Error control in scientific modelling (MATH 500, Herbst)

Sheet 6: Diagonalisation algorithms

```
1 begin
2     using MatrixDepot
3     using LinearAlgebra
4     using PlutoUI
5     using Printf
6     using Plots
7     using PlutoTeachingTools
8 end
```

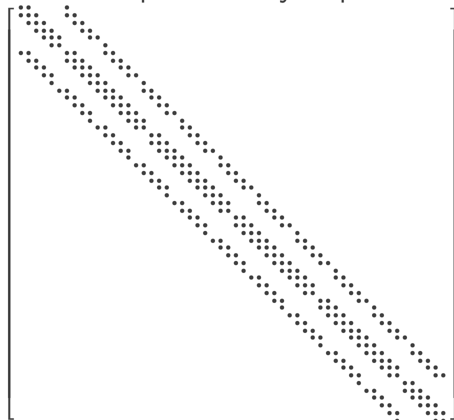
```
verify download of index files...

reading database

recreating database file

reading index files

adding metadata...

adding svd data...

writing database

exception during initialization: 'KeyError(MatrixDepot)'
```

```
EOFError()
```

Exercise 1

In this exercise we want to extend the `projected_subspace_iteration` approach from the lecture in order to numerically compute the eigenpairs closest to the eigenvalue 1 of the matrix

```
A = 100×100 SparseArrays.SparseMatrixCSC{Float64, Int64} with 460 stored entries:
```



```
1 A = matrixdepot("poisson", n)
```

Size of the test matrix: $n =$ 10

which is a sparse matrix resulting from solving a Poisson equation in 2 dimensions. You can assume that the eigenvalue 1 is twice degenerate, i.e. that two eigenvectors are associated with this eigenvalue.

(a) Employ the `projected_subspace_iteration` algorithm of the lectures to numerically compute the the eigenpairs closest to eigenvalue 1. *Hints:* Use spectral transformations; for sparse matrices the `\` operator works as expected.

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

```
1 # split: solution
2 function ortho_qr(X)
3     Matrix(qr(X).Q)
4 end
```

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

```julia
# split: solution
function projected_subspace_inverse_iteration(A; tol=1e-6, maxiter=100, verbose=true,
                                              num_eigvals=2,
                                              V=randn(eltype(A), size(A, 2), 2),
                                              ortho=ortho_qr)
    if num_eigvals > size(V, 2)
        error(
            "The number of requested eigvals exceeds the subspace dimension."
            * "Got num_eigvals=$num_eigvals and size(V, 2)=$(size(V, 2))"
        )
    end

    T = real(eltype(A))
    eigenvalues    = Vector{T}[]
    residual_norms = Vector{T}[]
    λ = T[]
    Y = nothing
    for i in 1:maxiter
        V = ortho(V)

        AV = A \ V  # This is the inverse instead of A * V
        λ, Y = eigen(Hermitian(V' * AV))  # Notice the change to subspace_iteration
                                          # This is the Rayleigh-Ritz step
        # Sort by absolute value and keep only the largest ones.
        perm = sortperm(λ, by=abs)
        idx = perm[end-num_eigvals+1:end]
        λ = λ[idx]
        Y = Y[:, idx]

        push!(eigenvalues, λ)

        residuals = AV * Y - V * Y * Diagonal(λ)
        norm_r = norm.(eachcol(residuals))
        push!(residual_norms, norm_r)

        verbose && @printf "%3i %s %s\n" i λ norm_r
        maximum(norm_r) < tol && break

        V = AV
    end
    X = V * Y

    (; λ, X, eigenvalues, residual_norms)
end
```

```
[1.00777, 1.00777]
```

```
1  # split: solution
2  let
3      σ = 1.0
4      A_shifted = A - σ * I
5      λ_shifted = projected_subspace_inverse_iteration(A_shifted).λ
6      (1 ./ λ_shifted) .+ σ  # transform back
7  end
```

```
  1 [0.4275709654804315, 4.016199084347303] [2.665941014596327, 21.2637605589
07695]
  2 [112.77555275716823, 128.44285983505773] [42.185395046198366, 5.447713120602
536]
  3 [128.6646347164857, 128.67573205666983] [1.1977044392652532, 0.1187443294412
2966]
  4 [128.67583293824376, 128.675843372133] [0.03675124254232565, 0.0033345898316
299283]
  5 [128.67584344961793, 128.67584346023818] [0.0011740257712229607, 0.000100527
39790357849]
  6 [128.67584346031052, 128.6758434603194] [3.544530859260728e-5, 1.41395612927
5713e-5]
  7 [128.67584346031737, 128.67584346032444] [5.45700862459359e-7, 1.11910937326
24265e-6]
  8 [128.6758434603173, 128.67584346032487] [2.1328094514491903e-8, 3.4818007908
098365e-8]
```

