Error control in scientific modelling (MATH 500, Herbst)

# Sheet 6: Diagonalisation algorithms (10 P)

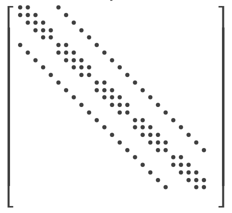
*To be handed in via moodle by 02.11.2023*

```
begin
    using MatrixDepot
    using LinearAlgebra
    using PlutoUI
    using Printf
    using Plots
    using Statistics
end
```

## Exercise 1 (1 + 1.5 + 1 + 0.5 + 1 P)

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 = 25×25 SparseArrays.SparseMatrixCSC{Float64, Int64} with 105 stored entries:
```



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

Size of the test matrix: $n =$ 5

Selection deleted

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.

**(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?

**(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 automise 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 $A$.

**(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 assesment?

**(a) Solution:** Selection deleted

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

```julia
ortho_qr(A) = Matrix(qr(A).Q)
```

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

```julia
function projected_subspace_iteration_modified(A; tol=1e-6, maxiter=100,
verbose=true,
                                      X=randn(eltype(A), size(A, 2), 2),
                                      ortho=ortho_qr)
    T = real(eltype(A))

    eigenvalues    = Vector{T}[]
    residual_norms = Vector{T}[]
    λ = T[]
    for i in 1:maxiter
        X = ortho(X)
        AX =  A \ X
        λ, Y = eigen(Symmetric(X' * AX))  # Notice the change to subspace_iteration
                               # This is the Rayleigh-Ritz step
        push!(eigenvalues, λ)

        max_inx = sortperm((abs.(λ)), rev = true)[1:2]  # Get the indices of the two
        largest absolute values

        residuals = AX * Y - X * Y * Diagonal(λ)
        norm_r = norm.(eachcol(residuals))
        push!(residual_norms, norm_r[max_inx])

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

        X = AX
    end

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

Selection deleted

```
(λ = [3.73205, 3.73205], X = 25×2 Matrix{Float64}:      , eigenvalues = [[0.177982, 1.161
                              -0.0524451     -0.197272
```

```julia
begin
    σ = 1  # Our approximation to the eigenvalue of interest
    shifted  = A - σ * I        # Shift the matrix
    # Do not need to factorize since it's a Possion matrix
    λ, X, eigenvalues, residual_norms = projected_subspace_iteration_modified(shifted)
end
```

```
 1    1.161    1.657
 2    3.581   0.7108
 3    3.727   0.1193
 4    3.732   0.0272
 5    3.732 0.006923
 6    3.732 0.001815
 7    3.732 0.0004806
 8    3.732 0.0001279
 9    3.732 3.411e-05
10    3.732 9.116e-06
11    3.732 2.439e-06
12    3.732 6.527e-07
13    3.732 1.748e-07
14    3.732 4.682e-08
15    3.732 1.254e-08
16    3.732 3.36e-09
17    3.732 9.002e-10
18    3.732 2.412e-10
19    3.732 6.462e-11
20    3.732 1.732e-11
21    3.732 4.662e-12
22    3.732 2.272e-12
23    3.732 1.059e-12
24    3.732 2.945e-12
25    3.732 3.193e-13
26    3.732 2.188e-11
27    3.732 3.852e-11
28    3.732 4.576e-11
29    3.732 8.355e-11
30    3.732 2.506e-10
31    3.732 1.989e-10
```

```
[1.26795, 1.26795]
```

```julia
1 ./ λ .+ σ
```

```
[[1.65738, 0.681627], [0.710768, 1.31152], [0.119336, 1.02404], [0.0271956, 1.53688], [0.0
```

```julia
residual_norms
```

Selection deleted

**(b) Solution:**

2

```julia
@bind k PlutoUI.Slider(2:1:5; show_value=true, default=2)
```

```
[[1.31711, 1.04396], [1.42278, 2.48811], [0.320568, 2.26386], [0.0679202, 1.49248], [0.015
```

```julia
begin
    S = randn(eltype(A), size(A, 2), k)
    result = projected_subspace_iteration_modified(shifted, X=S)

    λ_S = result.λ
    r_norms = result.residual_norms
    # reshaped_r_norms = reshape(hcat(r_norms...), k, :)
end
```

```
 1   0.6774    1.317
 2     3.09    1.423
 3    3.698   0.3206
 4     3.73  0.06792
 5    3.732   0.0151
 6    3.732 0.003478
 7    3.732 0.0008259
 8    3.732 0.0002018
 9    3.732 5.058e-05
10    3.732 1.294e-05
11    3.732 3.359e-06
12    3.732 8.818e-07
13    3.732 2.332e-07
14    3.732 6.196e-08
15    3.732 1.651e-08
16    3.732 4.41e-09
17    3.732 1.179e-09
18    3.732 3.156e-10
19    3.732 8.451e-11
20    3.732 2.322e-11
21    3.732 1.152e-11
22    3.732 7.275e-12
23    3.732 2.39e-10
24    3.732 1.009e-10
25    3.732 5.098e-11
26    3.732 3.83e-10
27    3.732 3.082e-10
28    3.732 8.84e-10
29    3.732 1.139e-09
30    3.732 3.915e-09
31    3.732 1.593e-08
```

```
[1.26795, 1.26795]
```

