PracticeSession10

June 10, 2025

Numerical Linear Algebra for Computational Science and Information Engineering

Locally Optimal Block Preconditioned Conjugate Gradient

Nicolas Venkovic (nicolas.venkovic@tum.edu)

[1]:
```
using LinearAlgebra, Plots, Printf, Latexify, LaTeXStrings, BenchmarkTools,␣
  ↪SparseArrays
using MatrixMarket: mmread
```

Exercise #1: Implement Basic_LOBPCG

[2]:
```
function RR(X::AbstractArray{Float64,2},
            AX::AbstractArray{Float64,2},
            BX::AbstractArray{Float64,2})
  # Computes m least dominant generalized eigenpairs of
  # (A,B) w.r.t. Range(X), by Rayleigh Ritz projection

  n, m = size(X)
  G1 = BLAS.gemm('T', 'N', 1., X, AX) # G1 = X'AX
  G2 = BLAS.gemm('T', 'N', 1., X, BX) # G2 = X'BX
  hX = Array{Float64,2}(undef, m, m)
  chol = cholesky(Symmetric(G2))      # Find L, U s.t. X'BX = L * U
  rdiv!(G1, chol.U)
  ldiv!(G2, chol.L, G1)               # G2 = L^(-1) * X'AX * U^(-1)
  Lambda, hY = eigen(Symmetric(G2))   # Solve for Lambda, hY s.t.
                                      # G2 * hY = hY * diagm(Lambda)
  ldiv!(hX, chol.U, hY)               # hX = U^(-1) * hY
  return hX, Lambda
end;
```

[3]:
```
function RR(X::AbstractArray{Float64,2},
            Z::AbstractArray{Float64,2},
            AX::AbstractArray{Float64,2},
            AZ::AbstractArray{Float64,2},
            BZ::AbstractArray{Float64,2},
            check_L_cond=false,
```

```
          Lambda::AbstractVector{Float64}=Float64[])
# Computes m least dominant generalized eigenpairs of
# (A,B) w.r.t. Range([X,Z]), by Rayleigh Ritz projection

# In
# X         : s.t. X'B*X = I

tau = 2 * eps(Float64)

n, m = size(X)
_, q = size(Z)
G1 = Array{Float64,2}(undef, m+q, m+q)
G2 = Array{Float64,2}(undef, m+q, m+q)
VtBV = Array{Float64,2}(undef, m+q, m+q)
hX = Array{Float64,2}(undef, m+q, m)

if isempty(Lambda)
  G1[1:m, 1:m] = BLAS.gemm('T', 'N', 1., X, AX) # X'AX
else
  G1[1:m, 1:m] = diagm(Lambda)
end
G1[1:m, m+1:m+q] = BLAS.gemm('T', 'N', 1., X, AZ) # X'AZ
G1[m+1:m+q, m+1:m+q] = BLAS.gemm('T', 'N', 1., Z, AZ) # Z'AZ
G1[m+1:m+q, 1:m] = G1[1:m, m+1:m+q]' # G1 = [X,Z]'[AX,AZ]

VtBV[1:m, 1:m] = diagm(ones(m))
VtBV[1:m, m+1:m+q] = BLAS.gemm('T', 'N', 1., X, BZ) # X'BZ
VtBV[m+1:m+q, m+1:m+q] = BLAS.gemm('T', 'N', 1., Z, BZ) # Z'BZ
VtBV[m+1:m+q, 1:m] = VtBV[1:m, m+1:m+q]' # VtBV = [X,Z]'[BX,BZ]

D = diagm(diag(VtBV).^(-.5))
G2 .= D * VtBV * D

chol = cholesky(Symmetric(G2)) # Find L, U s.t. D * VtBV * D = L * U

if check_L_cond
  if cond(chol.L)^(-3) < tau
    return hX, zeros(m), false, VtBV
  end
end

G1 .= D * G1 * D
rdiv!(G1, chol.U)
ldiv!(G2, chol.L, G1)              # G2 = L^(-1) * D * V'A*V * D * U^(-1)
Lambda, hY = eigen(Symmetric(G2)) # Solve for Lambda, hY s.t.
                                  # G2 * hY = hY * diagm(Lambda)
ldiv!(hX, chol.U, hY[:, 1:m])
```

```julia
  hX .= D * hX                        # hX = D * U^(-1) * hY

  if check_L_cond
    return hX, Lambda[1:m], true, VtBV
  else
    return hX, Lambda[1:m]
  end
end;
```

[4]:
```julia
function RR(X::AbstractArray{Float64,2},
            Z::AbstractArray{Float64,2},
            P::AbstractArray{Float64,2},
            AX::AbstractArray{Float64,2},
            AZ::AbstractArray{Float64,2},
            AP::AbstractArray{Float64,2},
            BZ::AbstractArray{Float64,2},
            BP::AbstractArray{Float64,2},
            check_L_cond=false,
            Lambda::AbstractVector{Float64}=Float64[])
  # Computes m least dominant generalized eigenpairs of
  # (A,B) w.r.t. Range([X,Z,P]), by Rayleigh Ritz projection

  # In
  # X          : s.t. X'B*X = I

  tau = 2 * eps(Float64)

  n, m = size(X)
  _, q = size(Z)
  G1 = Array{Float64,2}(undef, m+2*q, m+2*q)
  G2 = Array{Float64,2}(undef, m+2*q, m+2*q)
  VtBV = Array{Float64,2}(undef, m+2*q, m+2*q)
  hX = Array{Float64,2}(undef, m+2*q, m)

  if isempty(Lambda)
    G1[1:m, 1:m] = BLAS.gemm('T', 'N', 1., X, AX) # X'AX
  else
    G1[1:m, 1:m] = diagm(Lambda)
  end
  G1[1:m, m+1:m+q] = BLAS.gemm('T', 'N', 1., X, AZ) # X'AZ
  G1[1:m, m+q+1:m+2*q] = BLAS.gemm('T', 'N', 1., X, AP) # X'AP
  G1[m+1:m+q, m+1:m+q] = BLAS.gemm('T', 'N', 1., Z, AZ) # Z'AZ
  G1[m+1:m+q, m+q+1:m+2*q] = BLAS.gemm('T', 'N', 1., Z, AP) # Z'AP
  G1[m+q+1:m+2*q, m+q+1:m+2*q] = BLAS.gemm('T', 'N', 1., P, AP) # P'AP
  G1[m+1:m+q, 1:m] = G1[1:m, m+1:m+q]'
  G1[m+q+1:m+2*q, 1:m] = G1[1:m, m+q+1:m+2*q]'
  G1[m+q+1:m+2*q, m+1:m+q] = G1[m+1:m+q, m+q+1:m+2*q]' # G1 = [X,Z,P]'[AX,AZ,AP]
```

```julia
    VtBV[1:m, 1:m] = diagm(ones(m))
    VtBV[1:m, m+1:m+q] = BLAS.gemm('T', 'N', 1., X, BZ) # X'BZ
    VtBV[1:m, m+q+1:m+2*q] = BLAS.gemm('T', 'N', 1., X, BP) # X'BP
    VtBV[m+1:m+q, m+1:m+q] = BLAS.gemm('T', 'N', 1., Z, BZ) # Z'BZ
    VtBV[m+1:m+q, m+q+1:m+2*q] = BLAS.gemm('T', 'N', 1., Z, BP) # Z'BP
    VtBV[m+q+1:m+2*q, m+q+1:m+2*q] = BLAS.gemm('T', 'N', 1., P, BP) # P'BP
    VtBV[m+1:m+q, 1:m] = VtBV[1:m, m+1:m+q]'
    VtBV[m+q+1:m+2*q, 1:m] = VtBV[1:m, m+q+1:m+2*q]'
    VtBV[m+q+1:m+2*q, m+1:m+q] = VtBV[m+1:m+q, m+q+1:m+2*q]' # VtBV =␣
 ↪[X,Z,P]'[AX,AZ,AP]

    D = diagm(diag(VtBV).^(-.5))
    G2 .= D * VtBV * D

    chol = cholesky(Symmetric(G2)) # Find L, U s.t. D * VtBV * D = L * U

    if check_L_cond
      if cond(chol.L)^(-3) < tau
        return hX, zeros(m), false, VtBV
      end
    end

    G1 .= D * G1 * D
    rdiv!(G1, chol.U)
    ldiv!(G2, chol.L, G1) # G2 = L^(-1) * D * V'A*V * D * U^(-1)
    Lambda, hY = eigen(Symmetric(G2)) # Solve for Lambda, hY s.t.
                                      # G2 * hY = hY * diagm(Lambda)
    ldiv!(hX, chol.U, hY[:, 1:m])
    hX .= D * hX                      # hX = D * U^(-1) * hY

    if check_L_cond
      return hX, Lambda[1:m], true, VtBV

    else
      return hX, Lambda[1:m]
    end
end;
```