(b) Modify your algorithm, such that you can use varying subspace sizes, but only test convergence in the eigenpairs closest to 1. For example use more than two initial guess vectors in `X`, but only check the residual norms for those two eigenpairs corresponding to the eigenvalue 1. Experiment with different subspace sizes between 2 and 5 and plot the observed residual norms wrt. iteration number. Which variant converges in the least number of iterations? If you increase n using the Slider, does this change your assessment? *Bonus: You might notice that for some few intermediate values of n you get qualitatively different behavior than with $n + 1$ or $n - 1$, do you have a hypothesis why?*

```
run_1b (generic function with 1 method)
```

```
1  # split: solution
2  function run_1b(n)
3      @show n
4      A = matrixdepot("poisson", n)
5      σ = 1.0
6      A_shifted = A - σ * I
7
8      max_subspace_dim = 5
9
10     V = randn(size(A, 2), max_subspace_dim)
11     V = ortho_qr(V)
12     results = []
13     for subspace_dim in 2:max_subspace_dim
14         @show subspace_dim
15         res = @time projected_subspace_inverse_iteration(A_shifted; V=V[:,
           1:subspace_dim], verbose=false)
16         push!(results, (subspace_dim, res))
17     end
18     results
19 end
```

results_1b =

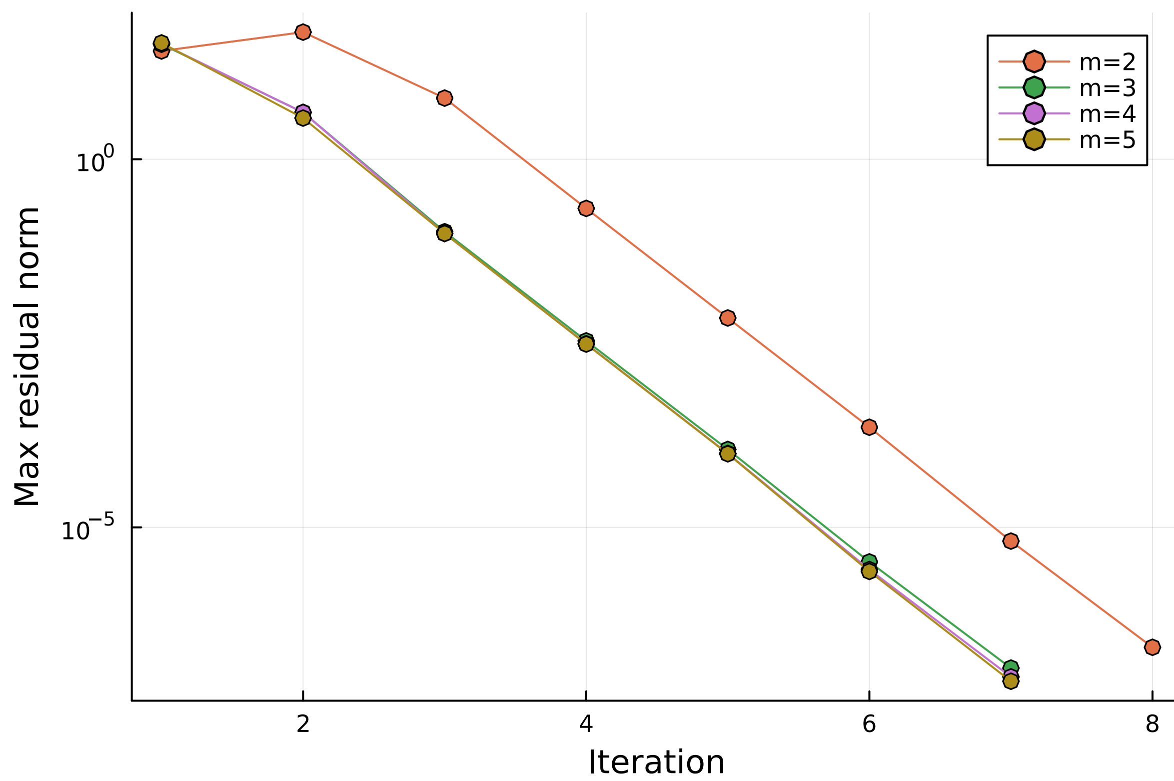
```
Dict(50 ⇒ [(2, (λ = [   more], X = 2500×2 Matrix{Float64}:   , eigenvalues = [   more], residu
                                    0.0216493    -0.00478254
```

```julia
# split: solution
results_1b = Dict(n=>run_1b(n) for n in (10, 20, 30, 50, 100))
```

