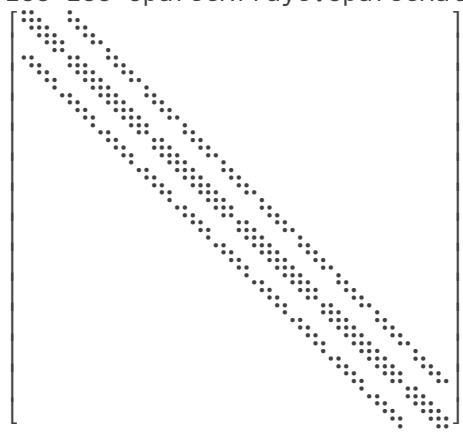
Sheet 6: Diagonalisation algorithms

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

Exercise 1

Size of the test matrix: $n =$ 10

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



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

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

```
function ortho_qr(X)
    Matrix(qr(X).Q)
end
```

(a)

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

```
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
```

```
Enter cell code...
```

```
[1.00777, 1.00777]
```

```
let
    σ = 1.0
    A_shifted = A - σ * I
    λ_shifted = projected_subspace_inverse_iteration(A_shifted).λ
    (1 ./ λ_shifted) .+ σ  # transform back
end
```

```
  1 [1.0889292240460282, 4.185720762385721] [9.556969784487963, 22.09037685
1393035]
  2 [126.91163298923968, 128.57267767929812] [15.001112127273364, 3.6432245657
243447]
  3 [128.6744582162202, 128.67580766988124] [0.42517693878768476, 0.0677807580
4128285]
  4 [128.67584205184852, 128.6758434326619] [0.013590452020993682, 0.001880481
3080444062]
  5 [128.67584345879447, 128.67584346029523] [0.0004477881791293326, 5.687539
374787123e-5]
  6 [128.675843460317, 128.67584346032046] [1.4803232692435838e-5, 2.625540109
0050515e-6]
  7 [128.67584346031973, 128.67584346032027] [4.474440826730428e-7, 2.31157120
76100616e-7]
```

(b)

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

```
function run_1b(n)
    @show n
    A = matrixdepot("poisson", n)
    σ = 1.0
    A_shifted = A - σ * I

    max_subspace_dim = 5

    V = randn(size(A, 2), max_subspace_dim)
    V = ortho_qr(V)
    results = []
    for subspace_dim in 2:max_subspace_dim
        @show subspace_dim
        res = @time projected_subspace_inverse_iteration(A_shifted; V=V[:,
        1:subspace_dim], verbose=false)
        push!(results, (subspace_dim, res))
    end
    results
end
```