[5]:
```julia
function Basic_LOBPCG(A, B, X0, nev::Int;
                      T=I, itmax::Int=200, tol::Float64=1e-6,
                      A_products::Symbol=:implicit,
                      B_products::Symbol=:implicit,
                      debug::Bool=false)
    # Knyazev, A. V. (2001)
    # Toward the optimal preconditioned eigensolver: Locally optimal block␣
 ↪preconditioned conjugate gradient method
```

```
# SIAM journal on scientific computing, 23(2), 517-541

# In
# A          : left  hand-side operator, symmetric positive definite, n-by-n
# B          : right hand-side operator, symmetric positive definite, n-by-n
# X0         : initial iterates, n-by-m (m < n)
# nev        : number of wanted eigenpairs, nev <= m
# T          : precondontioner, symmetric positive definite, n-by-n
# itmax      : maximum number of iterations
# tol        : tolerance used for convergence criterion
# A_products: if :implicit, the matrix products with A are updated implicitly
# B_products: if :implicit, the matrix products with B are updated implicitly
# debug      : if true, and A_products == :implicit, prints out error norm of
#              implicit product updates

# Out
# Lambda: last iterates of least dominant eigenvalues, m-by-1
# X     : last iterates of least dominant eigenvectors, n-by-m
# res   : normalized norms of eigenresiduals, m-by-it

n, m = size(X0)

X = Array{Float64,2}(undef, n, m)
R = Array{Float64,2}(undef, n, m)
Z = Array{Float64,2}(undef, n, m)
P = Array{Float64,2}(undef, n, m)
W = Array{Float64,2}(undef, n, m)
AX = Array{Float64,2}(undef, n, m)
AZ = Array{Float64,2}(undef, n, m)
AP = Array{Float64,2}(undef, n, m)
BX = Array{Float64,2}(undef, n, m)
BZ = Array{Float64,2}(undef, n, m)
BP = Array{Float64,2}(undef, n, m)

res = Array{Float64,2}(undef, m, itmax+1)
k = 0

copy!(X, X0)
mul!(AX, A, X) # AX .= A * X
mul!(BX, B, X) # BX .= B * X
hX, Lambda = RR(X, AX, BX)
mul!(W, X, hX); copy!(X, W) # X .= X * hX
if A_products == :implicit
  mul!(W, AX, hX); copy!(AX, W) # AX .= AX * hX
else
  mul!(AX, A, X) # AX .= A * X
end
```

```
if B_products == :implicit
  mul!(W, BX, hX); copy!(BX, W) # BX .= BX * hX
else
  mul!(BX, B, X) # BX .= B * X
end
# R .= AX - BX * diagm(Lambda)
copy!(R, AX); mul!(R, BX, diagm(Lambda), -1, 1)
res[:, 1] = norm.(eachcol(R))
res[:, 1] ./= abs.(Lambda)
for i in k+1:nev
  res[i, 1] < tol ? k += 1 : break
end
println("it = 0, k = $k")
println("extrema(res) = ", extrema(res[:, 1]))
_AX = view(AX, :, k+1:m)
_R = view(R, :, k+1:m); _W = view(W, :, k+1:m)
_Z = view(Z, :, k+1:m); _AZ = view(AZ, :, k+1:m); _BZ = view(BZ, :, k+1:m)
_P = view(P, :, k+1:m); _AP = view(AP, :, k+1:m); _BP = view(BP, :, k+1:m)
if k < nev
  for j in 1:itmax
    println("it = $j, k = $k")
    ldiv!(_Z, T, _R) # _Z .= T \ _R
    mul!(_AZ, A, _Z) # _AZ .= A * _Z
    mul!(_BZ, B, _Z) # _BZ .= B * _Z
    if j == 1
      hX, Lambda = RR(X, _Z, AX, _AZ, _BZ)
      hX_X = view(hX, 1:m, :); hX_Z = view(hX, m+1:2*m-k, :)
      _hX_Z = view(hX_Z, :, k+1:m)
      mul!(_P, _Z, _hX_Z) # _P .= _Z * _hX_Z
      if A_products == :implicit
        mul!(_AP, _AZ, _hX_Z) # _AP .= _AZ * _hX_Z
      else
        mul!(_AP, A, _P) # _AP .= A * _P
      end
      if B_products == :implicit
        mul!(_BP, _BZ, _hX_Z) # _BP .= _BZ * _hX_Z
      else
        mul!(_BP, B, _P) # _BP .= B * _P
      end
    else
      hX, Lambda = RR(X, _Z, _P, AX, _AZ, _AP, _BZ, _BP)
      hX_X = view(hX, 1:m, :); hX_Z = view(hX, m+1:2*m-k, :)
      hX_P = view(hX, 2*m-k+1:3*m-2*k, :)
      _hX_Z = view(hX_Z, :, k+1:m); _hX_P = view(hX_P, :, k+1:m)
      # _P .= _Z * _hX_Z + _P * _hX_P
      mul!(_W, _P, _hX_P)
      mul!(_W, _Z, _hX_Z, 1, 1)
```

```
    copy!(_P, _W)
    if A_products == :implicit
      # _AP .= _AZ * _hX_Z + _AP * _hX_P
      mul!(_W, _AP, _hX_P)
      mul!(_W, _AZ, _hX_Z, 1, 1)
      copy!(_AP, _W)
    else
      mul!(_AP, A, _P) # _AP .= A * _P
    end
    if B_products == :implicit
      # _BP .= _BZ * _hX_Z + _BP * _hX_P
      mul!(_W, _BP, _hX_P)
      mul!(_W, _BZ, _hX_Z, 1, 1)
      copy!(_BP, _W)
    else
      mul!(_BP, B, _P) # _BP .= B * _P
    end
  end
  if k > 0
    if j == 1
      # X .= X * hX_X + _Z * hX_Z
      mul!(W, _Z, hX_Z)
      mul!(W, X, hX_X, 1, 1)
      copy!(X, W)
    else
      # X .= X * hX_X + _Z * hX_Z + _P * hX_P
      mul!(W, _P, hX_P)
      mul!(W, _Z, hX_Z, 1, 1)
      mul!(W, X, hX_X, 1, 1)
      copy!(X, W)
    end
    mul!(AX, A, X) # AX .= A * X
    mul!(BX, B, X) # BX .= B * X
  else
    # X .= P + X * hX_X
    copy!(W, P)
    mul!(W, X, hX_X, 1, 1)
    copy!(X, W)
    if A_products == :implicit
      # AX .= AP + AX * hX_X
      copy!(W, AP)
      mul!(W, AX, hX_X, 1, 1)
      copy!(AX, W)
    else
      mul!(AX, A, X) # AX .= A * X
    end
    if B_products == :implicit
```

```
        # BX .= BP + BX * hX_X
        copy!(W, BP)
        mul!(W, BX, hX_X, 1, 1)
        copy!(BX, W)
      else
        mul!(BX, B, X)
      end
    end
    # R .= AX - BX * diagm(Lambda)
    copy!(R, AX); mul!(R, BX, diagm(Lambda), -1, 1)
    res[:, j+1] = norm.(eachcol(R))
    res[:, j+1] ./= abs.(Lambda)
    if debug
      if A_products == :implicit
        println("||AX-A*X|| = ", norm(AX - A * X))
        println("||AP-A*P|| = ", norm(AP - A * P))
      end
      if B_products == :implicit
        println("||BX-B*X|| = ", norm(BX - B * X))
        println("||BP-B*P|| = ", norm(BP - B * P))
      end
    end
    println("extrema(res) = ", extrema(res[:, j+1]))
    for i in k+1:nev
      res[i, j+1] < tol ? k += 1 : break
    end
    k < nev ? nothing : return Lambda, X, res[:, 1:j+1]
    if m - k < size(_R)[2]
      _AX = view(AX, :, k+1:m)
      _R = view(R, :, k+1:m); _W = view(W, :, k+1:m)
      _Z = view(Z, :, k+1:m); _AZ = view(AZ, :, k+1:m); _BZ = view(BZ, :, k+1:
↪m)
      _P = view(P, :, k+1:m); _AP = view(AP, :, k+1:m); _BP = view(BP, :, k+1:
↪m)
    end
  end
end
return Lambda, X, res
end;
```

Exercise #2: Implement BLOPEX_LOBPCG

[6]:
```
function RR(X::AbstractArray{Float64,2},
            AX::AbstractArray{Float64,2})
  # Computes m least dominant generalized eigenpairs of
  # (A,B) w.r.t. Range(X), by Rayleigh Ritz projection
```

```
  n, k = size(X)
  G = BLAS.gemm('T', 'N', 1., X, AX) # G = X'AX

  Lambda, hX = eigen(Symmetric(G))   # Solve for Lambda, hX s.t.
                                     # G * hX = hX * diagm(Lambda)
  return hX, Lambda
end;
```

[7]:
```
function RR_BLOPEX(X::AbstractArray{Float64,2},
                   Z::AbstractArray{Float64,2},
                   AX::AbstractArray{Float64,2},
                   AZ::AbstractArray{Float64,2},
                   BZ::AbstractArray{Float64,2},
                   Lambda::AbstractVector{Float64}=Float64[])
  # Computes m least dominant generalized eigenpairs of
  # (A,B) w.r.t. Range([X,Z]), by Rayleigh Ritz projection

  # In
  # X          : s.t. X'B*X = I

  n, m = size(X)
  G1 = Array{Float64,2}(undef, 2*m, 2*m)
  G2 = Array{Float64,2}(undef, 2*m, 2*m)
  hX = Array{Float64,2}(undef, 2*m, m)

  if isempty(Lambda)
    G1[1:m, m+1:2*m] = BLAS.gemm('T', 'N', 1., X, AX) # X'AX
  else
    G1[1:m, 1:m] = diagm(Lambda)
  end
  G1[1:m, m+1:2*m] = BLAS.gemm('T', 'N', 1., X, AZ) # X'AZ
  G1[m+1:2*m, m+1:2*m] = BLAS.gemm('T', 'N', 1., Z, AZ) # Z'AZ
  G1[m+1:2*m, 1:m] = G1[1:m, m+1:2*m]' # G1 = [X,Z]'[AX,AZ]

  G2[1:m, 1:m] = diagm(ones(m))
  G2[1:m, m+1:2*m] = BLAS.gemm('T', 'N', 1., X, BZ) # X'BZ
  G2[m+1:2*m, m+1:2*m] = diagm(ones(m))
  G2[m+1:2*m, 1:m] = G2[1:m, m+1:2*m]' # G2 = [X,Z]'[BX,BZ]

  chol = cholesky(Symmetric(G2))    # Find L, U s.t. [X,Z]'[BX,BZ] = L * U
  rdiv!(G1, chol.U)
  ldiv!(G2, chol.L, G1)             # G2 = L^(-1) * [X,Z]'[AX,AZ] * U^(-1)
  Lambda, hY = eigen(Symmetric(G2)) # Solve for Lambda, hY s.t.
                                    # G2 * hY = hY * diagm(Lambda)
  ldiv!(hX, chol.U, hY[:, 1:m])     # hX = U^(-1) * hY
  return hX, Lambda[1:m]
end;
```

[8]:
```julia
function RR_BLOPEX(X::AbstractArray{Float64,2},
                   Z::AbstractArray{Float64,2},
                   P::AbstractArray{Float64,2},
                   AX::AbstractArray{Float64,2},
                   AZ::AbstractArray{Float64,2},
                   AP::AbstractArray{Float64,2},
                   BZ::AbstractArray{Float64,2},
                   BP::AbstractArray{Float64,2},
                   Lambda::AbstractVector{Float64}=Float64[])
  # Computes m least dominant generalized eigenpairs of
  # (A,B) w.r.t. Range([X,Z,P]), by Rayleigh Ritz projection

  # In
  # X         : s.t. X'B*X = I

  n, m = size(X)
  G1 = Array{Float64,2}(undef, 3*m, 3*m)
  G2 = Array{Float64,2}(undef, 3*m, 3*m)
  hX = Array{Float64,2}(undef, 3*m, m)

  if isempty(Lambda)
    G1[1:m, 1:m] = BLAS.gemm('T', 'N', 1., X, AX) # X'AX
  else
    G1[1:m, 1:m] = diagm(Lambda)
  end
  G1[1:m, m+1:2*m] = BLAS.gemm('T', 'N', 1., X, AZ) # X'AZ
  G1[1:m, 2*m+1:3*m] = BLAS.gemm('T', 'N', 1., X, AP) # X'AP
  G1[m+1:2*m, m+1:2*m] = BLAS.gemm('T', 'N', 1., Z, AZ) # Z'AZ
  G1[m+1:2*m, 2*m+1:3*m] = BLAS.gemm('T', 'N', 1., Z, AP) # Z'AP
  G1[2*m+1:3*m, 2*m+1:3*m] = BLAS.gemm('T', 'N', 1., P, AP) # P'AP
  G1[m+1:2*m, 1:m] = G1[1:m, m+1:2*m]'
  G1[2*m+1:3*m, 1:m] = G1[1:m, 2*m+1:3*m]'
  G1[2*m+1:3*m, m+1:2*m] = G1[m+1:2*m, 2*m+1:3*m]' # G1 = [X,Z,P]'[AX,AZ,AP]

  G2[1:m, 1:m] = diagm(ones(m))
  G2[m+1:2*m, m+1:2*m] = diagm(ones(m))
  G2[2*m+1:3*m, 2*m+1:3*m] = diagm(ones(m))
  G2[1:m, m+1:2*m] = BLAS.gemm('T', 'N', 1., X, BZ) # X'BZ
  G2[1:m, 2*m+1:3*m] = BLAS.gemm('T', 'N', 1., X, BP) # X'BP
  G2[m+1:2*m, 2*m+1:3*m] = BLAS.gemm('T', 'N', 1., Z, BP) # Z'BP
  G2[m+1:2*m, 1:m] = G2[1:m, m+1:2*m]'
  G2[2*m+1:3*m, 1:m] = G2[1:m, 2*m+1:3*m]'
  G2[2*m+1:3*m, m+1:2*m] = G2[m+1:2*m, 2*m+1:3*m]' # G2 = [X,Z,P]'[AX,AZ,AP]

  chol = cholesky(Symmetric(G2))    # Find L, U s.t. [X,Z,P]'[BX,BZ,BP] = L * U
  rdiv!(G1, chol.U)
  ldiv!(G2, chol.L, G1)             # G2 = L^(-1) * [X,Z,P]'[AX,AZ,AP] * U^(-1)
```

```
  Lambda, hY = eigen(Symmetric(G2)) # Solve for Lambda, hY s.t.
                                    # G2 * hY = hY * diagm(Lambda)
  ldiv!(hX, chol.U, hY[:, 1:m])     # hX = U^(-1) * hY

  return hX, Lambda[1:m]
end;
```