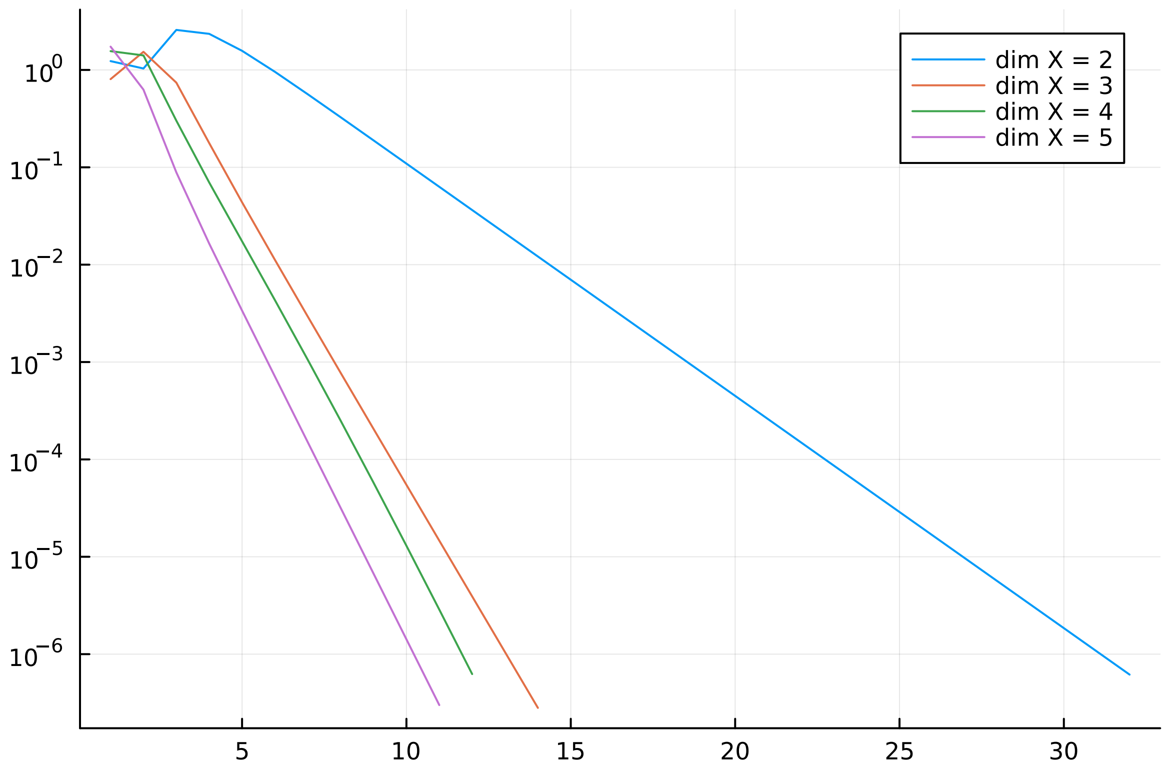
```julia
1 ./ λ_S .+ σ # Original eigenvalues
```

Selection deleted

```julia
begin
    res_norms = []
    for i in 2:5
        subspace = randn(eltype(A), size(A, 2), i)
        r_norm = projected_subspace_iteration_modified(shifted,
        X=subspace).residual_norms
        push!(res_norms, mapslices(maximum, hcat(r_norm...); dims=1));
    end;
end
```

```
 1   0.9989    1.232
 2    3.451   0.8989
 3    3.715   0.2302
 4    3.731  0.05909
 5    3.732  0.01559
 6    3.732  0.00416
 7    3.732 0.001113
 8    3.732 0.000298
 9    3.732 7.983e-05
10    3.732 2.139e-05
11    3.732 5.73e-06
12    3.732 1.535e-06
13    3.732 4.113e-07
14    3.732 1.102e-07
15    3.732 2.953e-08
16    3.732 7.913e-09
17    3.732 2.12e-09
18    3.732 5.681e-10
19    3.732 1.523e-10
20    3.732 4.079e-11
21    3.732 1.093e-11
22    3.732 2.661e-11
23    3.732 4.056e-11
24    3.732 4.69e-11
25    3.732 3.368e-11
26    3.732 4.085e-11
27    3.732 5.786e-10
28    3.732 8.875e-11
29    3.732 4.964e-10
30    3.732 3.408e-10
31    3.732 6.457e-10
```

Selection deleted



```julia
begin
    p = plot(; yaxis=:log)
    plot!(p, 1:size(res_norms[1], 2), vec(res_norms[1]), label="dim X = 2")
    plot!(p, 1:size(res_norms[2], 2), vec(res_norms[2]), label="dim X = 3")
    plot!(p, 1:size(res_norms[3], 2), vec(res_norms[3]), label="dim X = 4")
    plot!(p, 1:size(res_norms[4], 2), vec(res_norms[4]), label="dim X = 5")
end
```

By observing the plot with the calculated residual norms, we can see that the biggest subspace size tends to converge in the least number of iterations.