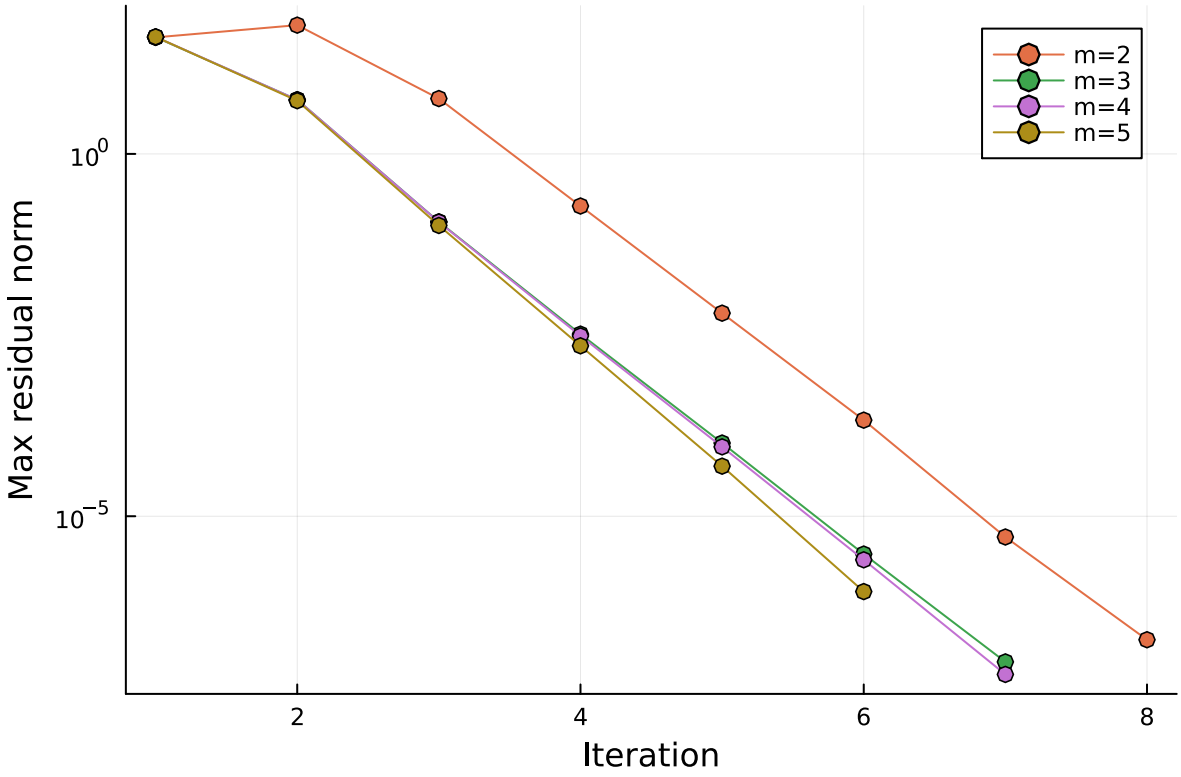
```
results_1b =
  Dict(50 => [(2, (λ = [   more], X = 2500×2 Matrix{Float64}:   , eigenvalues = [   more], resi
                                    0.0218524   -0.0037468
```

```
1 results_1b = Dict(n=>run_1b(n) for n in (10, 20, 30, 50, 100))
```