[9]:
```
function BLOPEX_LOBPCG(A, B, X0, nev::Int;
                       T=I, itmax::Int=200, tol::Float64=1e-6,
                       A_products::Symbol=:implicit,
                       B_products::Symbol=:implicit,
                       debug::Bool=false)
  # Knyazev, A. V., Argentati, M. E., Lashuk, I., & Ovtchinnikov, E. E. (2007)
  # Block locally optimal preconditioned eigenvalue Xolvers (BLOPEX) in Hypre␣
↪and PETSc
  # SIAM Journal on Scientific Computing, 29(5), 2224-2239.

  # In
  # A          : left  hand-side operator, symmetric positive definite, n-by-n
  # B          : right hand-side operator, symmetric positive definite, n-by-n
  # X0         : initial iterates, n-by-m (m < n)
  # nev        : number of wanted eigenpairs, nev <= m
  # T          : precondontioner, symmetric positive definite, n-by-n
  # itmax      : maximum number of iterations
  # tol        : tolerance used for convergence criterion
  # A_products: if :implicit, the matrix products with A are updated implicitly
  # B_products: if :implicit, the matrix products with B are updated implicitly
  # debug      : if true, and A_products == :implicit, prints out error norm of
  #              implicit product updates

  # Out
  # Lambda: last iterates of least dominant eigenvalues, m-by-1
  # X     : last iterates of least dominant eigenvectors, n-by-m
  # res   : normalized norms of eigenresiduals, m-by-it

  n, m = size(X0)

  X = Array{Float64,2}(undef, n, m)
  R = Array{Float64,2}(undef, n, m)
  Z = Array{Float64,2}(undef, n, m)
  P = Array{Float64,2}(undef, n, m)
  W = Array{Float64,2}(undef, n, m)
  AX = Array{Float64,2}(undef, n, m)
  AZ = Array{Float64,2}(undef, n, m)
  AP = Array{Float64,2}(undef, n, m)
  BX = Array{Float64,2}(undef, n, m)
  BZ = Array{Float64,2}(undef, n, m)
```

```julia
BP = Array{Float64,2}(undef, n, m)

res = Array{Float64,2}(undef, m, itmax+1)
k = 0

copy!(X, X0)
mul!(BX, B, X) # BX .= B * X
XtBX = BLAS.gemm('T', 'N', 1., X, BX) # XtBX = X'BX
chol = cholesky(Symmetric(XtBX))
rdiv!(X, chol.U) # X .= X * L^(-T)
if B_products == :implicit
  rdiv!(BX, chol.U) # BX .= BX * L^(-T)
else
  mul!(BX, B, X) # BX .= B * X
end
mul!(AX, A, X) # AX = A * X
hX, Lambda = RR(X, AX)
mul!(W, X, hX); copy!(X, W) # X .= X * hX
if A_products == :implicit
  mul!(W, AX, hX); copy!(AX, W) # AX .= AX * hX
else
  mul!(AX, A, X) # AX .= A * X
end
if B_products == :implicit
  mul!(W, BX, hX); copy!(BX, W) # BX .= BX * hX
else
  mul!(BX, B, X) # BX .= B * X
end

# R .= AX - BX * diagm(Lambda)
copy!(R, AX); mul!(R, BX, diagm(Lambda), -1, 1)
res[:, 1] = norm.(eachcol(R))
res[:, 1] ./= abs.(Lambda)
for i in k+1:nev
  res[i, 1] < tol ? k += 1 : break
end
println("it = 0, k = $k")
println("extrema(res) = ", extrema(res[:, 1]))

if k < nev
  for j in 1:itmax
    println("it = $j, k = $k")
    ldiv!(Z, T, R) # Z .= T \ R
    mul!(BZ, B, Z) # BZ .= B * Z
    ZtBZ = BLAS.gemm('T', 'N', 1., Z, BZ) # ZtBZ = Z'BZ
    chol = cholesky(Symmetric(ZtBZ))
    rdiv!(Z, chol.U) # Z .= Z * L^(-T)
```

```
    if B_products == :implicit
      rdiv!(BZ, chol.U) # BZ .= BZ * L^(-T)
    else
      mul!(BZ, B, Z) # BZ .= B * Z
    end
    mul!(AZ, A, Z) # AZ .= A * Z
    if j == 1
      hX, Lambda = RR_BLOPEX(X, Z, AX, AZ, BZ)
      hX_X = view(hX, 1:m, :); hX_Z = view(hX, m+1:2*m, :)
      mul!(P, Z, hX_Z) # P .= Z * hX_Z
      if A_products == :implicit
        mul!(AP, AZ, hX_Z) # AP .= AZ * hX_Z
      else
        mul!(AP, A, P) # AP .= A * P
      end
      if B_products == :implicit
        mul!(BP, BZ, hX_Z) # BP .= BZ * hX_Z
      else
        mul!(BP, B, P) # BP .= B * P
      end
    else
      PtBP = BLAS.gemm('T', 'N', 1., P, BP) # PtBP = P'BP
      chol = cholesky(Symmetric(PtBP))
      rdiv!(P, chol.U) # P .= P * L^(-T)
      if A_products == :implicit
        rdiv!(AP, chol.U) # AP .= AP * L^(-T)
      else
        mul!(AP, A, P) # AP .= A * P
      end
      if B_products == :implicit
        rdiv!(BP, chol.U) # BP .= BP * L^(-T)
      else
        mul!(BP, B, P) # BP .= B * P
      end
      hX, Lambda = RR_BLOPEX(X, Z, P, AX, AZ, AP, BZ, BP)
      hX_X = view(hX, 1:m, :); hX_Z = view(hX, m+1:2*m, :)
      hX_P = view(hX, 2*m+1:3*m, :)
      # P .= Z * hX_Z + P * hX_P
      mul!(W, P, hX_P)
      mul!(W, Z, hX_Z, 1, 1)
      copy!(P, W)
      if A_products == :implicit
        # AP .= AZ * hX_Z + AP * hX_P
        mul!(W, AP, hX_P)
        mul!(W, AZ, hX_Z, 1, 1)
        copy!(AP, W)
      else
```

```
      mul!(AP, A, P) # AP .= A * P
    end
    if B_products == :implicit
      # BP .= BZ * hX_Z + BP * hX_P
      mul!(W, BP, hX_P)
      mul!(W, BZ, hX_Z, 1, 1)
      copy!(BP, W)
    else
      mul!(BP, B, P) # BP .= B * P
    end
  end
  # X .= P + X * hX_X
  copy!(W, P)
  mul!(W, X, hX_X, 1, 1)
  copy!(X, W)
  if A_products == :implicit
    # AX .= AP + AX * hX_X
    copy!(W, AP)
    mul!(W, AX, hX_X, 1, 1)
    copy!(AX, W)
  else
    mul!(AX, A, X) # AX .= A * X
  end
  if B_products == :implicit
    # BX .= BP + BX * hX_X
    copy!(W, BP)
    mul!(W, BX, hX_X, 1, 1)
    copy!(BX, W)
  else
    mul!(BX, B, X) # BX .= B * X
  end
  # R .= AX - BX * diagm(Lambda)
  copy!(R, AX); mul!(R, BX, diagm(Lambda), -1, 1)
  res[:, j+1] = norm.(eachcol(R))
  res[:, j+1] ./= abs.(Lambda)
  if debug
    if A_products == :implicit
      println("||AX-A*X|| = ", norm(AX - A * X))
      println("||AP-A*P|| = ", norm(AP - A * P))
    end
    if B_products == :implicit
      println("||BX-B*X|| = ", norm(BX - B * X))
      println("||BP-B*P|| = ", norm(BP - B * P))
    end
  end
  println("extrema(res) = ", extrema(res[:, j+1]))
  for i in k+1:nev
```

```
        res[i, j+1] < tol ? k += 1 : break
      end
      k < nev ? nothing : return Lambda, X, res[:, 1:j+1]
    end
  end
  return Lambda, X, res
end;
```