**(c) Solution:**

The most economical configuration could be determined based on a trade-off between computational efficiency and the number of iterations required for convergence.

- On one hand, the number of iterations required decreases as the subspace size increases.
- On the other hand, increasing the subspace size will increase the time per iteration.

Selection deleted

If running the algorithm with the subspace size equal to $5$ has the capability of decreasing the number of iterations by a factor bigger than $2.5$, compared with subspace size $2$. Then it would imply that using a bigger subspace is more beneficial but by playing with different examples we can see that it is not always the case.

```julia
begin
    n_measurements = 5
    r_times = Dict()

    for i in 2:5
        subspace = randn(eltype(A), size(A, 2), i)
        times = []

        for _ in 1:n_measurements
            t = @elapsed begin
                projected_subspace_iteration_modified(A, X=subspace)
            end
            push!(times, t)
        end

        r_times[i] = mean(times)
    end
end
```

```
 1   0.4144   0.4995
 2    1.642   0.5339
 3    1.854   0.1283
 4    1.865  0.03338
 5    1.866 0.008955
 6    1.866 0.002475
 7    1.866 0.0007213
 8    1.866 0.0002292
 9    1.866 8.08e-05
10    1.866 3.104e-05
11    1.866 1.256e-05
12    1.866 5.209e-06
13    1.866 2.185e-06
14    1.866 9.206e-07
15    1.866 3.886e-07
16    1.866 1.642e-07
17    1.866 6.937e-08
18    1.866 2.932e-08
19    1.866 1.239e-08
20    1.866 5.237e-09
21    1.866 2.213e-09
22    1.866 9.355e-10
23    1.866 3.954e-10
24    1.866 1.671e-10
25    1.866 7.063e-11
26    1.866 2.985e-11
27    1.866 1.262e-11
28    1.866 5.332e-12
 1   0.4144   0.4995
 2    1.642   0.5339
 3    1.854   0.1283
```

Selection deleted

```
(λ = [0.440905, 0.499969, 0.788675, 0.788675, 1.86603], X = 25×5 Matrix{Float64}:
                                                              -0.0833326   -0.175042    0.105
```

```julia
begin
    subspace = randn(eltype(A), size(A, 2), 5)
    t = @time begin
                projected_subspace_iteration_modified(A, X=subspace)
    end
end
```

```
 1    0.6849    0.6062
 2     1.767    0.3765
 3     1.862   0.07325
 4     1.866   0.01918
 5     1.866  0.005214
 6     1.866  0.001288
 7     1.866  0.0002946
 8     1.866  6.484e-05
 9     1.866  1.424e-05
10     1.866  3.183e-06
11     1.866  7.26e-07
12     1.866  1.68e-07
13     1.866  3.926e-08
14     1.866  9.218e-09
15     1.866  2.171e-09
16     1.866  5.12e-10
17     1.866  1.209e-10
18     1.866  2.855e-11
 0.002137 seconds (1.82 k allocations: 530.508 KiB)
```

```julia
for i in 2:5
    println("subspace size: $i, average runtime: $(@sprintf("%.5f", r_times[i]))
    seconds")
end
```

```
subspace size: 2, average runtime: 0.00356 seconds
subspace size: 3, average runtime: 0.00361 seconds
subspace size: 4, average runtime: 0.00532 seconds
subspace size: 5, average runtime: 0.00393 seconds
```

Analyzing the running times for different examples we can conclude that in a lot of cases using a bigger subspace is more beneficial. At the same time, looking at the number of iterations it took before convergence, we can see that the relation between the computational time per iteration and the number of subspace vectors might not be linear.

Selection deleted

**(d) Solution:**

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

```julia
function projected_subspace_iteration_RQI(A; tol=1e-6, maxiter=100, verbose=true,
                                           X=randn(eltype(A), size(A, 2), 2),
                                           ortho=ortho_qr)
    T = real(eltype(A))

    eigenvalues    = Vector{T}[]
    residual_norms = Vector{T}[]
    λ = T[]
    for i in 1:maxiter
        X = ortho(X)
        AX =  A * X
        λ, Y = eigen(Symmetric(X' * AX))  # Notice the change to subspace_iteration
                                # This is the Rayleigh-Ritz step
        push!(eigenvalues, λ)

        max_inx = sortperm((abs.(λ)), rev = true)[1:2]  # Get the indices of the two
        largest absolute values

        residuals = AX * Y - X * Y * Diagonal(λ)
        norm_r = norm.(eachcol(residuals))
        push!(residual_norms, norm_r[max_inx])

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

        # X = (A - X * Diagonal(λ) * X') \ X
        for i in 1:size(X,2)
            X[:, i] = (A - λ[i] * I) \ X[:, i]
        end
    end

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