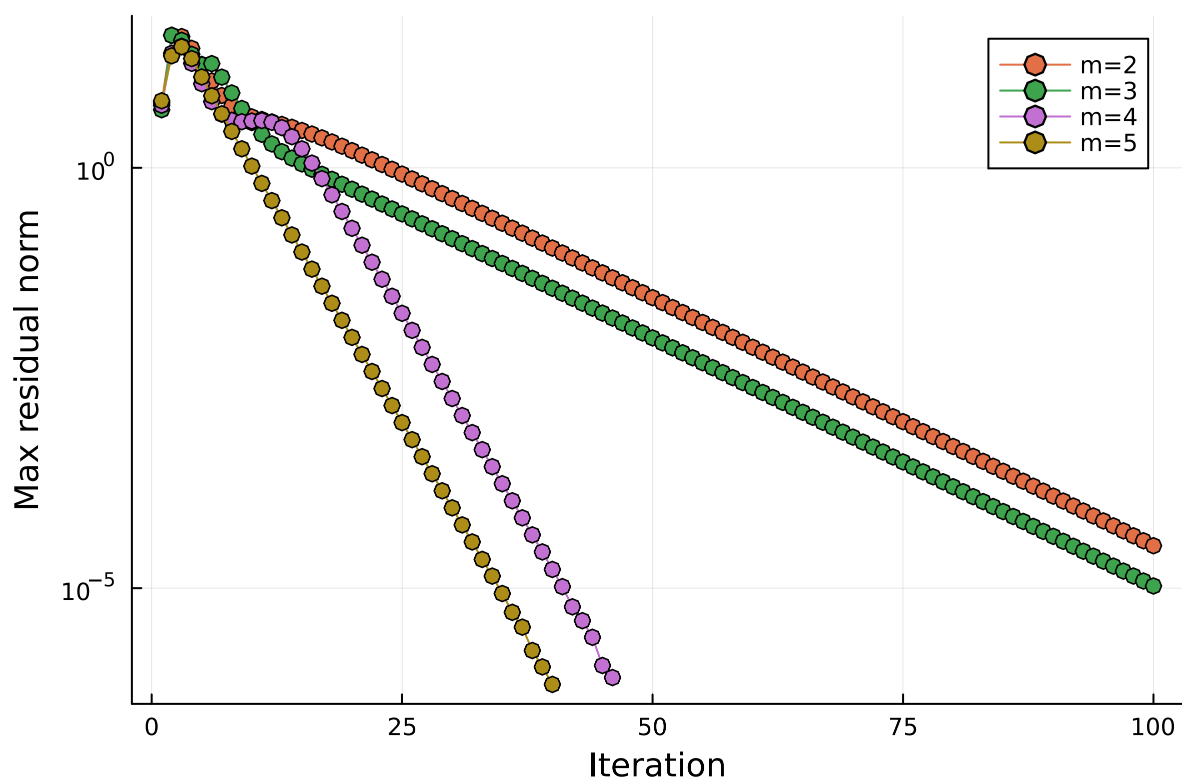
```
n = 10
subspace_dim = 2
  0.014557 seconds (9.34 k allocations: 2.473 MiB, 85.19% compilation time)
subspace_dim = 3
  0.001962 seconds (1.40 k allocations: 1.841 MiB)
subspace_dim = 4
  0.001692 seconds (1.40 k allocations: 1.869 MiB)
subspace_dim = 5
  0.001791 seconds (1.40 k allocations: 1.897 MiB)
n = 20
subspace_dim = 2
  0.023806 seconds (5.41 k allocations: 24.013 MiB)
subspace_dim = 3
  0.035130 seconds (5.18 k allocations: 21.524 MiB, 33.52% gc time)
subspace_dim = 4
  0.016007 seconds (3.54 k allocations: 16.133 MiB)
subspace_dim = 5
  0.014517 seconds (3.12 k allocations: 14.437 MiB)
n = 30
subspace_dim = 2
  0.236612 seconds (21.51 k allocations: 200.267 MiB, 5.49% gc time)
subspace_dim = 3
  0.206690 seconds (21.50 k allocations: 203.026 MiB, 7.01% gc time)
subspace_dim = 4
  0.097551 seconds (9.94 k allocations: 94.718 MiB, 7.94% gc time)
subspace_dim = 5
  0.085575 seconds (8.60 k allocations: 83.509 MiB, 6.70% gc time)
n = 50
subspace_dim = 2
  0.153764 seconds (6.05 k allocations: 160.728 MiB, 7.33% gc time)
subspace_dim = 3
  0.146789 seconds (5.62 k allocations: 151.114 MiB, 7.79% gc time)
subspace_dim = 4
```

plot_1b (generic function with 1 method)

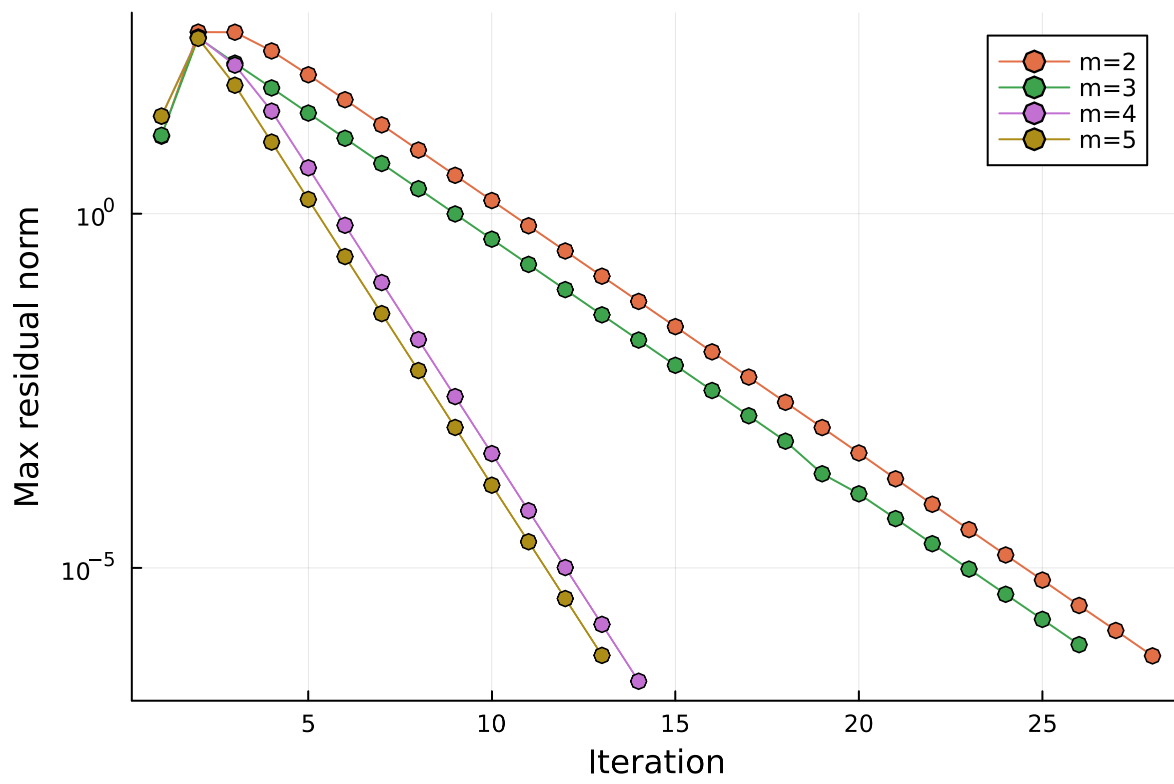
```julia
# split: solution
function plot_1b(results)
    plt = plot(xlabel="Iteration", ylabel="Max residual norm")
    for (subspace_dim, res) in results
        r = maximum.(res.residual_norms)
        plot!(plt, r, label="m=$subspace_dim", yaxis=:log, m=:o, c=subspace_dim)
    end
    plt
end
```