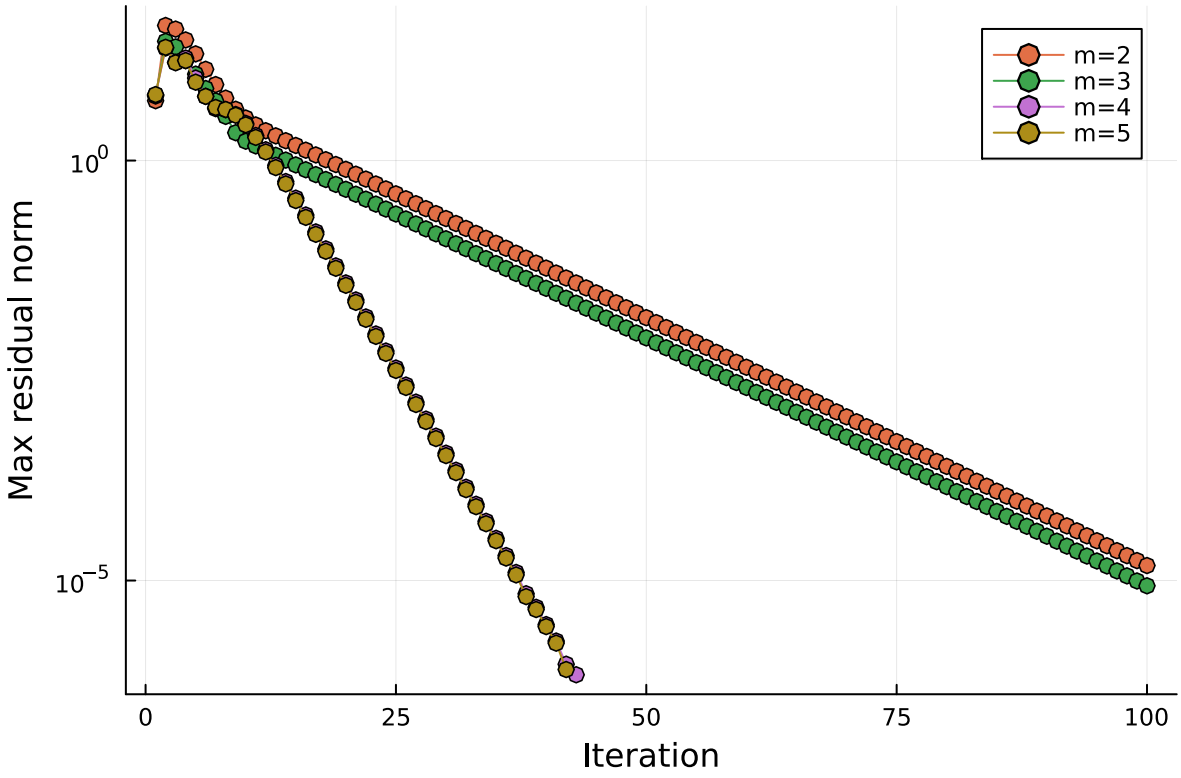
```
n = 10
subspace_dim = 2
  0.001913 seconds (1.23 k allocations: 2.100 MiB)
subspace_dim = 3
  0.001670 seconds (1.07 k allocations: 1.852 MiB)
subspace_dim = 4
  0.001750 seconds (1.07 k allocations: 1.879 MiB)
subspace_dim = 5
  0.001569 seconds (920 allocations: 1.637 MiB)
n = 20
subspace_dim = 2
  0.022827 seconds (3.82 k allocations: 23.134 MiB)
subspace_dim = 3
  0.033483 seconds (3.94 k allocations: 21.563 MiB, 29.63% gc time)
subspace_dim = 4
  0.016483 seconds (2.60 k allocations: 16.163 MiB)
subspace_dim = 5
  0.014114 seconds (2.30 k allocations: 14.462 MiB)
n = 30
subspace_dim = 2
  0.194771 seconds (16.74 k allocations: 200.406 MiB, 3.47% gc time)
subspace_dim = 3
  0.192782 seconds (16.98 k allocations: 203.136 MiB, 0.97% gc time)
subspace_dim = 4
  0.085091 seconds (7.44 k allocations: 88.592 MiB, 1.42% gc time)
subspace_dim = 5
  0.084244 seconds (7.09 k allocations: 87.731 MiB, 0.69% gc time)
n = 50
subspace_dim = 2
  0.129123 seconds (4.72 k allocations: 149.274 MiB, 0.88% gc time)
subspace_dim = 3
  0.133081 seconds (4.71 k allocations: 151.124 MiB, 1.16% gc time)
subspace_dim = 4
  0.072396 seconds (2.51 k allocations: 82.516 MiB, 0.70% gc time)
```

```
plot_1b (generic function with 1 method)
```