```
7815, 1.75947], [0.639019, 0.402487], [0.000767987, 0.0147031], [1.3192e-9, 1.29052e-6], [7.6
```

```julia
result_rqi = projected_subspace_iteration_RQI(A)
```

```
  1    4.791    1.678
  2    4.999    0.639
  3        5 0.000768
  4        5 1.319e-09
  5        5 7.634e-16
```

Selection deleted

```
[4.0, 5.0]
```

```julia
result_rqi.λ
```

```
25×2 Matrix{Float64}:
 -5.55112e-17  -2.22045e-16
 -1.11022e-16   0.0
 -2.22045e-16   5.55112e-17
  1.11022e-16  -2.22045e-16
  2.22045e-16   0.0
  3.33067e-16  -5.55112e-17
  2.22045e-16   3.05747e-17
  ⋮
  2.22045e-16   0.0
  2.22045e-16   2.22045e-16
  1.11022e-16   0.0
 -3.33067e-16   5.55112e-17
 -2.22045e-16   0.0
 -5.55112e-17   0.0
```

```julia
A * result_rqi.X - result_rqi.X * Diagonal(result_rqi.λ)
```

**(e) Solution:**

Selection deleted

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

```julia
function projected_subspace_iteration_chained(A; tol=1e-6, maxiter=100, verbose=true,
                                              X=randn(eltype(A), size(A, 2), 2),
                                              ortho=ortho_qr)
    T = real(eltype(A))

    eigenvalues    = Vector{T}[]
    residual_norms = Vector{T}[]
    λ = T[]

    σ = 1
    shifted  = A - σ * I
    X = projected_subspace_iteration_modified(shifted, X=X, maxiter=1).X

    for i in 1:maxiter
        X = ortho(X)
        AX =  A * X
        λ, Y = eigen(Symmetric(X' * AX))  # Notice the change to subspace_iteration
                              # This is the Rayleigh-Ritz step
        push!(eigenvalues, λ)

        max_inx = sortperm((abs.(λ)), rev = true)[1:2]  # Get the indices of the two largest absolute values

        residuals = AX * Y - X * Y * Diagonal(λ)
        norm_r = norm.(eachcol(residuals))
        push!(residual_norms, norm_r[max_inx])

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

        for i in 1:size(X,2)
            X[:, i] = (A - λ[i] * I) \ X[:, i]
        end
    end

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

```
(λ = [1.26795, 2.26795], X = 25×2 Matrix{Float64}:   , eigenvalues = [[1.42476, 2.22708],
                              -0.201944    0.0373695
```

```julia
projected_subspace_iteration_chained(A)
```

Selection deleted

```
  1    2.009    1.154
  1    2.227    1.406
  2    2.233   0.1736
  3    2.267  0.01282
  4    2.268 2.276e-05
  5    2.268 1.608e-13
```

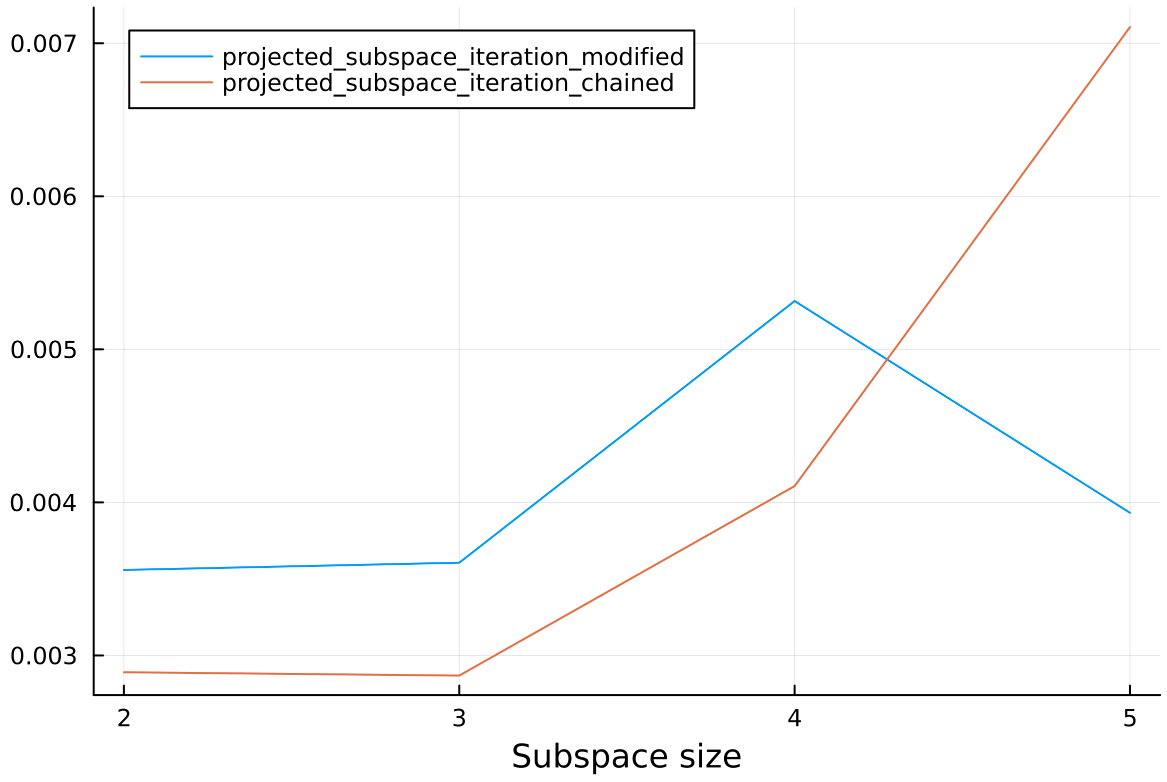
```julia
begin
    run_times_dyn = Dict()

    for i in 2:5
        subspace = randn(eltype(A), size(A, 2), i)
        times = []

        for _ in 1:n_measurements
            t = @elapsed begin
                projected_subspace_iteration_chained(A, X=subspace)
            end
            push!(times, t)
        end

        run_times_dyn[i] = mean(times)
    end
end
```

```
1   0.3428    0.4522
1    2.112     1.419
2    2.091    0.4521
3    2.233    0.1269
4    2.267   0.01329
5    2.268 2.956e-05
6    2.268 3.57e-13
1   0.3428    0.4522
1    2.112     1.419
2    2.091    0.4521
3    2.233    0.1269
4    2.267   0.01329
5    2.268 2.956e-05
6    2.268 3.57e-13
1   0.3428    0.4522
1    2.112     1.419
2    2.091    0.4521
3    2.233    0.1269
4    2.267   0.01329
5    2.268 2.956e-05
6    2.268 3.57e-13
1   0.3428    0.4522
1    2.112     1.419
2    2.091    0.4521
3    2.233    0.1269
4    2.267   0.01329
5    2.268 2.956e-05
6    2.268 3.57e-13
1   0.3428    0.4522
1    2.112     1.419
2    2.091    0.4521
```