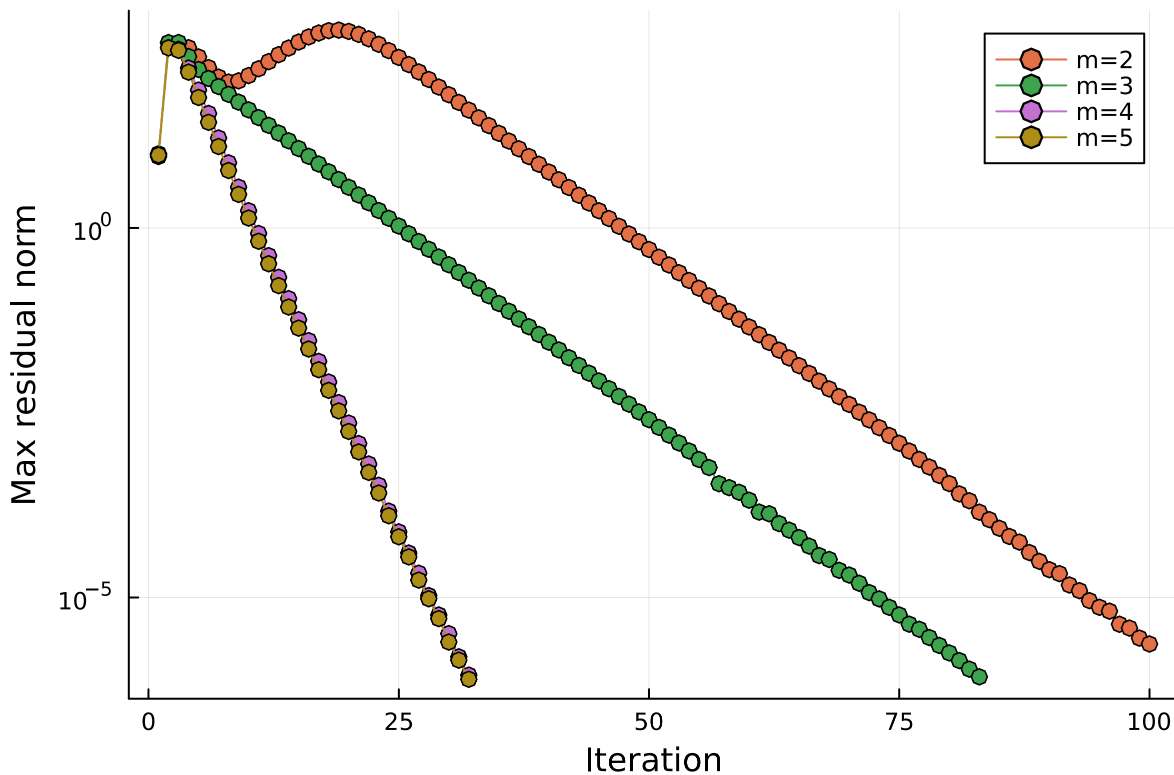

```julia
1 # split: solution
2 plot_1b(results_1b[10])
```




```julia
1 # split: solution
2 plot_1b(results_1b[30])
```




```
1 # split: solution
2 plot_1b(results_1b[50])
```




```
1  # split: solution
2  plot_1b(run_1b(120))
```

```
n = 120
subspace_dim = 2
  3.453666 seconds (21.82 k allocations: 3.366 GiB, 7.58% gc time, 0.15% compila
tion time)
subspace_dim = 3
  2.801160 seconds (18.01 k allocations: 2.829 GiB, 4.04% gc time)
subspace_dim = 4
  1.110043 seconds (6.96 k allocations: 1.105 GiB, 3.75% gc time)
subspace_dim = 5
  1.125217 seconds (7.00 k allocations: 1.119 GiB, 11.55% gc time)
```

(c) Given that the computational time per iteration scales roughly linearly in the number of subspace vectors, what is the most economical configuration for this setting? Measure the runtime of your algorithm in this setting using Julia's `@time` macro. Take the average of multiple measurements to reduce the influence due to the interference of other processes on your computer. *Optional:* If you want automate this, take a look at the BenchmarkTools Julia package.

