix}$$

And finally, the tridiagonal form.

$$R_2 A_1 R_2 = \begin{bmatrix} 4 & \sqrt{2} & 0 & 0 \\ \sqrt{2} & 4 & -\sqrt{2} & 0 \\ 0 & -\sqrt{2} & 4 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

Task 2

Subtask a

Here we present a simple QR iteration implementation. The matrix A_k is updated by computing its QR decomposition, $A_k = QR$ and then setting $A_{k+1} = RQ$. This is done until all the nondiagonal elements of A_k have an absolute values smaller than a given threshold.

The function can store some of the intermediary A_k.

```
qr_iteration (generic function with 1 method)
```

```julia
function qr_iteration(A; tol=1e-4, store=(1,5,10,15), maxitr=1000)
    Aₖ = Matrix{Float64}(A)
    d = ["Start" => copy(Aₖ)]
    i = 0
    while maximum(abs.(Aₖ - Diagonal(Aₖ))) >= tol && i <= maxitr
        i += 1
        Q, R = qr(Aₖ)
        Aₖ = R * Q
        if any(store .== i) || store == :all
            push!(d, "Itr $i" => copy(Aₖ))
        end
    end
    push!(d, "Itr $i (End)" => copy(Aₖ))
    return d
end
```

We apply our algorithm on the matrix

```
A = 4×4 Matrix{Int64}:
     4  3  2  1
     3  4  3  2
     2  3  4  3
     1  2  3  4
```

Here we can see the evolution A_k during the process:

```
Aₖ =
▼Pair{String, Matrix{Float64}}[
    1:       "Start" ⇒ 4×4 Matrix{Float64}:
                        4.0  3.0  2.0  1.0
                        3.0  4.0  3.0  2.0
                        2.0  3.0  4.0  3.0
                        1.0  2.0  3.0  4.0
    2:       "Itr 1" ⇒ 4×4 Matrix{Float64}:
                         9.73333    2.83495    0.878364  -0.231869
                         2.83495    3.78391    1.53993   -0.602799
                         0.878364   1.53993    1.51502   -0.509165
                        -0.231869  -0.602799  -0.509165   0.967742
    3:       "Itr 5" ⇒ 4×4 Matrix{Float64}:
                        11.0989       0.0308454     4.01196e-5   -3.28646e-6
                         0.0308454    3.41432       0.00681557   -0.000792987
                         4.01196e-5   0.00681557    0.884533     -0.0701363
                        -3.28646e-6  -0.000792987  -0.0701363     0.602252
    4:       "Itr 10" ⇒ 4×4 Matrix{Float64}:
                         11.099        8.49625e-5  1.40342e-10  1.38293e-12
                          8.49625e-5   3.41421     8.723e-6     1.21057e-7
                          1.4034e-10   8.723e-6    0.900746     0.00859066
                          1.38196e-12  1.21057e-7  0.00859066   0.586021
    5:       "Itr 15" ⇒ 4×4 Matrix{Float64}:
                         11.099         2.34023e-7    2.01592e-15  -9.39967e-16
                          2.34023e-7    3.41421       1.11633e-8   -1.80041e-11
                          4.94643e-16   1.11633e-8    0.900977     -0.000998768
                         -5.66144e-19  -1.80049e-11  -0.000998768   0.58579
    6:       "Itr 21 (End)" ⇒ 4×4 Matrix{Float64}:
                               11.099         1.98286e-10   1.52403e-15  -9.3494e-16
                                1.98287e-10   3.41421       3.77039e-12   4.28566e-1
                                1.41539e-22   3.76999e-12   0.90098      -7.54419e-5
                               -1.22365e-26  -4.59289e-16  -7.54419e-5    0.585786
]
```

```julia
Aₖ = qr_iteration(A)
```

Subtask b

Here we compare the eigenvalues found by the standard Julia's library LinearAlgebra (which in this context we consider the *true* eigenvalues of the matrix A). . .

```
@show eigen_true = ▶[0.585786, 0.90098, 3.41421, 11.099]
```

```julia
@show eigen_true = eigvals(A)
```

```
eigen_true = eigvals(A) = [0.5857864376269052, 0.9009804864072157, 3.414
2135623730954, 11.099019513592786]
```

. . . with the eigenvalues found by our simple QR iteration algorithm:

```
@show eigen_qr = ▶[0.585786, 0.90098, 3.41421, 11.099]
```

```julia
@show eigen_qr = sort(diag(Aₖ[end].second))
```

```
eigen_qr = sort(diag((Aₖ[end]).second)) = [0.5857864556839626, 0.9009804
683501574, 3.414213562373097, 11.099019513592786]
```

Here we see that the absolute difference between the "true" eigenvalues and ours is of the order of 10^{-8} in the worst case.

```
[1.80571e-8, 1.80571e-8, 1.77636e-15, 0.0]
```

```julia
abs.(eigen_true - eigen_qr)
```