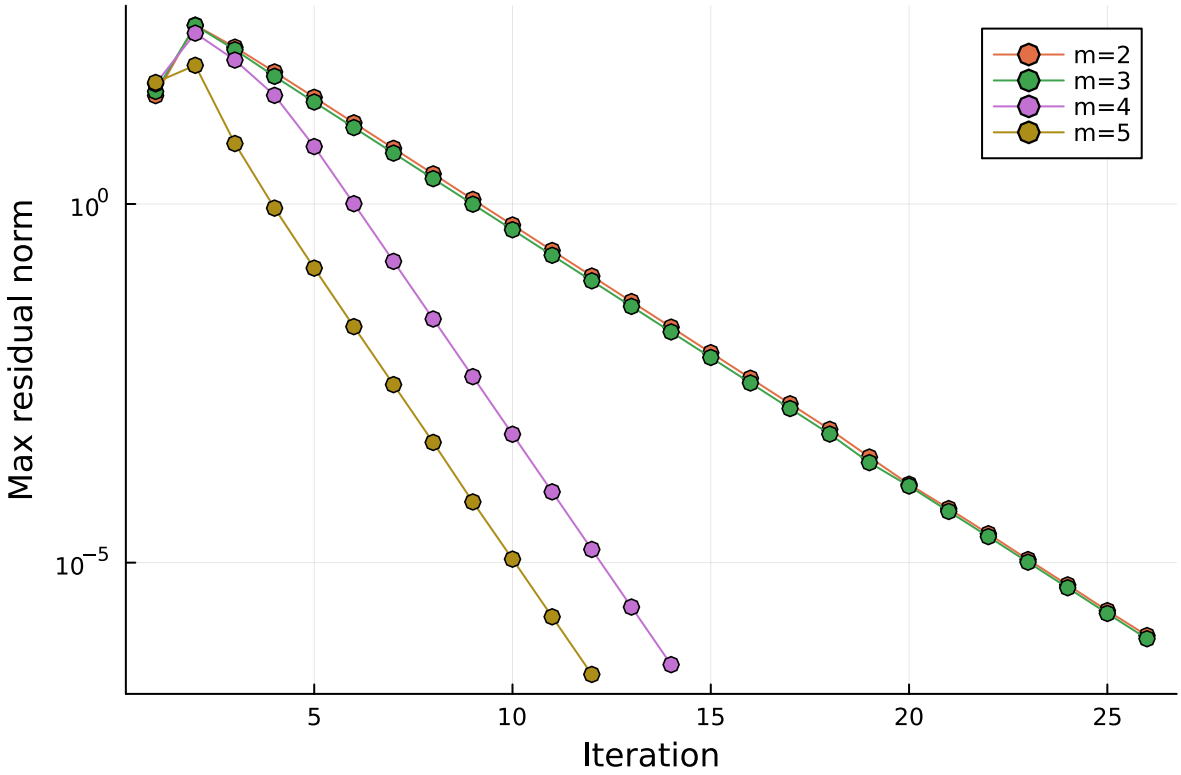
```
1 function plot_1b(results)
2     plt = plot(xlabel="Iteration", ylabel="Max residual norm")
3     for (subspace_dim, res) in results
4         r = maximum.(res.residual_norms)
5         plot!(plt, r, label="m=$subspace_dim", yaxis=:log, m=:o, c=subspace_dim)
6     end
7     plt
8 end
```



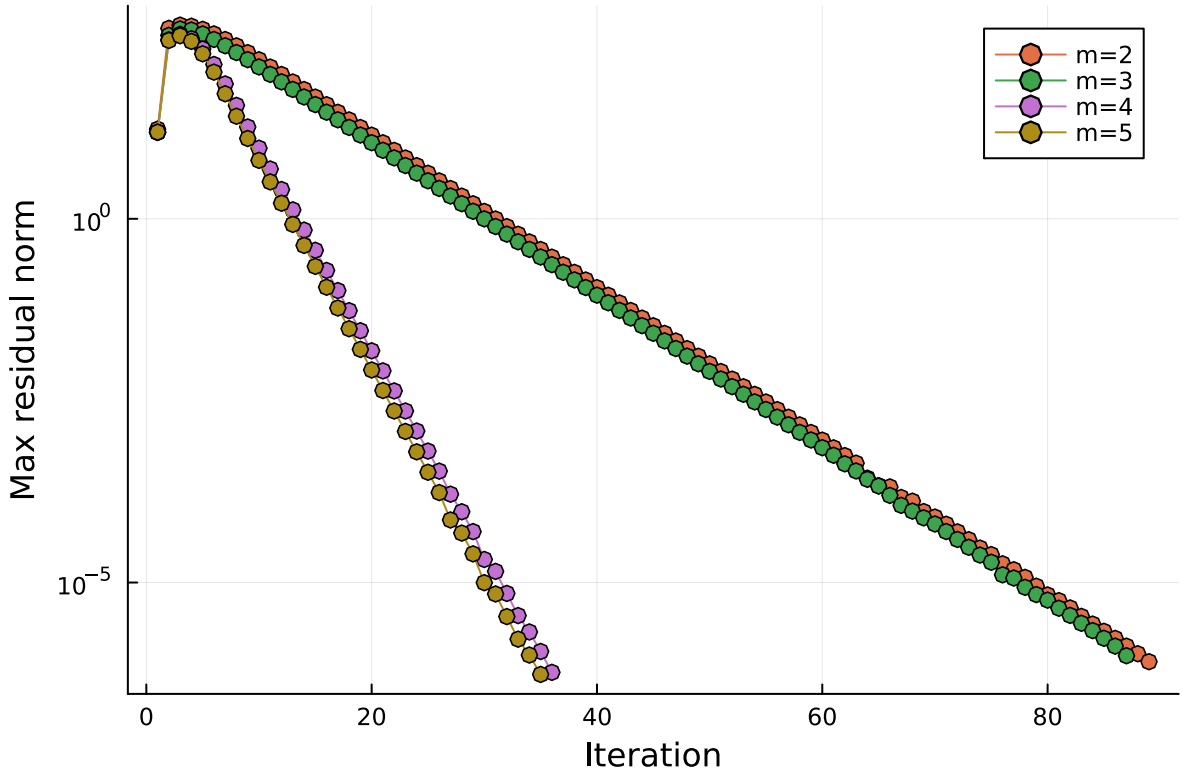

```
1 plot_1b(results_1b[10])
```




```
1 plot_1b(results_1b[30])
```




```
1 plot_1b(results_1b[50])
```




```
1 plot_1b(run_1b(120))
```

```
n = 120
subspace_dim = 2
  2.588281 seconds (15.98 k allocations: 2.996 GiB, 0.35% gc time)
subspace_dim = 3
  2.570902 seconds (15.77 k allocations: 2.966 GiB, 0.07% gc time)
subspace_dim = 4
  1.093133 seconds (6.65 k allocations: 1.243 GiB, 0.12% gc time)
subspace_dim = 5
  1.092434 seconds (6.31 k allocations: 1.224 GiB, 0.06% gc time)
```