(d) Modify the original `projected_subspace_iteration` a second time, but in a different way: Extend it, such that it employs in each iteration the optimal *dynamical* shift, just like in Rayleigh quotient interation (RQI). As an initial guess take random vectors and test your algorithm by running it a few times using `A`. Ensure that the resulting eigenpairs are indeed approximate eigenpairs of A.

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

```julia
# split: solution
function projected_subspace_iteration_dynamic(A;
        tol=1e-6, maxiter=100, verbose=true, ortho=ortho_qr,
        V=randn(eltype(A), size(A, 2), 2),
    )
    T = real(eltype(A))

    eigenvalues    = Vector{T}[]
    residual_norms = Vector{T}[]
    λ = T[]
    Y = nothing
    for i in 1:maxiter
        V = ortho(V)

        AV = A * V
        λ, Y = eigen(Hermitian(V' * AV))  # Notice the change to subspace_iteration
                                           # This is the Rayleigh-Ritz step
        push!(eigenvalues, λ)

        residuals = AV * Y - V * Y * Diagonal(λ)
        norm_r = norm.(eachcol(residuals))
        push!(residual_norms, norm_r)

        verbose && @printf "%3i %s %s\n" i λ norm_r
        maximum(norm_r) < tol && break

        Vnew = zero(V)
        for (i, Vi) in enumerate(eachcol(V))
            Vnew[:, i] .= (A - λ[i] * I) \ Vi
        end
        V = Vnew
    end
    X = V * Y

    (; λ, X, eigenvalues, residual_norms)
end
```

(e) Similar to RQI the procedure of (d) converges quickly to an eigenpair, but as you probably saw it is hard to predict which. If we want to employ it for approximating the eigenvalues around 1 we therefore need to already use an initial guess, which is very close to the corresponding eigenvectors we care about. The solution is to chain the algorithms of (a) and (d), i.e. to employ one step of your algorithm in (a) on a random initial guess (e.g. set `maxiter=1`) as the starting point for your procedure in (d). Code up this chained algorithm and time it a few times using the same settings as in (c), i.e. the same subspace size in particular. Is employing the dynamical shift worth it? If you increase n using the Slider, does this change your assessment?

```
([1.00777, 1.00777], 2×2 Matrix{Float64}:        )
                          1.00777      2.46397e-17
```

```
# split: solution
let
    σ = 1.0
    (; X) = projected_subspace_inverse_iteration(A - σ * I; maxiter=1)
    println("Switching to dynamic shift algorithm")
    (; λ, X) = projected_subspace_iteration_dynamic(A; V=X)
    λ, X'A*X
end
```

```
  1 [2.530418586931888, 9.603399031640484] [16.94875800630835, 33.64025861893243]
Switching to dynamic shift algorithm
  1 [1.0078451998393068, 1.0086184257113906] [0.027566813173926754, 0.05767707658227074]
  2 [1.007771465566984, 1.0077714664325141] [2.5194674087840054e-5, 5.185291403273121e-6]
  3 [1.0077714664470672, 1.0077714664470678] [1.497377282442665e-14, 2.9361893712146882e-15]
```

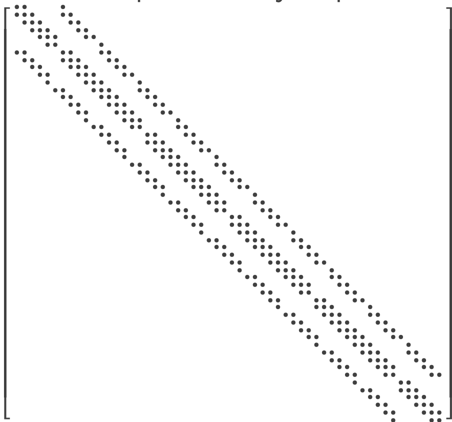
Exercise 2

In this exercise we will discuss some error estimation strategies for orthogonal projection methods.

We follow the Rayleigh-Ritz procedure discussed in the lectures to estimate the eigenpairs of the Hermitian matrix $B \in \mathbb{C}^{N\times N}$ using an m-subspace $\mathcal{S}$. Solving the eigenproblem projected into this subspace yields the pairs $(\tilde{\lambda}_i, \tilde{y}_i) \in \mathbb{R} \times \mathbb{C}^{m}$, from which we can in turn compute the approximate eigenvectors $(\tilde{\lambda}_i, \tilde{x}_i) \in \mathbb{R} \times \mathbb{C}^{N}$.