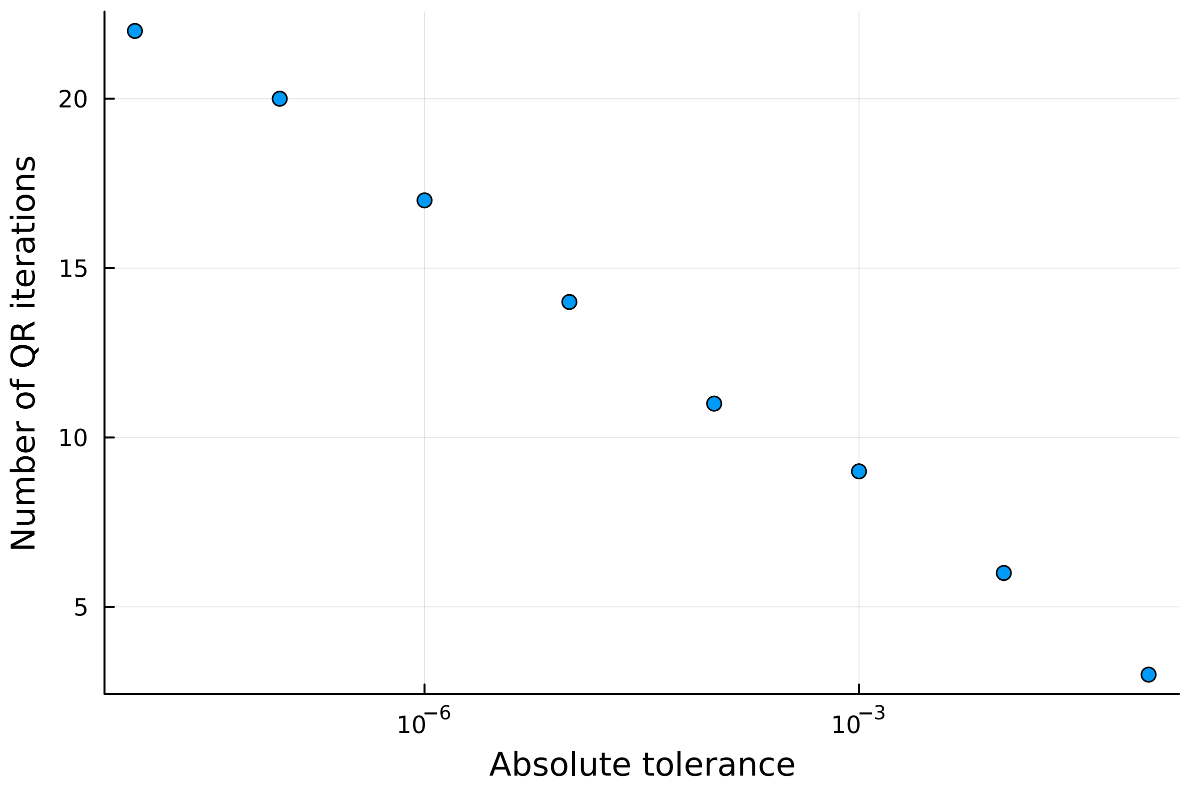
Subtask c

Now we perform the same qr iteration as before, but we change the stopping criteria. Instead of setting a threshold on the absolute value of the nondiagonal elements, we set a threshold on the absolute difference between the eigenvalues found by the default Julia `eigvals()` method and the values found by our implementation of the QR iteration.

```
qr_comparison (generic function with 1 method)
```

```julia
function qr_comparison(A; tol=1e-4, maxitr=1000)
    Aₖ = Matrix{Float64}(A)
    eigen_true = eigvals(A)
    i = 0
    while any(j -> j >= tol, abs.(sort(diag(Aₖ)) - eigen_true)) && i <= maxitr
        i += 1
        Q, R = qr(Aₖ)
        Aₖ = R * Q
    end
    return i
end
```

In the plot below we can see how many iteration the algorithm must perform in order to achieve a given precision (w.r.t. the values found with the standard method). The increase in precision is roughly exponential with the number of QR iterations.



```julia
begin
    x = [1e-1,1e-2,1e-3,1e-4,1e-5,1e-6,1e-7,1e-8]
    y = zero(x)
    for (i,t) in enumerate(x)
        y[i] = qr_comparison(A, tol=t)
    end
    p = plot(x,y, xscale=:log10, xlabel="Absolute tolerance", ylabel="Number of
    QR iterations", marker=true, line=false, legend=false)
end
```

Subtask d

Now we proceed with the (Rayleigh) shifted QR iteration with deflation.

The implemetation is organized as follows: at each iteration, we perform a single shifted QR iteration with a Rayleigh shift, that is, a QR decomposition for the matrix $A_k - \mu_k I = QR$, where μ_k is the last diagonal element of A_k, then the matrix A_k is updated such that $A_{k+1} = RQ + \mu_k I$.

After each update, we perform the following checks:

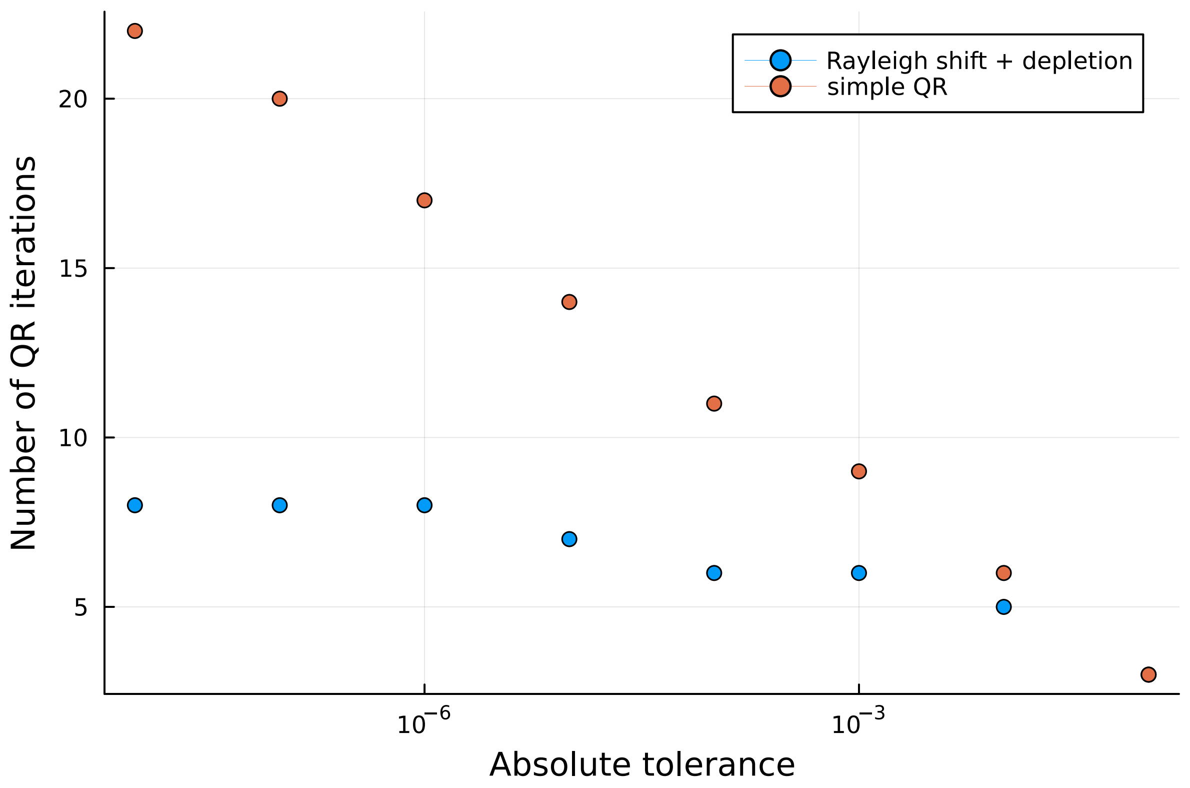
- if all the elements of the last column and the last row (diagonal element excluded) have an absolute value smaller than a given threshold, all those element are set to zero (*deflation*);
- if the last diagonal element is close to one of the "true" eigenvalues (that is, the absolute difference between them is less than a given threshold) then the eigenvalue is removed from the list of the eigenvalues to be found and the shifted QR iteration is then performed on the reduced version of A_k without the last column and row.

The loop ends when the size of A_k is 1×1, which means that the eigenvalue search is done.

```
qr_deflated_rayleigh (generic function with 1 method)
```

```julia
function qr_deflated_rayleigh(A; tol=1e-4, deflation_tol=1e-4, maxitr=1000)
    Aₖ = Matrix{Float64}(A)
    eigen_true = eigvals(A)
    i = 0
    while i <= maxitr
        i += 1
        μₖ = Aₖ[end, end]
        Q, R = qr(Aₖ - μₖ * one(Aₖ))
        Aₖ = R * Q + μₖ * one(Aₖ)
        # deflation
        if all(abs.(vcat(Aₖ[end,1:end-1], Aₖ[1:end-1,end])) .<= deflation_tol)
            Aₖ[end,1:end-1] .= 0
            Aₖ[1:end-1,end] .= 0
        end
        # matrix/"eigenlist" reduction
        if any(abs.(Aₖ[end:end] .- eigen_true) .<= tol)
            eigen_true = eigen_true[abs.(Aₖ[end:end] .- eigen_true) .>= tol]
            Aₖ = Aₖ[1:end-1,1:end-1]
            if size(Aₖ) == (1,1)
                break
            end
        end
    end
    return i
end
```

Here we plot the difference between the number of iteration needed with the depleted/shifted method and the simple method. We see that the former converges faster than the latter.



```
begin
    yrd = zero(x)
    for (i,t) in enumerate(x)
        yrd[i] = qr_deflated_rayleigh(A, tol=t)
    end
    p2 = plot(x,yrd, xscale=:log10, xlabel="Absolute tolerance", ylabel="Number
    of QR iterations", marker=true, line=false, label="Rayleigh shift +
    depletion")
    plot!(p2, x, y, marker=true, line=false, label="simple QR")
end
```

Problem 2

Task 1

We start from the expression:

$$\mu_j = 2h^{-2}(1 - \cos(\frac{\pi j}{N+1}))$$

For $j/N \to 0$ we can use the Taylor expansion for the cosine around 0, and we get:

$$\begin{aligned}\mu_j &\approx 2h^{-2}(1 - (1 - \frac{1}{2}(\frac{\pi j}{N+1})^2 + O(h^4(\pi j)^4)) = \\ &= h^{-2}(\frac{\pi j}{N+1})^2 + O(h^2(\pi j)^4) = \\ &\approx (\pi j)^2 + O(h^2(\pi j)^4)\end{aligned}$$

Where we have used the fact that $h = 1/(N+1)$. This is precisely the expression for the smallest eigenvalues of the L operator, up to an $O(h^2)$ error.

For large eigenvalues, $j \to N$, $j/N \to 1$. N is still large. We can still use a Taylor expansion for the cosine, but this time around π:

$$\begin{aligned}\mu_j &\approx 2h^{-2}(1-(-1+\frac{1}{2}(\frac{\pi N}{N+1}-\pi)^2) = \\ &= \dots = \\ &= h^{-2}[4-\frac{\pi^2}{(N+1)^2}] \approx \\ &\approx 4(N+1)^2\end{aligned}$$

Where the part in square brackets is the eigenvalue approximation for T_N.