(d)

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

```julia
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)

```
([1.00777, 1.00777], 2×2 Matrix{Float64}:  )
                      1.00777  -1.56125e-17
```

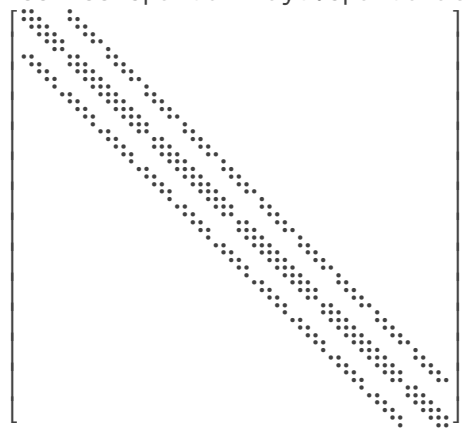
```
let
    σ = 1.0
    (; X) = projected_subspace_inverse_iteration(A - σ * I; maxiter=1)
    println("Switching to dynamic shift algorithm")
    (; λ, X) = projected_subspace_iteration_dynamic(A; V=X)
    λ, X'A*X
end
```

```
  1 [1.1156802155444547, 4.463497899153447] [9.627003137143841, 22.99927844
720948]
Switching to dynamic shift algorithm
  1 [1.0081290971068053, 1.011884468188912] [0.041964864149635045, 0.102481030
36628098]
  2 [1.007771418340094, 1.0077714659147405] [0.0001629832966818973, 5.12633338
10209224e-5]
  3 [1.0077714664470676, 1.0077714664470676] [9.08671580453475e-12, 2.73796564
12872735e-13]
```

```
Enter cell code...
```

Exercise 2

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



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

(a)

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

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

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

(b)

```
100×3 Matrix{Float64}:
 -0.2   4.26326e-16   0.120519
  0.0   0.0           5.55112e-17
  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
```

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
 2.0006837303475127
 5.275577069027194
 5.800051257362885
vectors:
3×3 Matrix{Float64}:
 -0.685007  0.181397  -0.705593
 -0.685496  0.167478   0.708552
  0.2467    0.969044   0.00962309
```

```
λ_til, Y_til = eigen(V'*B*V)
```

```
X_til = 100×3 Matrix{Float64}:
         0.166733      0.0805088     0.142278
         1.36946e-17   5.37927e-17   5.34189e-19
         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
```

```
X_til = V * Y_til
```