Exercise #5: Implement Ortho_LOBPCG

[10]:
```
function RR(X::AbstractArray{Float64,2},
            Z::AbstractArray{Float64,2},
            AX::AbstractArray{Float64,2},
            AZ::AbstractArray{Float64,2},
            Lambda::AbstractVector{Float64}=Float64[],
            modified::Bool=false)
  # Computes m least dominant generalized eigenpairs of
  # (A,B) w.r.t. Range([X,Z]), by Rayleigh Ritz projection

  # In
  # X, Z   : s.t. [X,Z]'B*[X,Z] = I

  n, m = size(X)
  _, q = size(Z)
  G = Array{Float64,2}(undef, m+q, m+q)
  hX = Array{Float64,2}(undef, m+q, m)

  if isempty(Lambda)
    G[1:m, 1:m] = BLAS.gemm('T', 'N', 1., X, AX)
  else
    G[1:m, 1:m] = diagm(Lambda)
  end
  G[1:m, m+1:m+q] = BLAS.gemm('T', 'N', 1., X, AZ)
  G[m+1:m+q, m+1:m+q] = BLAS.gemm('T', 'N', 1., Z, AZ)
  G[m+1:m+q, 1:m] = G[1:m, m+1:m+q]' # G = [X,Z]'[AX,AZ]

  Lambda, hZ = eigen(Symmetric(G))

  if modified
    hQt, _ = qr(hZ[1:m, m+1:end]')
    hY = hZ[:, m+1:end] * Matrix(hQt)
    return hZ[:, 1:m], Lambda[1:m], hY
  end

  return hZ[:, 1:m], Lambda[1:m]
end;
```

```
[11]: function RR(X::AbstractArray{Float64,2},
                  Z::AbstractArray{Float64,2},
                  P::AbstractArray{Float64,2},
                  AX::AbstractArray{Float64,2},
                  AZ::AbstractArray{Float64,2},
                  AP::AbstractArray{Float64,2},
                  Lambda::AbstractVector{Float64}=Float64[],
                  modified::Bool=false)
        # Computes m least dominant generalized eigenpairs of
        # (A,B) w.r.t. Range([X,Z,P]), by Rayleigh Ritz projection

        # In
        # X       : s.t. [X,Z,P]'B*[X,Z,P] = I

        n, m = size(X)
        _, q = size(Z)
        G = Array{Float64,2}(undef, m+2*q, m+2*q)

        if isempty(Lambda)
          G[1:m, 1:m] = BLAS.gemm('T', 'N', 1., X, AX) # X'AX
        else
          G[1:m, 1:m] = diagm(Lambda)
        end

        G[1:m, m+1:m+q] = BLAS.gemm('T', 'N', 1., X, AZ) # X'AZ
        G[1:m, m+q+1:m+2*q] = BLAS.gemm('T', 'N', 1., X, AP) # X'AP
        G[m+1:m+q, m+1:m+q] = BLAS.gemm('T', 'N', 1., Z, AZ) # Z'AZ
        G[m+1:m+q, m+q+1:m+2*q] = BLAS.gemm('T', 'N', 1., Z, AP) # Z'AP
        G[m+q+1:m+2*q, m+q+1:m+2*q] = BLAS.gemm('T', 'N', 1., P, AP) # P'AP
        G[m+1:m+q, 1:m] = G[1:m, m+1:m+q]'
        G[m+q+1:m+2*q, 1:m] = G[1:m, m+q+1:m+2*q]'
        G[m+q+1:m+2*q, m+1:m+q] = G[m+1:m+q, m+q+1:m+2*q]' # G = [X,Z,P]'[AX,AZ,AP]

        Lambda, hZ = eigen(Symmetric(G))

        if modified
          hQt, _ = qr(hZ[1:m, m+1:end]')
          hY = hZ[:, m+1:end] * Matrix(hQt)
          return hZ[:, 1:m], Lambda[1:m], hY
        end

        return hZ[:, 1:m], Lambda[1:m]
      end;
```

```
[12]: function Ortho!(U::AbstractArray{Float64,2},
                      B::AbstractArray{Float64,2},
                      V::AbstractArray{Float64,2},
```

```julia
                 BU::AbstractArray{Float64,2},
                 norm_BV::Float64,
                 tol_ortho::Float64=100*eps(Float64),
                 tol_replace::Float64=10*eps(Float64),
                 itmax1::Int=6, itmax2::Int=6)
  # Stathopoulos, A., & Wu, K. (2002)
  # A block orthogonalization procedure with constant synchronization requirements
  # SIAM Journal on Scientific Computing, 23(6), 2165-2182

  VtBU = BLAS.gemm('T', 'N', 1., V, BU) # VtBU = V'BU
  norm_U = 1.
  for i in 1:itmax1
    mul!(U, V, VtBU, -1, 1) # U .-= V * VtBU
    mul!(BU, B, U) # BU .= B * U
    for _ in 1:itmax2
      svqb!(U, BU, tol_replace)
      norm_U = norm(U)
      UtBU = BLAS.gemm('T', 'N', 1., U, BU) # UtBU = U'BU
      err = norm(UtBU - I) / norm(BU) / norm_U
      err < tol_ortho ? break : nothing
    end
    BLAS.gemm!('T', 'N', 1., V, BU, 0., VtBU) # VtBU .= V'BU
    err = norm(VtBU) / norm_BV / norm_U
    err < tol_ortho ? break : nothing
  end
end;

function svqb!(U::AbstractArray{Float64,2},
               BU::AbstractArray{Float64,2},
               tol::Float64)
  # Stathopoulos, A., & Wu, K. (2002)
  # A block orthogonalization procedure with constant synchronization requirements
  # SIAM Journal on Scientific Computing, 23(6), 2165-2182

  # In/out
  # U :
  # BU:

  # tol:

  W = similar(U)
  UtBU = U'BU
  D = diagm(diag(UtBU).^(-.5))
  UtBU .= D * UtBU * D
  Theta, Z = eigen(Symmetric(UtBU))
```

```julia
  theta_abs_max = maximum(abs.(Theta))
  Theta[Theta .< tol * theta_abs_max] .= tol * theta_abs_max
  Z .= D * Z * diagm(Theta.^(-.5))
  mul!(W, U, Z); copy!(U, W) # U .= U * Z
  mul!(W, BU, Z); copy!(BU, W) # BU .= BU * Z
end;
```

[13]:
```julia
function Ortho!(U::AbstractArray{Float64,2},
                V::AbstractArray{Float64,2},
                tol_ortho::Float64=100*eps(Float64),
                tol_replace::Float64=10*eps(Float64),
                itmax1::Int=6, itmax2::Int=6)
  # Stathopoulos, A., & Wu, K. (2002)
  # A block orthogonalization procedure with constant synchronization␣
 ↪requirements
  # SIAM Journal on Scientific Computing, 23(6), 2165-2182

  VtU = BLAS.gemm('T', 'N', 1., V, U) # VtU = V'U
  norm_U = 1.
  norm_V = norm(V)
  for i in 1:itmax1
    mul!(U, V, VtU, -1, 1) # U .-= V * VtU
    for j in 1:itmax2
      svqb!(U, tol_replace)
      norm_U = norm(U)
      UtU = BLAS.gemm('T', 'N', 1., U, U) # UtU = U'U
      err = norm(UtU - I) / norm_U^2
      err < tol_ortho ? break : nothing
    end
    BLAS.gemm!('T', 'N', 1., V, U, 0., VtU) # VtU .= V'U
    err = norm(VtU) / norm_V / norm_U
    err < tol_ortho ? break : nothing
  end
end;

function svqb!(U::AbstractArray{Float64,2}, tol::Float64)
  # Stathopoulos, A., & Wu, K. (2002)
  # A block orthogonalization procedure with constant synchronization␣
 ↪requirements
  # SIAM Journal on Scientific Computing, 23(6), 2165-2182

  UtU = U'U
  D = diagm(diag(UtU).^(-.5))
  UtU .= D * UtU * D
  Theta, Z = eigen(Symmetric(UtU))
  theta_abs_max = maximum(abs.(Theta))
  Theta[Theta .< tol * theta_abs_max] .= tol * theta_abs_max
```

```julia
  Z .= D * Z * diagm(Theta.^(-.5))
  U .= U * Z
end;
```

[14]:
```julia
function Ortho!(U::AbstractArray{Float64,2},
                B::AbstractArray{Float64,2},
                V::AbstractArray{Float64,2},
                BU::AbstractArray{Float64,2},
                norm_BV::Float64,
                tol_ortho::Float64=100*eps(Float64),
                tol_replace::Float64=10*eps(Float64),
                itmax1::Int=6, itmax2::Int=6)
  # Stathopoulos, A., & Wu, K. (2002)
  # A block orthogonalization procedure with constant synchronization requirements
  # SIAM Journal on Scientific Computing, 23(6), 2165-2182

  VtBU = BLAS.gemm('T', 'N', 1., V, BU) # VtBU = V'BU
  norm_U = 1.
  for i in 1:itmax1
    mul!(U, V, VtBU, -1, 1) # U .-= V * VtBU
    mul!(BU, B, U) # BU .= B * U
    for _ in 1:itmax2
      svqb!(U, BU, tol_replace)
      norm_U = norm(U)
      UtBU = BLAS.gemm('T', 'N', 1., U, BU) # UtBU = U'BU
      err = norm(UtBU - I) / norm(BU) / norm_U
      err < tol_ortho ? break : nothing
    end
    BLAS.gemm!('T', 'N', 1., V, BU, 0., VtBU) # VtBU .= V'BU
    err = norm(VtBU) / norm_BV / norm_U
    err < tol_ortho ? break : nothing
  end
end

function svqb!(U::AbstractArray{Float64,2},
               BU::AbstractArray{Float64,2},
               tol::Float64)
  # Stathopoulos, A., & Wu, K. (2002)
  # A block orthogonalization procedure with constant synchronization requirements
  # SIAM Journal on Scientific Computing, 23(6), 2165-2182

  # In/out
  # U :
  # BU:
```

```
  # tol:

  W = similar(U)
  UtBU = U'BU
  D = diagm(diag(UtBU).^(-.5))
  UtBU .= D * UtBU * D
  Theta, Z = eigen(Symmetric(UtBU))
  theta_abs_max = maximum(abs.(Theta))
  Theta[Theta .< tol * theta_abs_max] .= tol * theta_abs_max
  Z .= D * Z * diagm(Theta.^(-.5))
  mul!(W, U, Z); copy!(U, W) # U .= U * Z
  mul!(W, BU, Z); copy!(BU, W) # BU .= BU * Z
end

function Ortho!(U::AbstractArray{Float64,2},
                V::AbstractArray{Float64,2},
                tol_ortho::Float64=100*eps(Float64),
                tol_replace::Float64=10*eps(Float64),
                itmax1::Int=6, itmax2::Int=6)
  # Stathopoulos, A., & Wu, K. (2002)
  # A block orthogonalization procedure with constant synchronization␣
 ↪requirements
  # SIAM Journal on Scientific Computing, 23(6), 2165-2182

  VtU = BLAS.gemm('T', 'N', 1., V, U) # VtU = V'U
  norm_U = 1.
  norm_V = norm(V)
  for i in 1:itmax1
    mul!(U, V, VtU, -1, 1) # U .-= V * VtU
    for j in 1:itmax2
      svqb!(U, tol_replace)
      norm_U = norm(U)
      UtU = BLAS.gemm('T', 'N', 1., U, U) # UtU = U'U
      err = norm(UtU - I) / norm_U^2
      err < tol_ortho ? break : nothing
    end
    BLAS.gemm!('T', 'N', 1., V, U, 0., VtU) # VtU .= V'U
    err = norm(VtU) / norm_V / norm_U
    err < tol_ortho ? break : nothing
  end
end

function svqb!(U::AbstractArray{Float64,2}, tol::Float64)
  # Stathopoulos, A., & Wu, K. (2002)
  # A block orthogonalization procedure with constant synchronization␣
 ↪requirements
  # SIAM Journal on Scientific Computing, 23(6), 2165-2182
```

```julia
    UtU = U'U
    D = diagm(diag(UtU).^(-.5))
    UtU .= D * UtU * D
    Theta, Z = eigen(Symmetric(UtU))
    theta_abs_max = maximum(abs.(Theta))
    Theta[Theta .< tol * theta_abs_max] .= tol * theta_abs_max
    Z .= D * Z * diag(Theta.^(-.5))
    U .= U * Z
end
```

[14]: svqb! (generic function with 2 methods)