Selection deleted

```julia
for i in 2:5
    println("subspace size: $i, average runtime: $(@sprintf("%.5f",
    run_times_dyn[i])) seconds")
end
```

```
subspace size: 2, average runtime: 0.00289 seconds
subspace size: 3, average runtime: 0.00287 seconds
subspace size: 4, average runtime: 0.00411 seconds
subspace size: 5, average runtime: 0.00711 seconds
```



```julia
begin
    plot(r_times, label="projected_subspace_iteration_modified")
    plot!(run_times_dyn, label="projected_subspace_iteration_chained",
    xlabel="Subspace size")
end
```

After testing the algorithm for different sizes of matrix $A$ we can conclude that employing the dynamical shift can significantly decrease the number of iterations but in terms of time, it doesn't give an advantage. Changing the size of matrix A we can notice that the difference in running time between algorithms is increasing with the growing size of the matrix. In particular, for the algorithm with the dynamical shift, the time wrt to the size of the subspace is growing faster.

Selection deleted

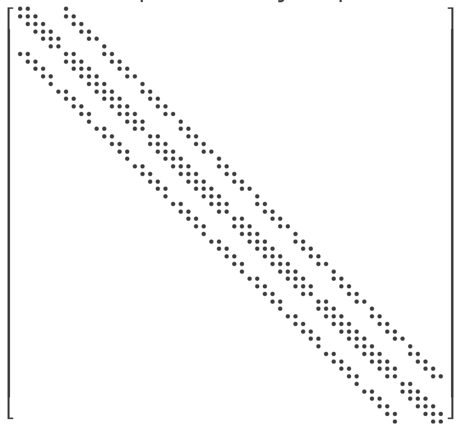
# Exercise 2 (1 + 1 + 1 + 1 + 1 P)

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 Ritz 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