(c)

```
λ =
 [0.162028, 0.398507, 0.398507, 0.634986, 0.771293, 0.771293, 1.00777, 1.00777, 1.25018, 1
```

```
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
```

```julia
residuals = B * X_til - X_til * Diagonal(λ_til)
```

```julia
for (λi, xi) in zip(λ_til, eachcol(X_til))
    r = B * xi - λi * xi
    @show minimum(abs, λ .- λi)
    @show norm(r)
    @show minimum(abs, λ .- λi) ≤ norm(r)
end
```

```
minimum(abs, λ .- λi) = 0.032179527443556566
norm(r) = 1.9987068709048952
minimum(abs, λ .- λi) ≤ norm(r) = true
minimum(abs, λ .- λi) = 0.12230032008859393
norm(r) = 1.143271627439608
minimum(abs, λ .- λi) ≤ norm(r) = true
minimum(abs, λ .- λi) = 0.1383912053553411
norm(r) = 0.5991193259712945
minimum(abs, λ .- λi) ≤ norm(r) = true
```

(d)

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

```
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 = [
                                    -0.0265981  -0.0287476   0.0144315
```

```
1 projected_subspace_iteration(B; V=V, maxiter=1000, tol=1e-6)
```

```
363    7.838 1.846e-10
364    7.838 1.735e-10
365    7.838 1.63e-10
366    7.838 1.532e-10
367    7.838 1.439e-10
368    7.838 1.352e-10
369    7.838 1.271e-10
370    7.838 1.194e-10
371    7.838 1.122e-10
372    7.838 1.054e-10
373    7.838 9.906e-11
374    7.838 9.308e-11
375    7.838 8.747e-11
376    7.838 8.219e-11
377    7.838 7.723e-11
378    7.838 7.257e-11
379    7.838 6.819e-11
380    7.838 6.408e-11
381    7.838 6.021e-11
382    7.838 5.658e-11
383    7.838 5.316e-11
384    7.838 4.995e-11
385    7.838 4.694e-11
386    7.838 4.411e-11
387    7.838 4.145e-11
388    7.838 3.895e-11
389    7.838 3.66e-11
390    7.838 3.439e-11
391    7.838 3.231e-11
392    7.838 3.036e-11
393    7.838 2.853e-11
394    7.838 2.681e-11
395    7.838 2.519e-11
396    7.838 2.367e-11
```

(e)