[15]:
```julia
function Ortho_LOBPCG(A, B, X0, nev::Int;
                      T=I, itmax::Int=200, tol::Float64=1e-6,
                      A_products::Symbol=:implicit,
                      B_products::Symbol=:implicit,
                      debug::Bool=false)
    # Hetmaniuk, U., & Lehoucq, R. (2006)
    # Basis selection in LOBPCG
    # Journal of Computational Physics, 218(1), 324-332

    # In
    # A          : left  hand-side operator, symmetric positive definite, n-by-n
    # B          : right hand-side operator, symmetric positive definite, n-by-n
    # X0         : initial iterates, n-by-m (m < n)
    # nev        : number of wanted eigenpairs, nev <= m
    # T          : precondontioner, symmetric positive definite, n-by-n
    # itmax      : maximum number of iterations
    # tol        : tolerance used for convergence criterion
    # A_products: if :implicit, the matrix products with A are updated implicitly
    # B_products: if :implicit, the matrix products with B are updated implicitly
    # debug      : if true, and A_products == :implicit, prints out error norm of
    #              implicit product updates

    # Out
    # Lambda: last iterates of least dominant eigenvalues, m-by-1
    # X     : last iterates of least dominant eigenvectors, n-by-m
    # res   : normalized norms of eigenresiduals, m-by-it

    n, m = size(X0)

    R = Array{Float64,2}(undef, n, m)
    XP = Array{Float64,2}(undef, n, 2*m)
    Z = Array{Float64,2}(undef, n, m)
    Q = Array{Float64,2}(undef, n, m)
    W = Array{Float64,2}(undef, n, m)
```

```
AX = Array{Float64,2}(undef, n, m)
AZ = Array{Float64,2}(undef, n, m)
AP = Array{Float64,2}(undef, n, m)
BX = Array{Float64,2}(undef, n, m)
BZ = Array{Float64,2}(undef, n, m)
BP = Array{Float64,2}(undef, n, m)

X = view(XP, :, 1:m); P = view(XP, :, m+1:2*m)

res = Array{Float64,2}(undef, m, itmax+1)
k = 0

copy!(X, X0)
mul!(AX, A, X) # AX .= A * X
mul!(BX, B, X) # BX .= B * X
hX, Lambda = RR(X, AX, BX)
mul!(W, X, hX); copy!(X, W) # X .= X * hX
if A_products == :implicit
  mul!(W, AX, hX); copy!(AX, W) # AX .= AX * hX
else
  mul!(AX, A, X) # AX .= A * X
end
if B_products == :implicit
  mul!(W, BX, hX); copy!(BX, W) # BX .= BX * hX
else
  mul!(BX, B, X) # BX .= B * X
end
# R .= AX - BX * diagm(Lambda)
copy!(R, AX); mul!(R, BX, diagm(Lambda), -1, 1)
res[:, 1] = norm.(eachcol(R))
res[:, 1] ./= abs.(Lambda)
for i in k+1:nev
  res[i, 1] < tol ? k += 1 : break
end
println("it = 0, k = $k")
println("extrema(res) = ", extrema(res[:, 1]))

if k < nev
  for j in 1:itmax
    println("it = $j, k = $k")
    ldiv!(Z, T, R) # Z .= T \ R
    if j == 1
      mul!(BZ, B, Z) # BZ .= B * Z
      norm_BX = norm(BX)
      Ortho!(Z, B, X, BZ, norm_BX)
      mul!(AZ, A, Z) # AZ .= A * Z
      hX, Lambda = RR(X, Z, AX, AZ)
```

```
    hX_X = view(hX, 1:m, :); hX_Z = view(hX, m+1:2*m, :)
    hY = vcat(zeros(m, m), hX_Z)
    Ortho!(hY, vcat(hX_X, hX_Z))
    hY_X = view(hY, 1:m, :); hY_Z = view(hY, m+1:2*m, :)
    if A_products == :implicit
      # AP .= AX * hY_X + AZ * hY_Z
      mul!(AP, AX, hY_X)
      mul!(AP, AZ, hY_Z, 1, 1)
      # AX .= AX * hX_X + AZ * hX_Z
      mul!(W, AX, hX_X)
      mul!(W, AZ, hX_Z, 1, 1)
      copy!(AX, W)
    end
    if B_products == :implicit
      # BP .= BX * hY_X + BZ * hY_Z
      mul!(BP, BX, hY_X)
      mul!(BP, BZ, hY_Z, 1, 1)
      # BX .= BX * hX_X + BZ * hX_Z
      mul!(W, BX, hX_X)
      mul!(W, BZ, hX_Z, 1, 1)
      copy!(BX, W)
    end
    # W .= X * hX_X + Z * hX_Z
    mul!(W, X, hX_X)
    mul!(W, Z, hX_Z, 1, 1)
    # P .= X * hY_X + Z * hY_Z
    mul!(Q, X, hY_X)
    mul!(Q, Z, hY_Z, 1, 1)
    copy!(P, Q)
  else
    mul!(BZ, B, Z) # BZ .= B * Z
    norm_BXP = sqrt(norm(BX)^2+norm(BP)^2)
    Ortho!(Z, B, XP, BZ, norm_BXP)
    mul!(AZ, A, Z) # AZ .= A * Z
    hX, Lambda = RR(X, Z, P, AX, AZ, AP)
    hX_X = view(hX, 1:m, :); hX_Z = view(hX, m+1:2*m, :)
    hX_P = view(hX, 2*m+1:3*m, :)
    hY = vcat(zeros(m, m), hX_Z, hX_P)
    Ortho!(hY, vcat(hX_X, hX_Z, hX_P))
    hY_X = view(hY, 1:m, :); hY_Z = view(hY, m+1:2*m, :)
    hY_P = view(hY, 2*m+1:3*m, :)
    if A_products == :implicit
      # W .= AX * hY_X + AZ * hY_Z + AP * hY_P
      mul!(W, AX, hY_X)
      mul!(W, AZ, hY_Z, 1, 1)
      mul!(W, AP, hY_P, 1, 1)
      # AX .= AX * hX_X + AZ * hX_Z + AP * hX_P
```

```
      mul!(Q, AX, hX_X)
      mul!(Q, AZ, hX_Z, 1, 1)
      mul!(Q, AP, hX_P, 1, 1)
      copy!(AX, Q)
      copy!(AP, W)
    end
    if B_products == :implicit
      # W .= BX * hY_X + BZ * hY_Z + BP * hY_P
      mul!(W, BX, hY_X)
      mul!(W, BZ, hY_Z, 1, 1)
      mul!(W, BP, hY_P, 1, 1)
      # BX .= BX * hX_X + BZ * hX_Z + BP * hX_P
      mul!(Q, BX, hX_X)
      mul!(Q, BZ, hX_Z, 1, 1)
      mul!(Q, BP, hX_P, 1, 1)
      copy!(BX, Q)
      copy!(BP, W)
    end
    # W .= X * hX_X + Z * hX_Z + P * hX_P
    mul!(W, X, hX_X)
    mul!(W, Z, hX_Z, 1, 1)
    mul!(W, P, hX_P, 1, 1)
    # P .= X * hY_X + Z * hY_Z + P * hY_P
    mul!(Q, X, hY_X)
    mul!(Q, Z, hY_Z, 1, 1)
    mul!(Q, P, hY_P, 1, 1)
    copy!(P, Q)
  end
  copy!(X, W)
  if A_products != :implicit
    mul!(AX, A, X) # AX .= A * X
    mul!(AP, A, P) # AP .= A * P
  end
  if B_products != :implicit
    mul!(BX, B, X) # BX .= B * X
    mul!(BP, B, P) # BP .= B * P
  end
  # R .= AX - BX * diagm(Lambda)
  copy!(R, AX); mul!(R, BX, diagm(Lambda), -1, 1)
  res[:, j+1] = norm.(eachcol(R))
  res[:, j+1] ./= abs.(Lambda)
  if debug
    println("||[P,X]'B*[P,X]|| = ", norm(hcat(P, X)'B*hcat(P, X)-I))
    if A_products == :implicit
      println("||AX-A*X|| = ", norm(AX - A * X))
      println("||AP-A*P|| = ", norm(AP - A * P))
    end
```

```
      if B_products == :implicit
        println("||BX-B*X|| = ", norm(BX - B * X))
        println("||BP-B*P|| = ", norm(BP - B * P))
      end
    end
    println("extrema(res) = ", extrema(res[:, j+1]))
    for i in k+1:nev
      res[i, j+1] < tol ? k += 1 : break
    end
    k < nev ? nothing : return Lambda, X, res[:, 1:j+1]
  end
 end
 return Lambda, X, res
end;
```