For the computational part of this exercise we will employ the matrix

B = 100×100 SparseArrays.SparseMatrixCSC{Float64, Int64} with 460 stored entries:



```
B = matrixdepot("poisson", 10)
```

which is a sparse matrix resulting from solving a Poisson equation in 2 dimensions as well as the subspace $\mathcal{S}$ spanned by the three vectors

```
100×3 Matrix{Float64}:
 -0.2   4.26326e-16   0.120519
  0.0   0.0           1.11022e-16
  0.0  -0.2          -0.0602595
  0.0   0.0           0.188311
 -0.2  -1.77636e-17  -0.0677919
  0.0   0.0           0.0
  0.0  -0.2           0.128051
  ⋮
  0.0  -0.2          -0.0602595
  0.0   0.0           0.0
 -0.2  -1.77636e-17   0.120519
  0.0   0.0           0.0
  0.0  -0.2          -0.0602595
  0.0   0.0           0.188311
```

```
begin
    V = zeros(size(B, 2), 3)
    V[1:2:end, 2] .= 1.0
    V[1:3:end, 3] .= 1.0
    V[1:4:end, 1] .= 1.0

    V = Matrix(qr(V).Q)
end
```

(a) Show that in general the Ritz eigenvectors $\tilde{x}_i$ and $\tilde{x}_j$ corresponding to different approximate eigenvalues $\tilde{\lambda}_i \neq \tilde{\lambda}_j$ are orthogonal.

solution (a)

We have $\tilde{x}_i = V\tilde{y}_i$ with $V^H V = I$ and $\tilde{y}_i$ coming from an eigendecomposition of $A_V = V^H A V = \sum_i \tilde{\lambda}_i \tilde{y}_i \tilde{y}_i^H$, thus we get

$$\langle \tilde{x}_i, \tilde{x}_j \rangle = \langle V\tilde{y}_i, V\tilde{y}_j \rangle = \langle \tilde{y}_i, V^H V\tilde{y}_j \rangle = \langle \tilde{y}_i, \tilde{y}_j \rangle = 0,$$

where we have used $\tilde{\lambda}_i \neq \tilde{\lambda}_j$ in the last step.

(b) Now concerning the subspace $\mathcal{S}$: Use the Rayleigh-Ritz procedure with subspace $\mathcal{S}$ to obtain three approximate eigenvectors $\tilde{x}_1, \tilde{x}_2, \tilde{x}_3$ and corresponding eigenvalues $\tilde{\lambda}_1, \tilde{\lambda}_2, \tilde{\lambda}_3$, respectively.

```
100×3 Matrix{Float64}:
 0.166733      0.0805088     0.142278
 2.73892e-17   1.07585e-16   1.06838e-18
 0.122233     -0.0918897    -0.14229
 0.0464564     0.182482      0.00181213
 0.120277     -0.101973      0.140466
 0.0           0.0           0.0
 0.16869       0.0905918    -0.140478
 ⋮
 0.122233     -0.0918897    -0.14229
 0.0           0.0           0.0
 0.166733      0.0805088     0.142278
 0.0           0.0           0.0
 0.122233     -0.0918897    -0.14229
 0.0464564     0.182482      0.00181213
```

```
# split: solution
begin
    λ_til, Y_til = eigen(V'*B*V)
    X_til = V * Y_til
end
```

```
[2.00068, 5.27558, 5.80005]
```

```
1 # split: solution
2 λ_til
```

(c) Use the Bauer-Fike theorem to obtain an *a posteriori* bound for the error in the computed eigenvalues of (b). Verify your computed value is indeed an upper bound by computing the exact eigenvalues of the densified matrix (`Matrix(B)`).

```
λ =
[0.162028, 0.398507, 0.398507, 0.634986, 0.771293, 0.771293, 1.00777, 1.00777, 1.25018, 1.25
```

```
1 # split: solution
2 λ = eigvals(Matrix(B))
```

```
residuals = 100×3 Matrix{Float64}:
  0.21112     -0.0108055  -0.113818
 -0.288967     0.0113809   1.20725e-5
  0.0311929   -0.145778    0.11204
 -0.149629    -0.0389069  -0.00143773
  0.0717824    0.0394822  -0.112368
 -0.335423    -0.171101   -0.00180006
  0.216987    -0.0135842   0.112402
  ⋮
  0.0311929   -0.145778    0.11204
 -0.288967     0.0113809   1.20725e-5
  0.21112     -0.0108055  -0.113818
 -0.335423    -0.171101   -0.00180006
  0.0776493    0.0367036   0.113852
 -0.0293522   -0.14088     0.139028
```

```
1 # split: solution
2 residuals = B * X_til - X_til * Diagonal(λ_til)
```

```
1 # split: solution
2 for (λi, xi) in zip(λ_til, eachcol(X_til))
3     r = B * xi - λi * xi
4     @show minimum(abs, λ .- λi)
5     @show norm(r)
6     @show minimum(abs, λ .- λi) ≤ norm(r)
7 end
```

```
minimum(abs, λ .- λi) = 0.03217952744355346
norm(r) = 1.9987068709048956
minimum(abs, λ .- λi) ≤ norm(r) = true
minimum(abs, λ .- λi) = 0.12230032008859837
norm(r) = 1.143271627439608
minimum(abs, λ .- λi) ≤ norm(r) = true
minimum(abs, λ .- λi) = 0.13839120535533578
norm(r) = 0.5991193259712942
minimum(abs, λ .- λi) ≤ norm(r) = true
```

(d) Starting from the subspace $\mathcal{S}$, respectively the vectors `V` run different iterative diagonalisation algorithms, e.g. the `projected_subspace_iteration` and the `lobpcg` routines from the lecture or your *dynamical* shift algorithm from Exercise 1(e). Use `tol=1e-6` and be not afraid to increase `maxiter` for this part of the exercise. You should observe that different eigenpairs are found in each case. Try to explain why each of the algorithms finds the respective eigenpairs. Keeping your orbservations in

mind, can one in general rely on an algorithm to find *all* eigenvalues with correct multiplicity within the part of the spectrum spanned by the smallest and largest eigenvalue the algorithm returns?