Task 2

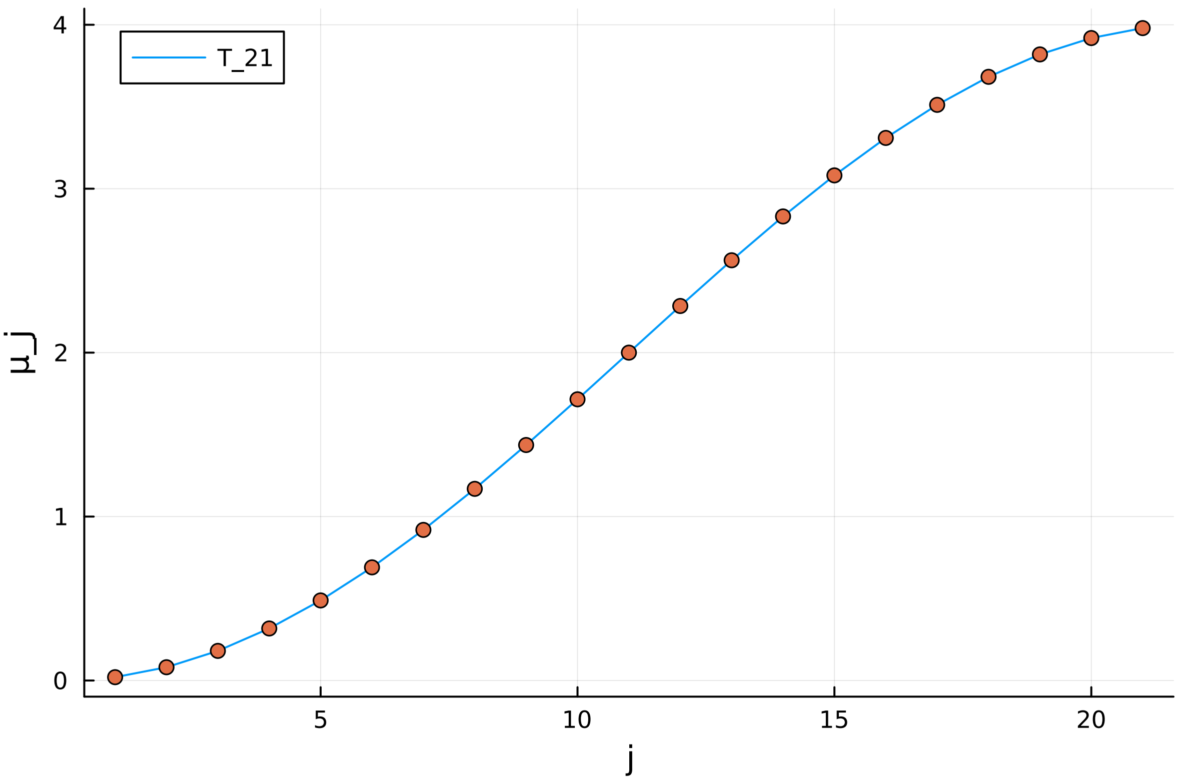
It can be noticed from their expression that $\mathbf{u}_j$ are precisely the exact eignefunctions of the L operator evaluated in $x = \frac{k}{N+1}$, up to a normalization factor.

Task 3

In the limit $N \to \inf$, we can use the expressions we found in the previous points for the smallest and biggest eigenvalues. The spectral condition number is thus:

$$\begin{aligned}\kappa &= \lambda_N/\lambda_1 \\ &\approx (4-\frac{\pi^2}{(N+1)^2})/(\frac{\pi^2}{(N+1)^2}) = \\ &= 4\frac{(N+1)^2}{\pi^2}-1\end{aligned}$$

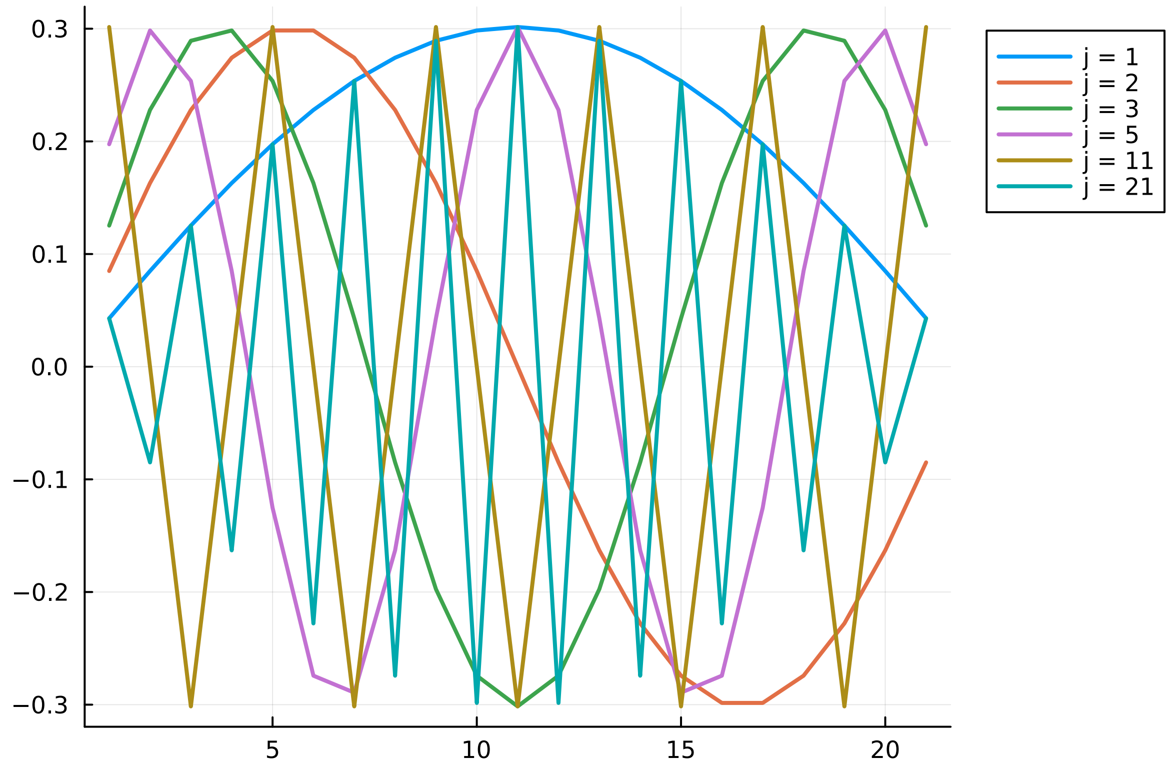
Task 4



```julia
begin
    N = 21
    j = 1:N
    μ = 2 * (1 .- cos.(π * j / (N + 1)))

    p3 = plot(j, μ, xlabel="j", ylabel="μ_j", label="T_21")
    plot!(p3, j, μ, seriestype=:scatter, label="")
end
```