Selection deleted

```
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
```

```julia
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 the Ritz eigenvectors $\tilde{x}_i$ and $\tilde{x}_j$ corresponding to different approximate eigenvalues $\tilde{\lambda}_i \neq \tilde{\lambda}_j$ are orthogonal.

**(b)** Use the Rayleigh-Ritz procedure with the given subspace 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.

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

**(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?

**(e)** Run the 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 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-4` and `1e-10` and plot the relationships between tolerance, estimated error and true error.

**(a) Solution:** In this case, the Rayleigh-Ritz procedure will be applied to the Hermitian matrix $B$ using a subspace $\mathcal{S}$, which means that we first need to compute $B_V = V^H BV$ with $V$ containing the basis vectors of $\mathcal{S}$ as columns. Considering Ritz pairs $(\tilde{\lambda}_i, \tilde{x}_i)$ and $(\tilde{\lambda}_j, \tilde{x}_j)$, where $\tilde{\lambda}_i \neq \tilde{\lambda}_j$, we have

$$B_V \tilde{x}_i = \tilde{\lambda}_i \tilde{x}_i,$$

$$B_V \tilde{x}_j = \tilde{\lambda}_j \tilde{x}_j.$$

Multiplying by $\tilde{x}_j^H$ and $\tilde{x}_i^H$ respectively:

$$(1) \quad \tilde{x}_j^H B_V \tilde{x}_i = \tilde{\lambda}_i \tilde{x}_j^H \tilde{x}_i,$$

$$(2) \quad \tilde{x}_i^H B_V \tilde{x}_j = \tilde{\lambda}_j \tilde{x}_i^H \tilde{x}_j.$$

Since $B_V = V^H BV$ and $B$ is Hermitian we have that:

$$\tilde{x}_j^H B_V \tilde{x}_i = \tilde{x}_i^H B_V \tilde{x}_j.$$

By substracting $(2)$ from $(1)$ we get:

$$0 = \tilde{x}_j^H B_V \tilde{x}_i - \tilde{x}_i^H B_V \tilde{x}_j = \tilde{\lambda}_i \tilde{x}_j^H \tilde{x}_i - \tilde{\lambda}_j \tilde{x}_i^H \tilde{x}_j$$

Since $\tilde{\lambda}_i \neq \tilde{\lambda}_j$ it means that:

$$\tilde{x}_i^H \tilde{x}_j = 0,$$

which shows that the Ritz eigenvectors $\tilde{x}_i$ and $\tilde{x}_j$ are orthogonal.

**(b) Solution:**

```
Bv = 3×3 Matrix{Float64}:
   4.0       -1.8     0.549868
  -1.8        4.0     0.5574
   0.549868   0.5574  5.07631
```

```julia
Bv = V' * B * V
```

```julia
e_values, e_vectors = eigen(Bv);
```

```julia
println("Approximate eigenvalues: \n", e_values)
```

```
Approximate eigenvalues: 
[2.0006837303475127, 5.275577069027193, 5.800051257362882]
```

```
approx_evectors = 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
```

```
approx_evectors = V * e_vectors
```

**(c) Solution:**

```
100×3 Matrix{Float64}:
 0.333581      0.424731      0.825221
 5.47972e-17   5.67576e-16   6.19665e-18
 0.24455      -0.484771     -0.825291
 0.0929445     0.962695      0.0105105
 0.240636     -0.537965      0.814711
 0.0           0.0           0.0
 0.337494      0.477924     -0.814781
 ⋮
 0.24455      -0.484771     -0.825291
 0.0           0.0           0.0
 0.333581      0.424731      0.825221
 0.0           0.0           0.0
 0.24455      -0.484771     -0.825291
 0.0929445     0.962695      0.0105105
```

```
V *  e_vectors * Diagonal(e_values)
```

```
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
```

```
residuals = Matrix(B) * V * e_vectors - V *  e_vectors * Diagonal(e_values)
```

```
error_bounds =  [1.99871, 1.14327, 0.599119]
```

```
error_bounds = [norm(residuals[:, i]) for i in 1:3]
```

```
exact_eigenvalues =
 [0.162028, 0.398507, 0.398507, 0.634986, 0.771293, 0.771293, 1.00777, 1.00777, 1.25018, 1.
```

```julia
exact_eigenvalues = eigen(Matrix(B)).values
```

```
actual_errors =  [0.0321795, 0.1223, 0.138391]
```

```julia
actual_errors = [minimum(abs.(exact_eigenvalues .- e_val)) for e_val in e_values]
```

```
true
```

```julia
all(error_bounds .>= actual_errors) # checking that the upper bound is true
```

```
 [1.99871, 1.14327, 0.599119]
```

```julia
error_bounds
```

```
 [0.0321795, 0.1223, 0.138391]
```

```julia
actual_errors
```

**(d) Solution:**

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

```julia
function projected_subspace_iteration(A; tol=1e-6, maxiter=100, verbose=true,
                                      X=randn(eltype(A), size(A, 2), 2),
                                      ortho=ortho_qr)
    T = real(eltype(A))

    eigenvalues    = Vector{T}[]
    residual_norms = Vector{T}[]
    λ = T[]
    for i in 1:maxiter
        X = ortho(X)

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

        residuals = AX * Y - X * 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

        X = AX
    end

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

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

```julia
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]
        maximum(norm_r) < tol && break

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

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

```
pci_B =
  (λ = [7.60149, 7.60149, 7.83797], X = 100×3 Matrix{Float64}:                , eigenvalues =
                                         -0.0144313   0.0276938  -0.0276938
```

```
pci_B = projected_subspace_iteration(B, X=V, maxiter=500)
```

```
162     7.838 5.225e-05
163     7.838 4.902e-05
164     7.838  4.6e-05
165     7.838 4.316e-05
166     7.838 4.05e-05
167     7.838 3.801e-05
168     7.838 3.567e-05
169     7.838 3.347e-05
170     7.838 3.142e-05
171     7.838 2.949e-05
172     7.838 2.768e-05
173     7.838 2.598e-05
174     7.838 2.438e-05
175     7.838 2.289e-05
176     7.838 2.149e-05
177     7.838 2.017e-05
178     7.838 1.894e-05
179     7.838 1.778e-05
180     7.838 1.669e-05
181     7.838 1.567e-05
182     7.838 1.472e-05
183     7.838 1.382e-05
184     7.838 1.297e-05
185     7.838 1.218e-05
186     7.838 1.144e-05
187     7.838 1.074e-05
188     7.838 1.009e-05
189     7.838 9.473e-06
190     7.838 8.897e-06
191     7.838 8.355e-06
192     7.838 7.847e-06
193     7.838 7.369e-06
194     7.838 6.921e-06
```