Exercise #4: Implement Skip_ortho_LOBPCG

[16]:

```
function Skip_ortho_LOBPCG(A, B, X0, nev::Int;
                           T=I, itmax::Int=200, tol::Float64=1e-6,
                           A_products::Symbol=:implicit,
                           B_products::Symbol=:implicit,
                           debug::Bool=false)
  # Duersch, J. A., Shao, M., Yang, C., & Gu, M. (2018)
  # A robust and efficient implementation of LOBPCG
  # SIAM Journal on Scientific Computing, 40(5), C655-C676

  # In
  # A          : left  hand-side operator, symmetric positive definite, n-by-n
  # B          : right hand-side operator, symmetric positive definite, n-by-n
  # X0         : initial iterates, n-by-m (m < n)
  # nev        : number of wanted eigenpairs, nev <= m
  # T          : precondontioner, symmetric positive definite, n-by-n
  # itmax      : maximum number of iterations
  # tol        : tolerance used for convergence criterion
  # A_products: if :implicit, the matrix products with A are updated implicitly
  # B_products: if :implicit, the matrix products with B are updated implicitly
  # debug      : if true, and A_products == :implicit, prints out error norm of
  #              implicit product updates

  # Out
  # Lambda: last iterates of least dominant eigenvalues, m-by-1
  # X     : last iterates of least dominant eigenvectors, n-by-m
  # res   : normalized norms of eigenresiduals, m-by-it

  n, m = size(X0)

  skipOrtho = true
```

```
modified = true

R = Array{Float64,2}(undef, n, m)
XP = Array{Float64,2}(undef, n, 2*m)
Z = Array{Float64,2}(undef, n, m)
W = Array{Float64,2}(undef, n, m)
Q = Array{Float64,2}(undef, n, m)
AX = Array{Float64,2}(undef, n, m)
AZ = Array{Float64,2}(undef, n, m)
AP = Array{Float64,2}(undef, n, m)
BX = Array{Float64,2}(undef, n, m)
BZ = Array{Float64,2}(undef, n, m)
BP = Array{Float64,2}(undef, n, m)

X = view(XP, :, 1:m); P = view(XP, :, m+1:2*m)

res = Array{Float64,2}(undef, m, itmax+1)
k = 0

copy!(X, X0)
mul!(AX, A, X) # AX .= A * X
mul!(BX, B, X) # BX .= B * X
hX, Lambda = RR(X, AX, BX)
mul!(W, X, hX); copy!(X, W) # X .= X * hX
if A_products == :implicit
  mul!(W, AX, hX); copy!(AX, W) # AX .= AX * hX
else
  mul!(AX, A, X) # AX .= A * X
end
if B_products == :implicit
  mul!(W, BX, hX); copy!(BX, W) # BX .= BX * hX
else
  mul!(BX, B, X) # BX .= B * X
end
# R .= AX - BX * diagm(Lambda)
copy!(R, AX); mul!(R, BX, diagm(Lambda), -1, 1)
res[:, 1] = norm.(eachcol(R))
res[:, 1] ./= abs.(Lambda)
for i in k+1:nev
  res[i, 1] < tol ? k += 1 : break
end
println("it = 0, k = $k")
println("extrema(res) = ", extrema(res[:, 1]))

if k < nev
  for j in 1:itmax
    println("it = $j, k = $k, skipOrtho = $skipOrtho")
```

```
    ldiv!(Z, T, R) # Z .= T \ R
    if j == 1
      if skipOrtho
        mul!(BZ, B, Z) # BZ .= B * Z
        mul!(AZ, A, Z) # AZ .= A * Z
        hX, Lambda, skipOrtho, VtBV = RR(X, Z, AX, AZ, BZ, true)
        hX_X = view(hX, 1:m, :); hX_Z = view(hX, m+1:2*m, :)
        if skipOrtho
          hY = vcat(zeros(m, m), hX_Z)
          norm_VtBVXZ = norm(VtBV * vcat(hX_X, hX_Z))
          Ortho!(hY, VtBV, vcat(hX_X, hX_Z), VtBV * hY, norm_VtBVXZ)
          hY_X = view(hY, 1:m, :); hY_Z = view(hY, m+1:2*m, :)
          if A_products == :implicit
            # AP .= AX * hY_X + AZ * hY_Z
            mul!(AP, AX, hY_X)
            mul!(AP, AZ, hY_Z, 1, 1)
            # AX .= AX * hX_X + AZ * hX_Z
            mul!(W, AX, hX_X)
            mul!(W, AZ, hX_Z, 1, 1)
            copy!(AX, W)
          end
          if B_products == :implicit
            # BP .= BX * hY_X + BZ * hY_Z
            mul!(BP, BX, hY_X)
            mul!(BP, BZ, hY_Z, 1, 1)
            # BX .= BX * hX_X + BZ * hX_Z
            mul!(W, BX, hX_X)
            mul!(W, BZ, hX_Z, 1, 1)
            copy!(BX, W)
          end
          # W .= X * hX_X + Z * hX_Z
          mul!(W, X, hX_X)
          mul!(W, Z, hX_Z, 1, 1)
          # P .= X * hY_X + Z * hY_Z
          mul!(P, X, hY_X)
          mul!(P, Z, hY_Z, 1, 1)
          copy!(X, W)
          if A_products != :implicit
            mul!(AP, A, P) # AP .= A * P
            mul!(AX ,A, X) # AX .= A * X
          end
          if B_products != :implicit
            mul!(BP, B, P) # BP .= B * P
            mul!(BX ,B, X) # BX .= B * X
          end
        end
      end
```

```
    if !skipOrtho
      mul!(BZ, B, Z) # BZ .= B * Z
      norm_BX = norm(BX)
      Ortho!(Z, B, X, BZ, norm_BX)
      mul!(AZ, A, Z) # AZ .= A * Z
      if modified
        hX, Lambda, hY = RR(X, Z, AX, AZ, Float64[], modified)
        hX_X = view(hX, 1:m, :); hX_Z = view(hX, m+1:2*m, :)
      else
        hX, Lambda = RR(X, Z, AX, AZ)
        hX_X = view(hX, 1:m, :); hX_Z = view(hX, m+1:2*m, :)
        hY = vcat(zeros(m, m), hX_Z)
        Ortho!(hY, vcat(hX_X, hX_Z))
      end
      hY_X = view(hY, 1:m, :); hY_Z = view(hY, m+1:2*m, :)
      if A_products == :implicit
        # AP .= AX * hY_X + AZ * hY_Z
        mul!(AP, AX, hY_X)
        mul!(AP, AZ, hY_Z, 1, 1)
      else
        mul!(AP, A, P) # AP .= A * P
      end
      if B_products == :implicit
        # BP .= BX * hY_X + BZ * hY_Z
        mul!(BP, BX, hY_X)
        mul!(BP, BZ, hY_Z, 1, 1)
      else
        mul!(BP, B, P) # BP .= B * P
      end
      if A_products == :implicit
        # AX .= AX * hX_X + AZ * hX_Z
        mul!(W, AX, hX_X)
        mul!(W, AZ, hX_Z, 1, 1)
        copy!(AX, W)
      else
        mul!(AX, A, X) # AX .= A * X
      end
      if B_products == :implicit
        # BX .= BX * hX_X + BZ * hX_Z
        mul!(W, BX, hX_X)
        mul!(W, BZ, hX_Z, 1, 1)
        copy!(BX, W)
      else
        mul!(BX, B, X) # BX .= B * X
      end
      # W .= X * hX_X + Z * hX_Z
      mul!(W, X, hX_X)
```

```
        mul!(W, Z, hX_Z, 1, 1)
        # P .= X * hY_X + Z * hY_Z
        mul!(P, X, hY_X)
        mul!(P, Z, hY_Z, 1, 1)
        copy!(X, W)
        if A_products != :implicit
          mul!(AX, A, X) # AX .= A * X
          mul!(AP, A, P) # AP .= A * P
        end
        if B_products != :implicit
          mul!(BX, B, X) # BX .= B * X
          mul!(BP, B, P) # BP .= B * P
        end
      end
    else
      if skipOrtho
        mul!(BZ, B, Z) # BZ .= B * Z
        mul!(AZ, A, Z) # AZ .= A * Z
        hX, Lambda, skipOrtho, VtBV = RR(X, Z, P, AX, AZ, AP, BZ, BP, true)
        hX_X = view(hX, 1:m, :); hX_Z = view(hX, m+1:2*m, :)
        hX_P = view(hX, 2*m+1:3*m, :)
        if skipOrtho
          hY = vcat(zeros(m, m), hX_Z, hX_P)
          norm_VtBVXZP = norm(VtBV * vcat(hX_X, hX_Z, hX_P))
          Ortho!(hY, VtBV, vcat(hX_X, hX_Z, hX_P), VtBV * hY, norm_VtBVXZP)
          hY_X = view(hY, 1:m, :); hY_Z = view(hY, m+1:2*m, :); hY_P =
↪view(hY, 2*m+1:3*m, :)
          if A_products == :implicit
            # W .= AX * hY_X + AZ * hY_Z + AP * hY_P
            mul!(W, AX, hY_X)
            mul!(W, AZ, hY_Z, 1, 1)
            mul!(W, AP, hY_P, 1, 1)
            # AX = AX * hX_X + AZ * hX_Z + AP * hX_P
            mul!(Q, AX, hX_X)
            mul!(Q, AZ, hX_Z, 1, 1)
            mul!(Q, AP, hX_P, 1, 1)
            copy!(AX, Q)
            copy!(AP, W)
          end
          if B_products == :implicit
            # W .= BX * hY_X + BZ * hY_Z + BP * hY_P
            mul!(W, BX, hY_X)
            mul!(W, BZ, hY_Z, 1, 1)
            mul!(W, BP, hY_P, 1, 1)
            # BX = BX * hX_X + BZ * hX_Z + BP * hX_P
            mul!(Q, BX, hX_X)
            mul!(Q, BZ, hX_Z, 1, 1)
```

```
          mul!(Q, BP, hX_P, 1, 1)
          copy!(BX, Q)
          copy!(BP, W)
        end
        # W .= X * hX_X + Z * hX_Z + P * hX_P
        mul!(W, X, hX_X)
        mul!(W, Z, hX_Z, 1, 1)
        mul!(W, P, hX_P, 1, 1)
        # P .= X * hY_X + Z * hY_Z + P * hY_P
        mul!(Q, X, hY_X)
        mul!(Q, Z, hY_Z, 1, 1)
        mul!(Q, P, hY_P, 1, 1)
        copy!(P, Q)
        copy!(X, W)
        if A_products != :implicit
          mul!(AX, A, X) # AX .= A * X
          mul!(AP, A, P) # AP .= A * P
        end
        if B_products != :implicit
          mul!(BX, B, X) # BX .= B * X
          mul!(BP, B, P) # BP .= B * P
        end
      end
    end
    if !skipOrtho
      mul!(BZ, B, Z) # BZ .= B * Z
      norm_BXP = sqrt(norm(BX)^2 + norm(BP)^2)
      Ortho!(Z, B, XP, BZ, norm_BXP)
      mul!(AZ, A, Z) # AZ .= A * Z
      if modified
        hX, Lambda, hY = RR(X, Z, P, AX, AZ, AP, Float64[], modified)
        hX_X = view(hX, 1:m, :); hX_Z = view(hX, m+1:2*m, :)
        hX_P = view(hX, 2*m+1:3*m, :)
      else
        hX, Lambda = RR(X, Z, P, AX, AZ, AP)
        hX_X = view(hX, 1:m, :); hX_Z = view(hX, m+1:2*m, :)
        hX_P = view(hX, 2*m+1:3*m, :)
        hY = vcat(zeros(m, m), hX_Z, hX_P)
        Ortho!(hY, vcat(hX_X, hX_Z, hX_P))
      end
      hY_X = view(hY, 1:m, :); hY_Z = view(hY, m+1:2*m, :)
      hY_P = view(hY, 2*m+1:3*m, :)
      if A_products == :implicit
        # W .= AX * hY_X + AZ * hY_Z + AP * hY_P
        mul!(W, AX, hY_X)
        mul!(W, AZ, hY_Z, 1, 1)
        mul!(W, AP, hY_P, 1, 1)
```

```
        # AX .= AX * hX_X + AZ * hX_Z + AP * hX_P
        mul!(Q, AX, hX_X)
        mul!(Q, AZ, hX_Z, 1, 1)
        mul!(Q, AP, hX_P, 1, 1)
        copy!(AX, Q)
        copy!(AP, W)
      end
      if B_products == :implicit
        # W .= BX * hY_X + BZ * hY_Z + BP * hY_P
        mul!(W, BX, hY_X)
        mul!(W, BZ, hY_Z, 1, 1)
        mul!(W, BP, hY_P, 1, 1)
        # BX .= BX * hX_X + BZ * hX_Z + BP * hX_P
        mul!(Q, BX, hX_X)
        mul!(Q, BZ, hX_Z, 1, 1)
        mul!(Q, BP, hX_P, 1, 1)
        copy!(BX, Q)
        copy!(BP, W)
      end
      # W .= X * hX_X + Z * hX_Z + P * hX_P
      mul!(W, X, hX_X)
      mul!(W, Z, hX_Z, 1, 1)
      mul!(W, P, hX_P, 1, 1)
      # P .= X * hY_X + Z * hY_Z + P * hY_P
      mul!(Q, X, hY_X)
      mul!(Q, Z, hY_Z, 1, 1)
      mul!(Q, P, hY_P, 1, 1)
      copy!(P, Q)
      copy!(X, W)
      if A_products != :implicit
        mul!(AX, A, X) # AX .= A * X
        mul!(AP, A, P) # AP .= A * P
      end
      if B_products != :implicit
        mul!(BX, B, X) # BX .= B * X
        mul!(BP, B, P) # BP .= B * P
      end
    end
  end
  # R .= AX - BX * diagm(Lambda)
  copy!(R, AX); mul!(R, BX, diagm(Lambda), -1, 1)
  res[:, j+1] = norm.(eachcol(R))
  res[:, j+1] ./= abs.(Lambda)
  if debug
    if A_products == :implict
      println("||AX-A*X|| = ", norm(AX - A * X))
      println("||AP-A*P|| = ", norm(AP - A * P))
```

```
        end
        if B_products == :implicit
          println("||BX-B*X|| = ", norm(BX - B * X))
          println("||BP-B*P|| = ", norm(BP - B * P))
        end
      end
      println("extrema(res) = ", extrema(res[:, j+1]))
      for i in k+1:nev
        res[i, j+1] < tol ? k += 1 : break
      end
      k < nev ? nothing : return Lambda, X, res[:, 1:j+1]
    end
  end
  return Lambda, X, res
end;
```