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

```
# split: solution
function projected_subspace_iteration(A; tol=1e-6, maxiter=100, verbose=true,
                                      V=randn(eltype(A), size(A, 2), 2),
                                      ortho=ortho_qr)
    T = real(eltype(A))

    eigenvalues    = Vector{T}[]
    residual_norms = Vector{T}[]
    λ = T[]
    Y = nothing
    for i in 1:maxiter
        V = ortho(V)

        AV = A * V
        λ, Y = eigen(Hermitian(V' * AV))  # Notice the change to subspace_iteration
                                          # This is the Rayleigh-Ritz step
        push!(eigenvalues, λ)

        residuals = AV * Y - V * Y * Diagonal(λ)
        norm_r = norm.(eachcol(residuals))
        push!(residual_norms, norm_r)

        verbose && @printf "%3i %8.4g %8.4g\n" i λ[end] norm_r[end]
        maximum(norm_r) < tol && break

        V = AV
    end
    X = V * Y

    (; λ, X, eigenvalues, residual_norms)
end
```

```
(λ = [7.60149, 7.60149, 7.83797], X = 100×3 Matrix{Float64}:                , eigenvalues = [[2
                                        -0.0265974  -0.0287483   0.0144315
```

```
1 # split: solution
2 projected_subspace_iteration(B; V=V, maxiter=1000, tol=1e-6)
```

```
  1      5.8   0.5991
  2     5.92   0.5683
  3    6.036    0.571
  4    6.143   0.5601
  5    6.244   0.5539
  6    6.341   0.5496
  7    6.531    1.107
  8    6.865   0.8769
  9    7.064      0.7
 10     7.19   0.5815
 11    7.277   0.5016
 12    7.342   0.4432
 13    7.392   0.3968
 14    7.433   0.3574
 15    7.465   0.3226
 16    7.492   0.2912
 17    7.513   0.2625
 18    7.531   0.2364
 19    7.545   0.2125
 20    7.556   0.1907
 21    7.565   0.1709
 22    7.573   0.1534
 23    7.595   0.2759
 24    7.614   0.2563
 25    7.631   0.2393
 26    7.645   0.2248
 27    7.658   0.2126
 28     7.67   0.2025
 29     7.68   0.1942
 30     7.69   0.1876
 31    7.699   0.1824
 32    7.708   0.1783
 33    7.716   0.1751
 34    7.724   0.1725
```

One very efficient algorithm for diagonalisation is the LOBPCG algorithm, which is part of the minimisation family of methods. We will discuss it in one of the future lectures in more detail. It is a block algorithm, which is thus capable of solving for a number of eigenvalues. A simple implementation is given below:

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

```
function lobpcg(A; X=randn(eltype(A), size(A, 2), 2), ortho=ortho_qr,
                   Pinv=I, tol=1e-6, maxiter=100, verbose=true)
    T = real(eltype(A))
    m = size(X, 2)  # block size

    eigenvalues    = Vector{T}[]
    residual_norms = Vector{T}[]
    λ = NaN
    P = nothing
    R = nothing

    for i in 1:maxiter
        if i > 1
            Z = hcat(X, P, R)
        else
            Z = X
        end
        Z = ortho(Z)

        # Rayleigh-Ritz step to get smallest eigenvalues
        AZ = A * Z
        λ, Y = eigen(Hermitian(Z' * AZ))
        λ = λ[1:m]
        Y = Y[:, 1:m]
        new_X = Z * Y

        # Store results and residual
        push!(eigenvalues, λ)
        R = AZ * Y - new_X * Diagonal(λ)
        norm_r = norm.(eachcol(R))
        push!(residual_norms, norm_r)
        verbose && @printf "%3i %8.4g %8.4g\n" i λ[end] norm_r[end]
        if maximum(norm_r) < tol
            X = new_X
            break
        end

        # Precondition residual, update X and P
        R = Pinv * R
        P = X - new_X
        X = new_X
    end

    (; λ, X, eigenvalues, residual_norms)
end
```

This algorithms solves for `size(X, 2)` eigenpairs (by default 2).