```
[4.0119e-12, 8.88178e-16, -1.77636e-15]
```

```
exact_eigenvalues[end-2:end] .- pci_B.λ
```

```
lobpcg_B =
  (λ = [0.162028, 0.398507, 0.398507], X = 100×3 Matrix{Float64}:              , eigenvalues
                                            0.0144315    0.0307379    0.0242709
```

```
lobpcg_B = lobpcg(B, X=V, maxiter=500)
```

```
 13   0.4092   0.1112
 14   0.4032  0.08742
 15   0.4001  0.05863
 16    0.399  0.03031
 17   0.3987  0.01528
 18   0.3986  0.01084
 19   0.3986  0.00854
 20   0.3985  0.00591
 21   0.3985 0.005007
 22   0.3985 0.003944
 23   0.3985 0.003156
 24   0.3985 0.002037
 25   0.3985 0.001342
 26   0.3985 0.0009197
 27   0.3985 0.0007453
 28   0.3985 0.0005209
 29   0.3985 0.0003258
 30   0.3985 0.0002077
 31   0.3985 0.0001616
 32   0.3985 0.0001144
 33   0.3985 6.195e-05
 34   0.3985 3.628e-05
 35   0.3985 2.103e-05
 36   0.3985 1.196e-05
 37   0.3985 1.033e-05
 38   0.3985 7.205e-06
 39   0.3985 4.391e-06
 40   0.3985 2.669e-06
 41   0.3985 1.994e-06
 42   0.3985 1.372e-06
 43   0.3985 9.893e-07
```

```
[2.77556e-17, -5.27356e-15, -6.21225e-13]
```

```
exact_eigenvalues[1:3] .- lobpcg_B.λ
```

We can notice that the `projected_subspace_iteration` method focuses on approximating the three largest eigenvalues of the original matrix $B$ but can be generalized for any part of the spectrum. LOBPCG gives the smallest three eigenvalues which can be explained by the fact that iterative minimization of the generalized Rayleigh quotient allows to get the smallest eigenvalues.

LOBPCG is particularly useful for large problems where efficiency in finding a specific set of eigenpairs is desired, while `projected_subspace_iteration` method might be more suitable for smaller problems where computing the whole spectrum is feasible. Both algorithms are efficient at approximating eigenvalues within specific ranges but might not ensure the capture of all eigenvalues.

**(e) Solution:**

```julia
begin
    result_B = lobpcg(B, X=randn(eltype(B), size(B, 2), 4))

    e_val = result_B.λ
    e_vec = result_B.X
end;
```

```
 1    4.291    2.085
 2    2.327    1.313
 3    1.578   0.9677
 4    1.198   0.6787
 5   0.9651   0.5635
 6   0.8273    0.417
 7   0.7553     0.31
 8   0.7004   0.2924
 9   0.6627   0.2138
10   0.6451   0.1348
11   0.6395  0.06856
12   0.6377  0.04615
13   0.6366   0.0389
14   0.6359  0.03038
15   0.6355    0.021
16   0.6353  0.01833
17   0.6351  0.01456
18   0.6351 0.008261
19    0.635 0.006876
20    0.635 0.005281
21    0.635 0.004681
22    0.635 0.003051
23    0.635  0.00186
24    0.635 0.0009842
25    0.635 0.0006657
26    0.635 0.0005189
27    0.635 0.0003056
28    0.635 0.0002059
29    0.635 0.0001585
30    0.635 0.0001291
31    0.635 8.995e-05
32    0.635 5.962e-05
```

```
[0.162028, 0.398507, 0.398507, 0.634986]
```

```julia
e_val
```

```
error_Bauer_Fike = 2.784413844756934e-11
```

```julia
#the Bauer-Fike theorem
error_Bauer_Fike = norm(B * e_vec[:, 1] - e_val[1] * e_vec[:, 1])
```

```
δ = 0.23647887816325847
```

```julia
δ = abs(e_val[1] - e_val[2]) - norm(B * e_vec[:, 2] - e_val[2] * e_vec[:, 2])
```

```
error_Kato_Temple = 3.278500185340723e-21
```

```julia
# the Kato-Temple theorem
error_Kato_Temple =  error_Bauer_Fike .^2 ./ δ
```