Exercise #5: Benchmark

[17]:
```
struct BJop
  n::Int
  nb::Int
  bsize::Int
  factos::Union{Vector{SparseArrays.CHOLMOD.Factor{Float64,Int64}},
                Vector{SparseArrays.UMFPACK.UmfpackLU{Float64,Int64}}}
  facto_type::String
end;

function slice(i::Int, nb::Int, bsize::Int, n::Int)
  if i < nb
    return ((i - 1 ) * bsize + 1):(i * bsize)
  else
    return ((i - 1 ) * bsize + 1):n
  end
end;

function BJPreconditioner(nb::Int, A::SparseMatrixCSC{Float64})
  n = A.n
  bsize = Int(floor((n // nb)))
  if isposdef(A)
    factos = map(i -> cholesky(A[slice(i, nb, bsize, n),
                                 slice(i, nb, bsize, n)]), 1:nb)
    return BJop(n, nb, bsize, factos, "chol")
  else
    factos = map(i -> lu(A[slice(i, nb, bsize, n),
                           slice(i, nb, bsize, n)]), 1:nb)
    return BJop(n, nb, bsize, factos, "lu")
  end
```

```
end;

function invT!(X::AbstractArray{Float64,2},
               precond::BJop,
               Z::AbstractArray{Float64,2})
  for i in 1:precond.nb
    islice = slice(i, precond.nb, precond.bsize, precond.n)
    Z[islice, :] = precond.factos[i] \ X[islice, :]
  end
end;

function invT!(x::Vector{Float64},
               precond::BJop,
               z::Vector{Float64})
  for i in 1:precond.nb
    islice = slice(i, precond.nb, precond.bsize, precond.n)
    z[islice] = precond.factos[i] \ x[islice]
  end
end;

function invT(X::AbstractArray{Float64,2}, precond::BJop)
  Z = similar(X)
  invT!(X, precond, Z)
  return Z
end;

function invT(x::Vector{Float64}, precond::BJop)
  z = similar(x)
  invT!(x, precond, z)
  return z
end;

import Base: \
(\)(T::BJop, X) = invT(X, T::BJop);

import LinearAlgebra: ldiv!
ldiv!(Z, T::BJop, X) = invT!(X, T::BJop, Z);
```

[18]:
```
matrix_source = "/home/venkovic/Dropbox/Git/matrix-market/";
A = mmread(matrix_source * "bcsstk12.mtx");
B = mmread(matrix_source * "bcsstm12.mtx");
```

[19]:
```
n, _ = size(A);
m = 200;
nev = 10;
T = BJPreconditioner(10, A);
X0 = rand(n, m);
```

[20]:
```
tol = 1e-5;
```

[21]:
```
Lambda, X, res = @time Basic_LOBPCG(A, B, X0, nev, T=T, tol=tol);
```

```
it = 0, k = 0
extrema(res) = (0.3300303655975059, 1.537769678077098)
it = 1, k = 0
extrema(res) = (0.4049282907328257, 21.22165503549347)
it = 2, k = 0
extrema(res) = (0.16200941298647892, 7.182750266997795)
it = 3, k = 0
extrema(res) = (0.07349862403277621, 11.437971535689176)
it = 4, k = 0
extrema(res) = (0.020491807123343607, 21.65946790346044)
it = 5, k = 0
extrema(res) = (0.006833383721195593, 40.78220661300664)
it = 6, k = 0
extrema(res) = (0.0022527567539516215, 40.15919254360722)
it = 7, k = 0
extrema(res) = (0.0006738751143275161, 13.563516352095787)
it = 8, k = 0
extrema(res) = (0.00019891735013226718, 4.093336436417616)
it = 9, k = 0
extrema(res) = (7.339932070691965e-5, 1.2290927245690366)
it = 10, k = 0
extrema(res) = (2.229155583161873e-5, 0.3003242827551401)
it = 11, k = 0
extrema(res) = (6.2023309211219835e-6, 0.0776289064051738)
it = 12, k = 0
extrema(res) = (1.8169954512491298e-6, 0.04169199431181375)
it = 13, k = 0
extrema(res) = (6.222493687609023e-7, 0.029629747719154168)
it = 14, k = 0
extrema(res) = (1.9252518379901585e-7, 0.02136143179346194)
it = 15, k = 0
extrema(res) = (6.045279500132419e-8, 0.01469914303905413)
it = 16, k = 0
extrema(res) = (1.95590358740195e-8, 0.010048497551251728)
it = 17, k = 0
extrema(res) = (6.0483263809570256e-9, 0.007269042372730877)
it = 18, k = 0
extrema(res) = (2.208494237893491e-9, 0.005069785622807576)
it = 19, k = 0
extrema(res) = (7.294482380356976e-10, 0.0038852428578237726)
it = 20, k = 1
extrema(res) = (5.512539070253902e-8, 2.137010833435236)
it = 21, k = 1
```

```
PosDefException: matrix is not positive definite; Factorization failed.

Stacktrace:
  [1] checkpositivedefinite
    @ ~/.julia/juliaup/julia-1.11.5+0.x64.linux.gnu/share/julia/stdlib/v1.11/
 ↪LinearAlgebra/src/factorization.jl:68 [inlined]
  [2] #cholesky!#163
    @ ~/.julia/juliaup/julia-1.11.5+0.x64.linux.gnu/share/julia/stdlib/v1.11/
 ↪LinearAlgebra/src/cholesky.jl:269 [inlined]
  [3] cholesky! (repeats 2 times)
    @ ~/.julia/juliaup/julia-1.11.5+0.x64.linux.gnu/share/julia/stdlib/v1.11/
 ↪LinearAlgebra/src/cholesky.jl:267 [inlined]
  [4] _cholesky
    @ ~/.julia/juliaup/julia-1.11.5+0.x64.linux.gnu/share/julia/stdlib/v1.11/
 ↪LinearAlgebra/src/cholesky.jl:411 [inlined]
  [5] cholesky (repeats 2 times)
    @ ~/.julia/juliaup/julia-1.11.5+0.x64.linux.gnu/share/julia/stdlib/v1.11/
 ↪LinearAlgebra/src/cholesky.jl:401 [inlined]
  [6] RR(X::Matrix{Float64}, Z::SubArray{Float64, 2, Matrix{Float64}, Tuple{Base.
 ↪Slice{Base.OneTo{Int64}}, UnitRange{Int64}}, true}, P::SubArray{Float64, 2,␣
 ↪Matrix{Float64}, Tuple{Base.Slice{Base.OneTo{Int64}}, UnitRange{Int64}},␣
 ↪true}, AX::Matrix{Float64}, AZ::SubArray{Float64, 2, Matrix{Float64},␣
 ↪Tuple{Base.Slice{Base.OneTo{Int64}}, UnitRange{Int64}}, true}, AP::
 ↪SubArray{Float64, 2, Matrix{Float64}, Tuple{Base.Slice{Base.OneTo{Int64}},␣
 ↪UnitRange{Int64}}, true}, BZ::SubArray{Float64, 2, Matrix{Float64}, Tuple{Base.
 ↪Slice{Base.OneTo{Int64}}, UnitRange{Int64}}, true}, BP::SubArray{Float64, 2,␣
 ↪Matrix{Float64}, Tuple{Base.Slice{Base.OneTo{Int64}}, UnitRange{Int64}},␣
 ↪true}, check_L_cond::Bool, Lambda::Vector{Float64})
    @ Main ./In[4]:53
  [7] RR (repeats 2 times)
    @ ./In[4]:17 [inlined]
  [8] Basic_LOBPCG(A::SparseMatrixCSC{Float64, Int64}, B::
 ↪SparseMatrixCSC{Float64, Int64}, X0::Matrix{Float64}, nev::Int64; T::BJop,␣
 ↪itmax::Int64, tol::Float64, A_products::Symbol, B_products::Symbol, debug::
 ↪Bool)
    @ Main ./In[5]:95
  [9] top-level scope
    @ ./timing.jl:581 [inlined]
 [10] top-level scope
    @ ./In[21]:0
```

[22]:
```
Lambda, X, res = @time BLOPEX_LOBPCG(A, B, X0, nev, T=T, tol=tol);
```

```
it = 0, k = 0
extrema(res) = (0.33003036559751003, 1.537769678077147)
it = 1, k = 0
extrema(res) = (11.405001218661159, 2.5264165891670524e6)
it = 2, k = 0
extrema(res) = (0.3710911673382447, 18.74916589306212)
```

```
it = 3, k = 0
extrema(res) = (0.09850625489582956, 6.625861632782449)
it = 4, k = 0
extrema(res) = (0.05026652634571067, 16.892006024946916)
it = 5, k = 0
extrema(res) = (0.014242521818215528, 28.99385155466423)
it = 6, k = 0
extrema(res) = (0.004520221978272018, 39.84748592888609)
it = 7, k = 0
extrema(res) = (0.001244983645943328, 21.199912308193987)
it = 8, k = 0
extrema(res) = (0.0003716998725228198, 6.882985255259134)
it = 9, k = 0
extrema(res) = (0.00014279240592520972, 2.3849327092660975)
it = 10, k = 0
extrema(res) = (4.535275405467433e-5, 0.8105885598938721)
it = 11, k = 0
extrema(res) = (1.274775259325286e-5, 0.2488648582008478)
it = 12, k = 0
extrema(res) = (3.967984126421847e-6, 0.07045788565493503)
it = 13, k = 0
extrema(res) = (1.1908663128205423e-6, 0.037633240012277476)
it = 14, k = 0
extrema(res) = (4.056493820949837e-7, 0.02825083439903704)
it = 15, k = 0
extrema(res) = (1.3147364930033877e-7, 0.022741800196865208)
it = 16, k = 0
extrema(res) = (4.343376197385628e-8, 0.018201103460178805)
it = 17, k = 0
extrema(res) = (1.462934696109161e-8, 0.01437629190363668)
it = 18, k = 0
extrema(res) = (5.254364589003228e-9, 0.014358354115119414)
it = 19, k = 0
extrema(res) = (1.8795460014487177e-9, 0.01213127136734405)
it = 20, k = 0
extrema(res) = (5.757537677183721e-10, 0.00924249687939879)
it = 21, k = 1
extrema(res) = (2.0468320621610878e-10, 0.007674076239567144)
  5.411020 seconds (4.34 M allocations: 1.026 GiB, 14.99% gc time, 35.16%
compilation time)
```