Task 5



```julia
begin
    u(k, j) = sqrt(2/(N+1)) * sin(π * j * k / (N+1))
    j1 = [1,2,3,5,11,21]
    eigenvectors = [u(k, j) for k in 1:N, j in j1]

    # Initialize the plot with the first series to ensure it creates a plot object
    plo = plot(1:N, eigenvectors[:, 1], label="j = $(j1[1])",
    legend=:outertopright, figsize=(1000, 600),  linewidth=2)
    # Add the rest of the series in a loop
    for idx in 2:length(j1)
        plot!(plo, 1:N, eigenvectors[:, idx], label="j = $(j1[idx])",
    figsize=(1000, 600), linewidth=2)
    end

    plo
end
```

Task 6

In the implementation of the power method we use the Julia's backslash operator which performs a forward-backward substitution if the first argument is a Cholesky object.

```
inverse_power_method (generic function with 1 method)
```

```julia
function inverse_power_method(T; tol=1e-8, maxitr=1000)
    x = normalize(rand(size(T,1)))
    λ = [0.]
    C = cholesky(T)
    i = 1
    while i <= maxitr
        y = C \ x
        if norm(x - y / norm(y, 2), 2) < tol
            break
        end
        x = normalize(y)
        r = x' * T * x
        push!(λ, r)
        i += 1
    end
    return λ, x
end
```

We compute then

```julia
begin
    N1 = 500
    T_N = zeros(N1, N1)
    # Fill the diagonal with 2s
    for i in 1:N1
        T_N[i, i] = 2
    end
    # Fill the off-diagonals with -1s
    for i in 1:N1-1
        T_N[i, i+1] = -1
        T_N[i+1, i] = -1
    end
    # Since the exercise involves the operator h^(-2)*T_N, we calculate h
    h = 1 / (N1 + 1)
    # Adjust T_N accordingly
    T_N = (h^(-2)) * T_N
    R = cholesky(T_N).U
    nothing
end
```

The matrix T_N is then

```
500×500 Matrix{Float64}:
  502002.0  -251001.0        0.0        0.0  …        0.0        0.0        0.0
 -251001.0   502002.0  -251001.0        0.0           0.0        0.0        0.0
       0.0  -251001.0   502002.0  -251001.0           0.0        0.0        0.0
       0.0        0.0  -251001.0   502002.0           0.0        0.0        0.0
       0.0        0.0        0.0  -251001.0           0.0        0.0        0.0
       0.0        0.0        0.0        0.0  …        0.0        0.0        0.0
       0.0        0.0        0.0        0.0           0.0        0.0        0.0
       ⋮                                     ⋱
       0.0        0.0        0.0        0.0           0.0        0.0        0.0
       0.0        0.0        0.0        0.0  …        0.0        0.0        0.0
       0.0        0.0        0.0        0.0    -251001.0        0.0        0.0
       0.0        0.0        0.0        0.0     502002.0  -251001.0        0.0
       0.0        0.0        0.0        0.0    -251001.0   502002.0  -251001.0
       0.0        0.0        0.0        0.0          0.0  -251001.0   502002.0
```

```
T_N
```

While R is

```
500×500 UpperTriangular{Float64, Matrix{Float64}}:
 708.521  -354.26     0.0       0.0     …     0.0       0.0       0.0       0.0
    ⋅      613.597  -409.065    0.0           0.0       0.0       0.0       0.0
    ⋅        ⋅       578.505  -433.879        0.0       0.0       0.0       0.0
    ⋅        ⋅         ⋅       560.135        0.0       0.0       0.0       0.0
    ⋅        ⋅         ⋅         ⋅            0.0       0.0       0.0       0.0
    ⋅        ⋅         ⋅         ⋅      …     0.0       0.0       0.0       0.0
    ⋅        ⋅         ⋅         ⋅            0.0       0.0       0.0       0.0
   ⋮                                    ⋱
    ⋅        ⋅         ⋅         ⋅            0.0       0.0       0.0       0.0
    ⋅        ⋅         ⋅         ⋅      …  -500.496     0.0       0.0       0.0
    ⋅        ⋅         ⋅         ⋅          501.504  -500.497     0.0       0.0
    ⋅        ⋅         ⋅         ⋅             ⋅      501.503  -500.498     0.0
    ⋅        ⋅         ⋅         ⋅             ⋅         ⋅      501.502  -500.499
    ⋅        ⋅         ⋅         ⋅             ⋅         ⋅         ⋅      501.501
```

```
R
```