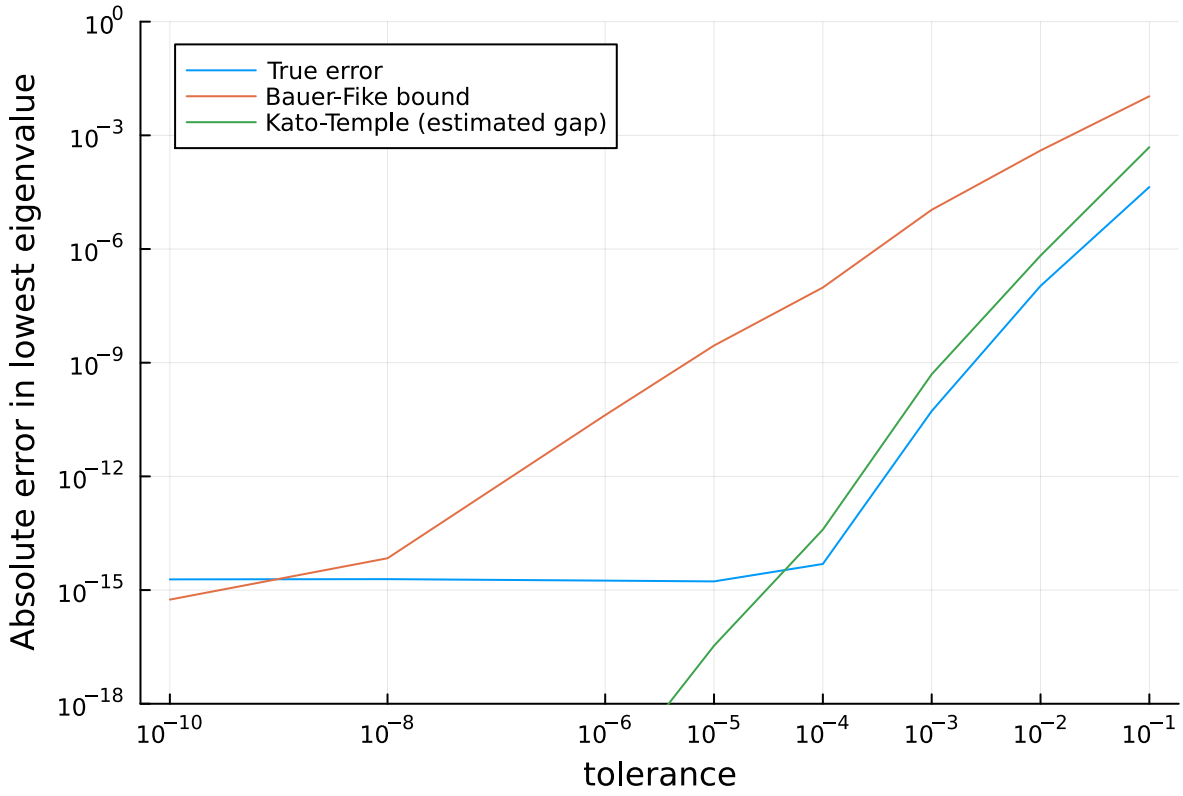
```
lobpcg (generic function with 1 method)
```

```
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
```

```
(λ = [0.162028, 0.398507, 0.398507, 0.634986], X = 100×4 Matrix{Float64}:
                                                    0.0144315   0.0387883   0.00541867
```

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

```
  1    4.934    1.885
  2    2.607    1.247
  3    1.776    1.055
  4    1.278   0.8084
  5   0.9834   0.6463
  6   0.7763   0.4617
  7   0.7042   0.2842
  8   0.6739   0.2016
  9   0.6552   0.1443
 10   0.6458    0.115
 11   0.6399  0.08551
 12   0.6368  0.05677
 13   0.6356  0.03245
 14   0.6353  0.01714
 15   0.6352  0.01013
 16   0.6351 0.009287
 17   0.6351  0.00763
 18    0.635 0.005716
 19    0.635 0.005795
 20    0.635 0.003473
 21    0.635 0.001653
 22    0.635 0.0009035
 23    0.635 0.0004884
 24    0.635 0.0002611
 25    0.635 0.0001938
 26    0.635 0.0001109
 27    0.635 6.896e-05
 28    0.635 4.135e-05
 29    0.635 2.79e-05
 30    0.635 2.41e-05
 31    0.635 2.21e-05
 32    0.635 1.577e-05
 33    0.635 1.109e-05
 34    0.635 1.32e-05
```




```julia
let Pinv = Diagonal(1 ./ diag(B))
    X = randn(size(B, 2), 4)

    tol_range = [1e-10, 1e-8, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1]
    true_error = Float64[]
    bauer_fike = Float64[]
    kato_temple = Float64[]
    for tol in tol_range
        result = lobpcg(B; X, Pinv, tol, maxiter=10_000, verbose=false)

        # Bauer-Fike
        r =  result.residual_norms[end][1]
        push!(bauer_fike, r)

        # Kato-Temple
        gap = abs(result.λ[1] - result.λ[2])
        push!(kato_temple, r^2 / gap)

        # True error
        push!(true_error, abs(λ[1] - result.λ[1]))
    end

    plt = plot(xaxis=:log, yaxis=:log, xlabel="tolerance", ylabel="Absolute error in lowest eigenvalue", legend=:topleft)
    ylims!((1e-18, 1))
    xticks!(tol_range)
    plot!(tol_range, true_error, label="True error")
    plot!(tol_range, bauer_fike, label="Bauer-Fike bound")
    plot!(tol_range, kato_temple, label="Kato-Temple (estimated gap)")
end
```