```
tolerances =  [0.0001, 1.0e-5, 1.0e-6, 1.0e-7, 1.0e-8, 1.0e-9, 1.0e-10]
```

```julia
tolerances = [10.0^p for p in -4:-1:-10]
```

```
first_exact_evalue = 0.16202810554201064
```

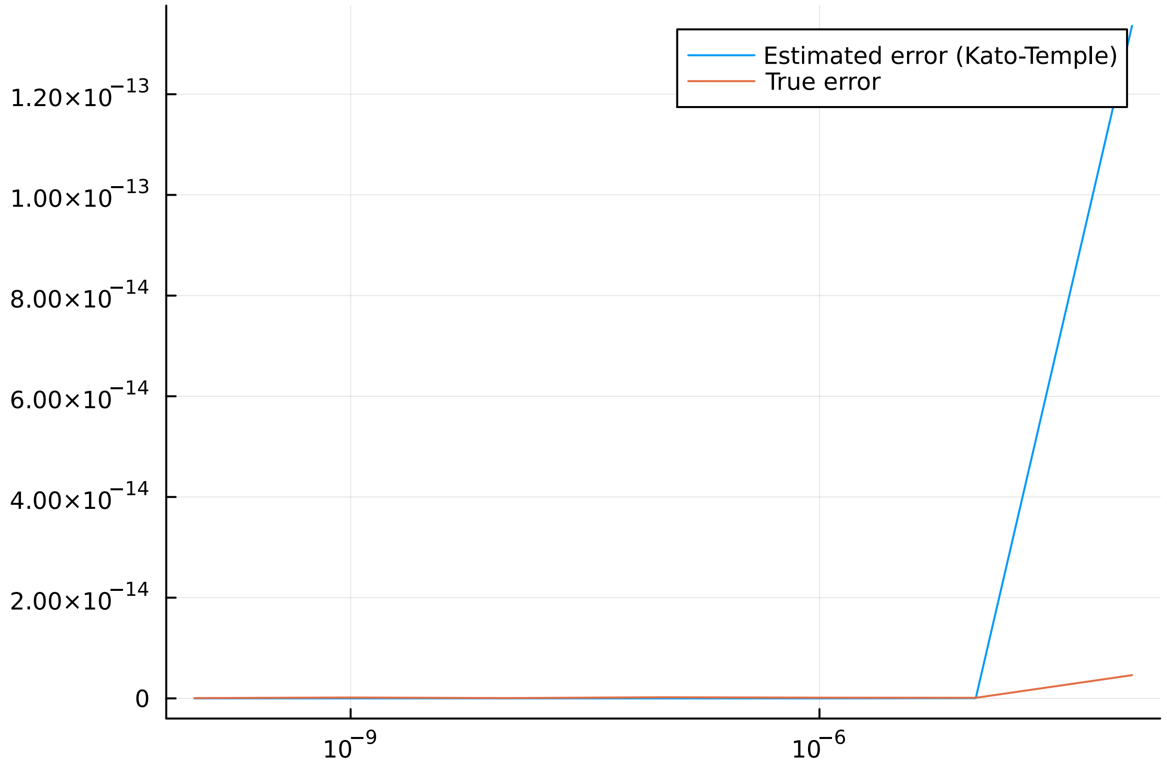
```julia
first_exact_evalue = exact_eigenvalues[1]
```

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

```julia
e_values
```

```julia
begin
    true_errors = []
    estimated_errors = []

    for tol in tolerances
        result_lobpcg = lobpcg(B, X=randn(eltype(B), size(B, 2), 4),tol=tol)

        e_values = result_lobpcg.λ
        e_vectors = result_lobpcg.X

        δ = abs(e_values[1] - e_values[2]) - norm(B * e_vectors[:, 2] - e_values[2]
        * e_vectors[:, 2])

        e_Bauer_Fike = norm(B * e_vectors[:, 1] - e_values[1] * e_vectors[:, 1])

        e_Kato_Temple =  e_Bauer_Fike .^2 ./ δ
        e_true = abs(first_exact_evalue - e_values[1])

        push!(estimated_errors, e_Kato_Temple)
        push!(true_errors, e_true)
    end
end
```

```
18     0.635 0.003046
19     0.635 0.002735
20     0.635 0.001855
21     0.635 0.001413
22     0.635 0.001429
23     0.635   0.0012
24     0.635 0.0008065
25     0.635 0.0005221
26     0.635 0.0003975
27     0.635 0.000291
28     0.635 0.0001776
29     0.635 0.0001021
30     0.635 7.57e-05
 1     4.646    2.161
 2     2.819    1.101
 3     2.136    1.046
 4     1.552   0.8874
 5     1.167   0.7234
 6    0.8874   0.6764
 7    0.6961   0.4733
 8    0.6501   0.1538
 9     0.642  0.09024
10     0.639  0.05247
11    0.6378  0.04065
12     0.637  0.03979
13     0.636   0.0372
14    0.6354  0.02451
15    0.6352  0.01563
16    0.6351  0.01024
17     0.635 0.007232
18     0.635 0.004485
19     0.635 0.003085
20     0.635 0.001984
```



```julia
begin
    plot(tolerances, estimated_errors, label="Estimated error (Kato-Temple)")
    plot!(tolerances, true_errors, label="True error", xscale=:log10)
end
```