We store the known value for λ_1

```
λ₁ = 9.869604401089358
```

```
λ₁ = pi^2
```

. . . and we apply the inverse power method to find it:

```
▾ (
      1: ▸ [0.0, 10.0016, 9.87087, 9.86959, 9.86957, 9.86957, 9.86957, 9.86957, 9.86957,
      2: ▸ [0.000396192, 0.000792368, 0.00118851, 0.00158461, 0.00198065, 0.00237661, 0.
)
```

```
lambda, eigenvector = inverse_power_method(T_N)
```

We can check the last value returned for the eigenvalue, and the number of steps taken before convergence:

```
9.869572060923938
```

```
lambda[end]
```

▸ (12)

```
size(lambda)
```

Here we compare the difference between the true eigenvalue and the one found with the inverse power method:

```
diff = 3.2340165420308153e-5
```

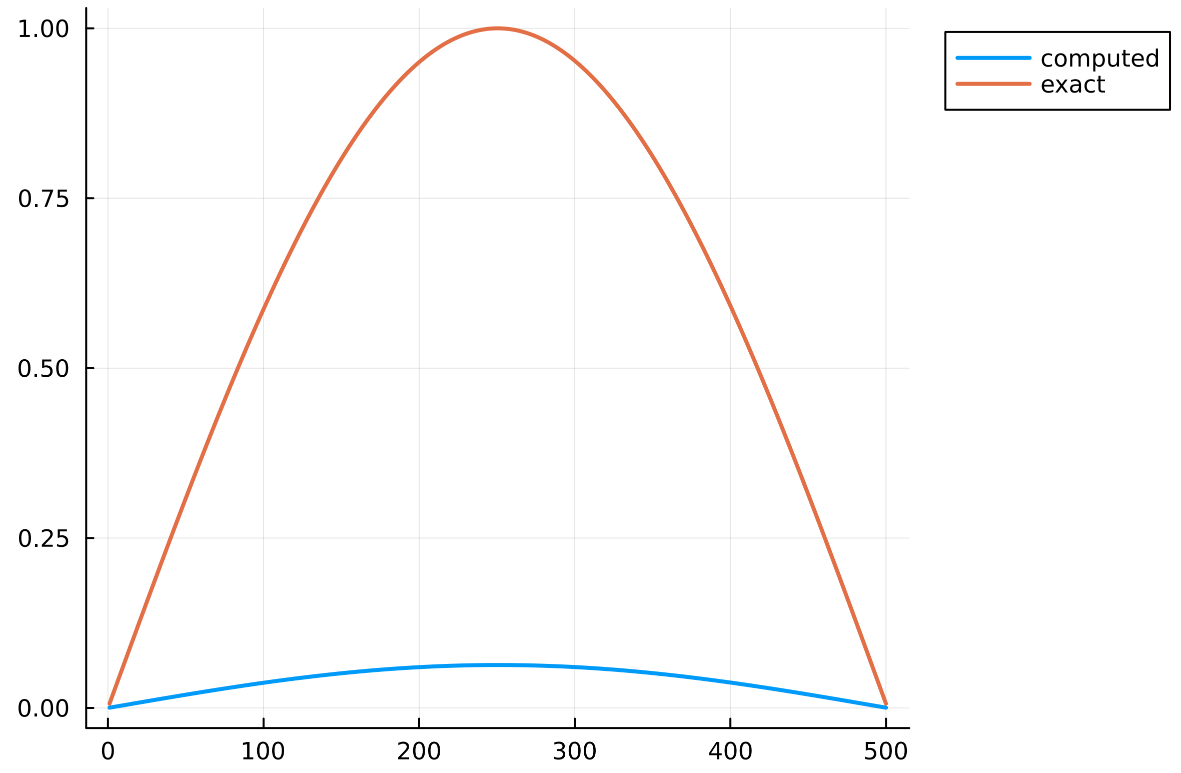
```
diff = abs(lambda[end] - λ₁)
```

. . . and the eigenvector:

```
diff2 = 14.827191791344415
```

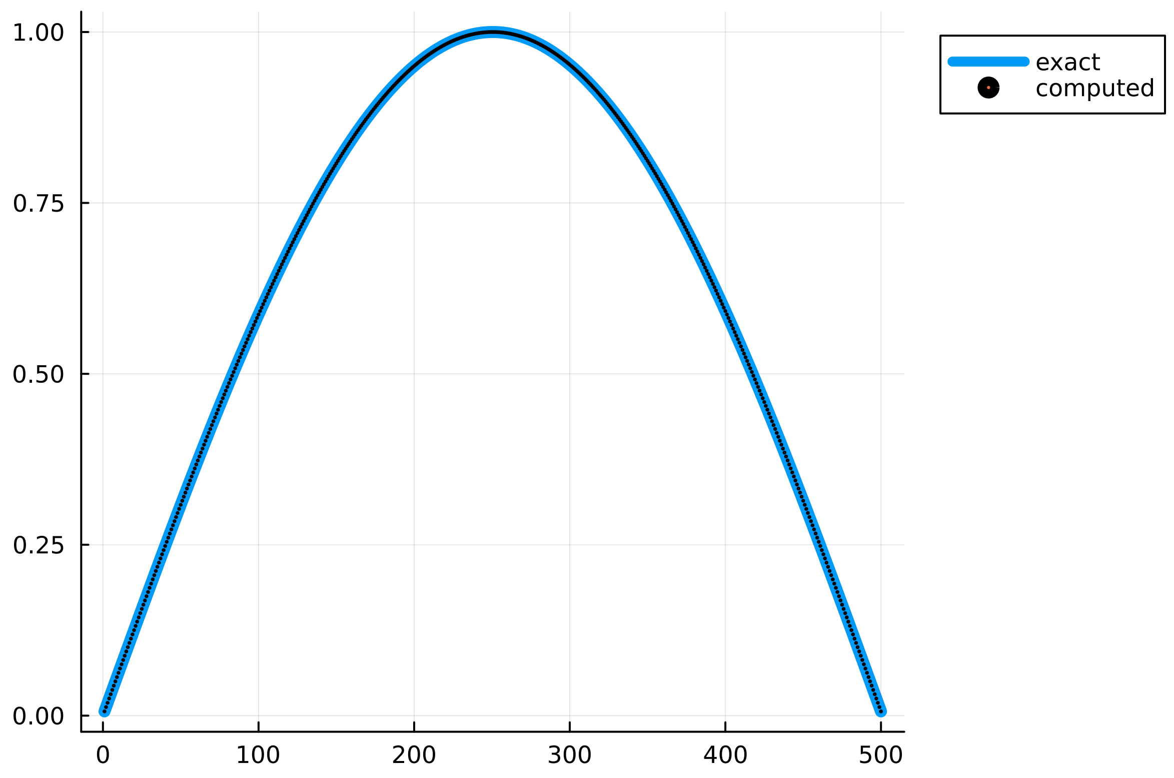
```
diff2 = norm(eigenvector - sin.(π * (1:N1) / (N1 + 1)), 2)
```

Here we plot the computed eigenvector against the real one:



```
begin
    plot(1:N1,eigenvector, label="computed", legend=:outertopright,
    figsize=(1000, 600),  linewidth=2)
    plot!(1:N1, sin.(π * (1:N1) / (N1 + 1)), label="exact",  linewidth=2)
end
```

The computed eigenvector is off, but just because of the normalization inside of the algorithm (the exact eigenvector is not normalized to 1). If we rescale by the norm of the exact eigenvector we get:



```
begin
    test_norm = norm(sin.(π * (1:N1) / (N1 + 1)),2)