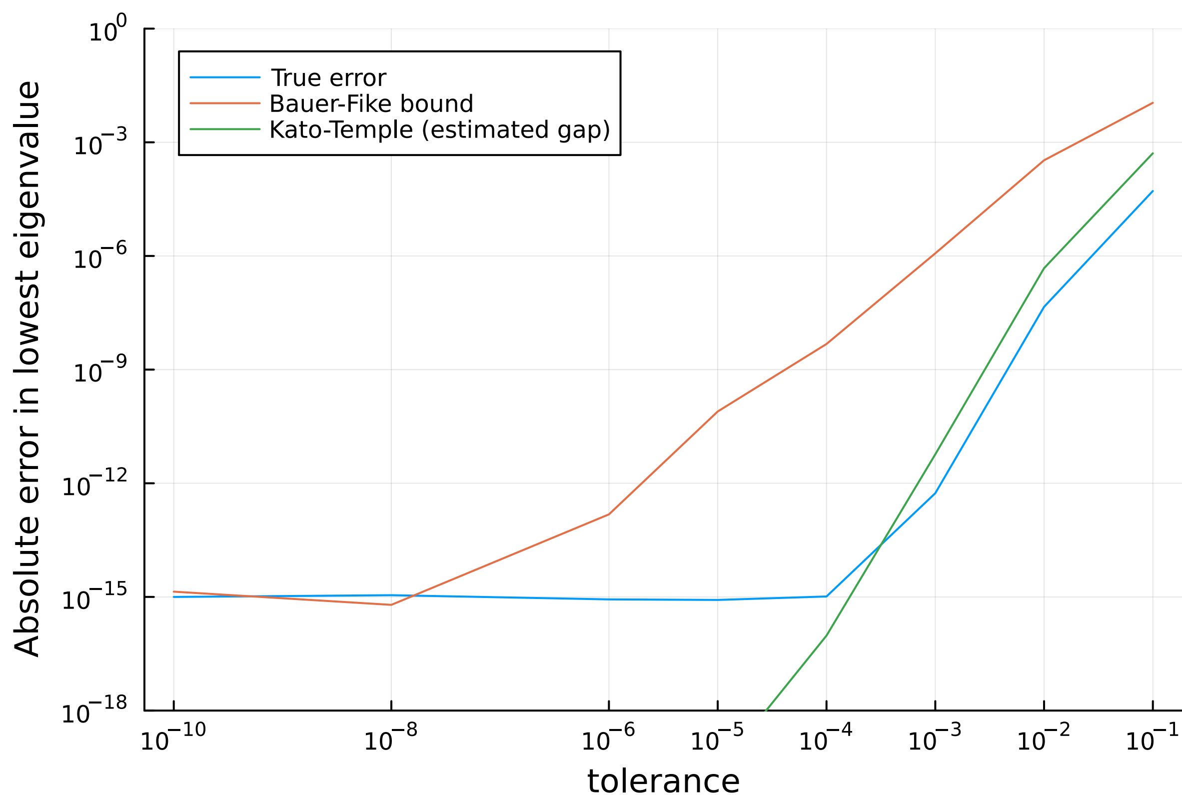
(e) Run this LOBPCG algorithm on `B` starting from a random guess aiming for 4 eigenvectors. Use the Bauer-Fike and Kato-Temple theorems to estimate the error in the first (lowest) eigenvalue. Use the

tightest estimate that is available to you. You may assume that the LOBPCG algorithm did not miss any eigenvalue, i.e. that you have indeed approximations for the first and second eigenpair at your disposal. Vary the tolerance between `1e-1` and `1e-10` and plot the relationships between tolerance, estimated error and true error. *Hint: Keep in mind your floating-point precision when interpreting your results.*

```
(λ = [0.162028, 0.398507, 0.398507, 0.634986], X = 100×4 Matrix{Float64}:
                                                   -0.0144315   0.0086911  -0.0381885   0.
```

```
1 # split: solution
2 let Pinv = Diagonal(1 ./ diag(B))
3     X = randn(size(B, 2), 4)
4     lobpcg(B; X, Pinv, tol=1e-6)
5 end
```

```
 1    4.395     1.825
 2    2.221     1.288
 3    1.567    0.8261
 4    1.226    0.7777
 5   0.8757    0.6839
 6    0.763    0.3478
 7   0.7138    0.2568
 8   0.6854    0.1887
 9   0.6687    0.1517
10   0.6569    0.1322
11   0.6477    0.1067
12   0.6423   0.08027
13   0.6388   0.07321
14   0.6365    0.0551
15   0.6353   0.03025
16   0.6351   0.01416
17    0.635  0.007438
18    0.635  0.002779
19    0.635  0.002266
20    0.635   0.00156
21    0.635  0.001133
22    0.635  0.001286
23    0.635  0.001051
24    0.635  0.0007105
25    0.635  0.0005222
26    0.635  0.0003278
27    0.635  0.0001779
28    0.635  0.0001028
29    0.635  7.042e-05
30    0.635  5.163e-05
31    0.635  4.559e-05
32    0.635  2.874e-05
33    0.635  2.023e-05
34    0.635  1.367e-05
```




```
1  # split: solution
2  let Pinv = Diagonal(1 ./ diag(B))
3      X = randn(size(B, 2), 4)
4
5      tol_range = [1e-10, 1e-8, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1]
6      true_error = Float64[]
7      bauer_fike = Float64[]
8      kato_temple = Float64[]
9      for tol in tol_range
10         result = lobpcg(B; X, Pinv, tol, maxiter=10_000, verbose=false)
11
12         # Bauer-Fike
13         r =  result.residual_norms[end][1]
14         push!(bauer_fike, r)
15
16         # Kato-Temple
17         gap = abs(result.λ[1] - result.λ[2])
18         push!(kato_temple, r^2 / gap)
19
20         # True error
21         push!(true_error, abs(λ[1] - result.λ[1]))
22     end
23
24     plt = plot(xaxis=:log, yaxis=:log, xlabel="tolerance", ylabel="Absolute error in
       lowest eigenvalue", legend=:topleft)
25     ylims!((1e-18, 1))
26     xticks!(tol_range)
27     plot!(tol_range, true_error, label="True error")
28     plot!(tol_range, bauer_fike, label="Bauer-Fike bound")
29     plot!(tol_range, kato_temple, label="Kato-Temple (estimated gap)")
30 end
```