    plot(1:N1, sin.(π * (1:N1) / (N1 + 1)), label="exact",  linewidth=6)
    plot!(1:N1,test_norm*eigenvector, label="computed", legend=:outertopright,
    figsize=(1000, 600),  linewidth=2, seriestype=:scatter, markersize=0.6)
end
```

And the computed difference is now:

```
diff3 = 9.450806953544148e-8
```

```
diff3 = norm(test_norm*eigenvector - sin.(π * (1:N1) / (N1 + 1)), 2)
```

Task 7

The Julia backslash operator performs a forward-backward substitution if the first argument is a LU decomposition as well.

```
shift_and_invert_power_method (generic function with 1 method)
```

```julia
function shift_and_invert_power_method(T, shift; tol=1e-12, maxitr=1000)
    x = normalize(rand(size(T,1)))
    θ₀=shift
    λ = [θ₀]
    A = (T - θ₀*one(T))
    LU = lu(A)
    i = 1
    while i <= maxitr
        residual = norm((T - λ[i] * one(T)) * x, Inf)
        if residual <= tol * norm(T, Inf)
            break
        end
        y = LU \ x
        x = normalize(y)
        r = x' * T * x
        push!(λ, r)
        i += 1
    end
    return λ, x
end
```

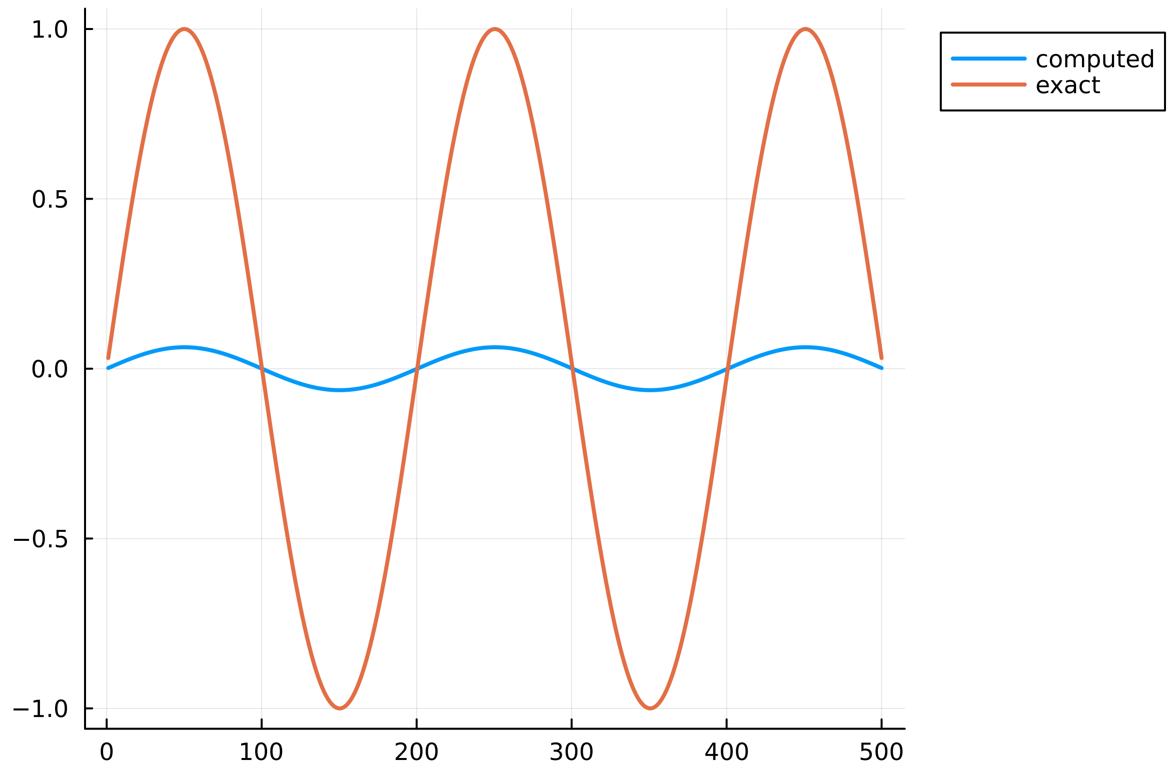
Let's apply the new algorithm. The cholesky decomposition could not be used as A is no longer positive-definite!

```
(
    1:  [247.74, 246.599, 246.72, 246.72, 246.72]
    2:  [0.00198065, 0.00395935, 0.00593416, 0.00790313, 0.00986434, 0.0118159, 0.013
)
```

```julia
lambda5, eigenvector5 = shift_and_invert_power_method(T_N, 25*pi^2+1)
```

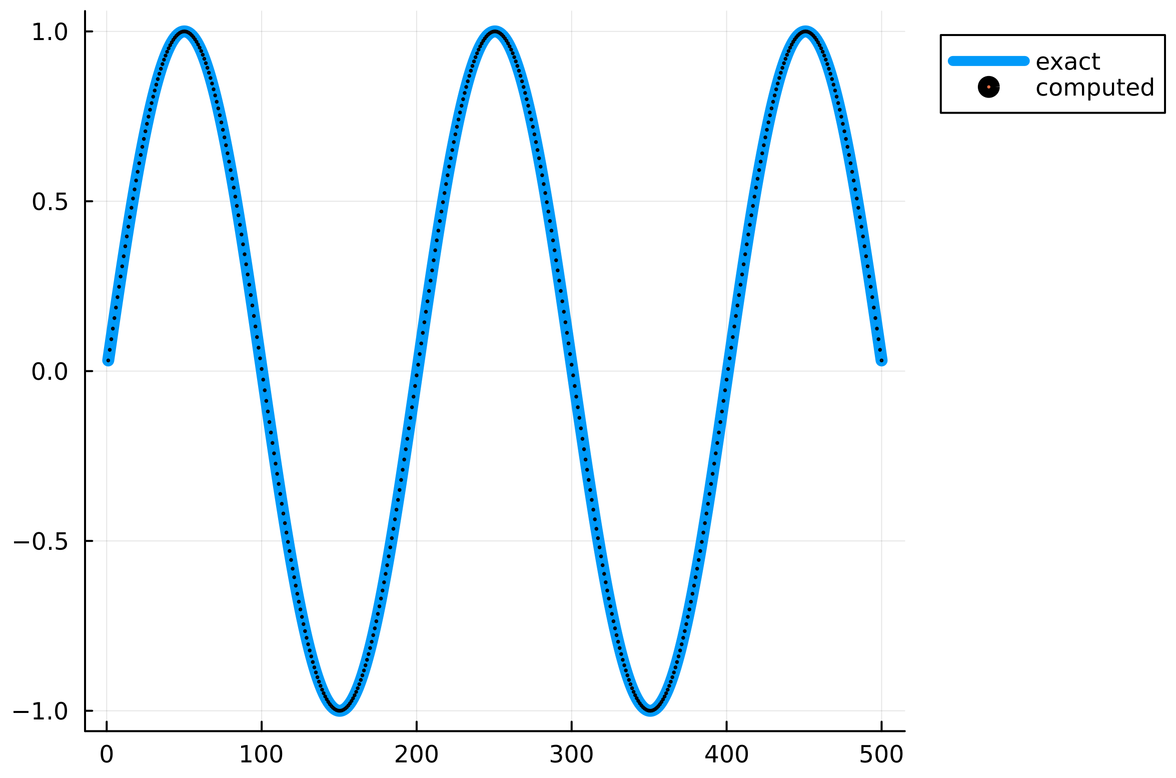
```
(5)
```

```julia
size(lambda5)
```



```julia
begin
    plot(1:N1,eigenvector5, label="computed", legend=:outertopright,
    figsize=(1000, 600),  linewidth=2)
    plot!(1:N1, sin.(5*π * (1:N1) / (N1 + 1)), label="exact",  linewidth=2)
end
```

As said before, we need to rescale the computed eigenvector to ensure we can appreciate the overlap between the computed one and the exact one.



```
begin
    test_norm5 = norm(sin.(5*π * (1:N1) / (N1 + 1)),2)

    plot(1:N1, sin.(5*π * (1:N1) / (N1 + 1)), label="exact",  linewidth=6)
    plot!(1:N1,test_norm5*eigenvector5, label="computed", legend=:outertopright,
    figsize=(1000, 600),  linewidth=2, seriestype=:scatter, markersize=0.6)
end
```

```
diff5 = 5.406786513283991e-8
```

```
diff5 = norm(test_norm5*eigenvector5 - sin.(5*π * (1:N1) / (N1 + 1)), 2)
```