[23]:
```
Lambda, X, res = @time Ortho_LOBPCG(A, B, X0, nev, T=T, tol=tol);
```

```
it = 0, k = 0
extrema(res) = (0.3300303655975059, 1.537769678077098)
it = 1, k = 0
extrema(res) = (0.40492829073269426, 21.221655035487384)
it = 2, k = 0
```

```
extrema(res) = (0.16200941298757182, 7.182750266996059)
it = 3, k = 0
extrema(res) = (0.07349862403046775, 11.437971535689487)
it = 4, k = 0
extrema(res) = (0.02049180712031476, 21.659467904094267)
it = 5, k = 0
extrema(res) = (0.006833383665072171, 40.78220661675316)
it = 6, k = 0
extrema(res) = (0.0022527568092916674, 40.159190790099515)
it = 7, k = 0
extrema(res) = (0.0006738751071814033, 13.56351530219227)
it = 8, k = 0
extrema(res) = (0.0001989173502197883, 4.093336298469632)
it = 9, k = 0
extrema(res) = (7.339932281283249e-5, 1.2290930325994853)
it = 10, k = 0
extrema(res) = (2.2291552638477075e-5, 0.30032423179723855)
it = 11, k = 0
extrema(res) = (6.202334290113706e-6, 0.07762887827692491)
it = 12, k = 0
extrema(res) = (1.8169929375997989e-6, 0.04169214824676901)
it = 13, k = 0
extrema(res) = (6.222485630247953e-7, 0.029629923556402375)
it = 14, k = 0
extrema(res) = (1.9252450500574176e-7, 0.021361611876170774)
it = 15, k = 0
extrema(res) = (6.044935730316275e-8, 0.01469704121625765)
it = 16, k = 0
extrema(res) = (1.957094362903719e-8, 0.010034446355838679)
it = 17, k = 0
extrema(res) = (6.0620105741208605e-9, 0.00726870571749568)
it = 18, k = 0
extrema(res) = (2.142250976118473e-9, 0.005103339471330485)
it = 19, k = 0
extrema(res) = (6.785992631756814e-10, 0.0037844693814731278)
it = 20, k = 1
extrema(res) = (2.52726160871652e-10, 0.0024346738828153486)
  8.592583 seconds (9.34 M allocations: 1.387 GiB, 8.31% gc time, 43.51%
compilation time)
```

[24]:
```
Lambda, X, res = @time Skip_ortho_LOBPCG(A, B, X0, nev, T=T, tol=tol);
```

```
it = 0, k = 0
extrema(res) = (0.3300303655975059, 1.537769678077098)
it = 1, k = 0, skipOrtho = true
extrema(res) = (0.4049282907328257, 21.22165503549347)
it = 2, k = 0, skipOrtho = true
extrema(res) = (0.16200941298626606, 7.182750266996343)
```

```
it = 3, k = 0, skipOrtho = true
extrema(res) = (0.07349862403494663, 11.437971535686156)
it = 4, k = 0, skipOrtho = true
extrema(res) = (0.020491807116805875, 21.659467904033058)
it = 5, k = 0, skipOrtho = true
extrema(res) = (0.006833383659256521, 40.78220661697174)
it = 6, k = 0, skipOrtho = true
extrema(res) = (0.002252756809008092, 40.1591907896633)
it = 7, k = 0, skipOrtho = true
extrema(res) = (0.000673875106755757, 13.56351530186385)
it = 8, k = 0, skipOrtho = true
extrema(res) = (0.0001989173500304196, 4.0933362985239246)
it = 9, k = 0, skipOrtho = true
extrema(res) = (7.339932275009455e-5, 1.22909303225764)
it = 10, k = 0, skipOrtho = true
extrema(res) = (2.229155253313179e-5, 0.3003242337467159)
it = 11, k = 0, skipOrtho = true
extrema(res) = (6.202334473199743e-6, 0.07762887830347989)
it = 12, k = 0, skipOrtho = true
extrema(res) = (1.816992535656678e-6, 0.04169215654096534)
it = 13, k = 0, skipOrtho = true
extrema(res) = (6.222489849144442e-7, 0.02962989856092638)
it = 14, k = 0, skipOrtho = true
extrema(res) = (1.9252399130548884e-7, 0.021361588389590423)
it = 15, k = 0, skipOrtho = true
extrema(res) = (6.044877589218183e-8, 0.014696430609415658)
it = 16, k = 0, skipOrtho = true
extrema(res) = (1.9565880239618814e-8, 0.010033382403694587)
it = 17, k = 0, skipOrtho = true
extrema(res) = (6.062828533337184e-9, 0.0072624650212738204)
it = 18, k = 0, skipOrtho = true
extrema(res) = (2.1370139176285207e-9, 0.005080564087687539)
it = 19, k = 0, skipOrtho = true
extrema(res) = (6.702640376390765e-10, 0.003866767420569932)
it = 20, k = 1, skipOrtho = true
extrema(res) = (2.3970281316847327e-10, 0.002487395693003118)
 10.626086 seconds (9.80 M allocations: 1.908 GiB, 9.45% gc time, 30.88%
compilation time)
```

[25]:
```
tol = 1e-6;
```

[26]:
```
Lambda, X, res = @time Basic_LOBPCG(A, B, X0, nev, T=T, tol=tol);
```

```
it = 0, k = 0
extrema(res) = (0.3300303655975059, 1.537769678077098)
it = 1, k = 0
extrema(res) = (0.4049282907328257, 21.22165503549347)
it = 2, k = 0
```

```
extrema(res) = (0.16200941298647892, 7.182750266997795)
it = 3, k = 0
extrema(res) = (0.07349862403277621, 11.437971535689176)
it = 4, k = 0
extrema(res) = (0.020491807123343607, 21.65946790346044)
it = 5, k = 0
extrema(res) = (0.006833383721195593, 40.78220661300664)
it = 6, k = 0
extrema(res) = (0.0022527567539516215, 40.15919254360722)
it = 7, k = 0
extrema(res) = (0.0006738751143275161, 13.563516352095787)
it = 8, k = 0
extrema(res) = (0.00019891735013226718, 4.093336436417616)
it = 9, k = 0
extrema(res) = (7.339932070691965e-5, 1.2290927245690366)
it = 10, k = 0
extrema(res) = (2.229155583161873e-5, 0.3003242827551401)
it = 11, k = 0
extrema(res) = (6.2023309211219835e-6, 0.0776289064051738)
it = 12, k = 0
extrema(res) = (1.8169954512491298e-6, 0.04169199431181375)
it = 13, k = 0
extrema(res) = (6.222493687609023e-7, 0.029629747719154168)
it = 14, k = 0
extrema(res) = (1.9252518379901585e-7, 0.02136143179346194)
it = 15, k = 0
extrema(res) = (6.045279500132419e-8, 0.01469914303905413)
it = 16, k = 0
extrema(res) = (1.95590358740195e-8, 0.010048497551251728)
it = 17, k = 0
extrema(res) = (6.0483263809570256e-9, 0.007269042372730877)
it = 18, k = 0
extrema(res) = (2.208494237893491e-9, 0.005069785622807576)
it = 19, k = 0
extrema(res) = (7.294482380356976e-10, 0.0038852428578237726)
it = 20, k = 0
extrema(res) = (6.571463669099657e-10, 0.0026547794157476714)
it = 21, k = 0
extrema(res) = (1.8611396487269212e-9, 0.002291650578674644)
it = 22, k = 0
```

```
PosDefException: matrix is not positive definite; Factorization failed.

Stacktrace:
  [1] checkpositivedefinite
    @ ~/.julia/juliaup/julia-1.11.5+0.x64.linux.gnu/share/julia/stdlib/v1.11/
  ↪LinearAlgebra/src/factorization.jl:68 [inlined]
```

```
 [2] #cholesky!#163
   @ ~/.julia/juliaup/julia-1.11.5+0.x64.linux.gnu/share/julia/stdlib/v1.11/
↪LinearAlgebra/src/cholesky.jl:269 [inlined]
 [3] cholesky! (repeats 2 times)
   @ ~/.julia/juliaup/julia-1.11.5+0.x64.linux.gnu/share/julia/stdlib/v1.11/
↪LinearAlgebra/src/cholesky.jl:267 [inlined]
 [4] _cholesky
   @ ~/.julia/juliaup/julia-1.11.5+0.x64.linux.gnu/share/julia/stdlib/v1.11/
↪LinearAlgebra/src/cholesky.jl:411 [inlined]
 [5] cholesky (repeats 2 times)
   @ ~/.julia/juliaup/julia-1.11.5+0.x64.linux.gnu/share/julia/stdlib/v1.11/
↪LinearAlgebra/src/cholesky.jl:401 [inlined]
 [6] RR(X::Matrix{Float64}, Z::SubArray{Float64, 2, Matrix{Float64}, Tuple{Base.
↪Slice{Base.OneTo{Int64}}, UnitRange{Int64}}, true}, P::SubArray{Float64, 2,␣
↪Matrix{Float64}, Tuple{Base.Slice{Base.OneTo{Int64}}, UnitRange{Int64}},␣
↪true}, AX::Matrix{Float64}, AZ::SubArray{Float64, 2, Matrix{Float64},␣
↪Tuple{Base.Slice{Base.OneTo{Int64}}, UnitRange{Int64}}, true}, AP::
↪SubArray{Float64, 2, Matrix{Float64}, Tuple{Base.Slice{Base.OneTo{Int64}},␣
↪UnitRange{Int64}}, true}, BZ::SubArray{Float64, 2, Matrix{Float64}, Tuple{Base.
↪Slice{Base.OneTo{Int64}}, UnitRange{Int64}}, true}, BP::SubArray{Float64, 2,␣
↪Matrix{Float64}, Tuple{Base.Slice{Base.OneTo{Int64}}, UnitRange{Int64}},␣
↪true}, check_L_cond::Bool, Lambda::Vector{Float64})
   @ Main ./In[4]:53
 [7] RR (repeats 2 times)
   @ ./In[4]:17 [inlined]
 [8] Basic_LOBPCG(A::SparseMatrixCSC{Float64, Int64}, B::
↪SparseMatrixCSC{Float64, Int64}, X0::Matrix{Float64}, nev::Int64; T::BJop,␣
↪itmax::Int64, tol::Float64, A_products::Symbol, B_products::Symbol, debug::
↪Bool)
   @ Main ./In[5]:95
 [9] top-level scope
   @ ./timing.jl:581 [inlined]
[10] top-level scope
   @ ./In[26]:0
```

[27]:
```
Lambda, X, res = @time BLOPEX_LOBPCG(A, B, X0, nev, T=T, tol=tol);
```

```
it = 0, k = 0
extrema(res) = (0.33003036559751003, 1.537769678077147)
it = 1, k = 0
extrema(res) = (11.404074965724426, 2.6078961976521253e6)
it = 2, k = 0
extrema(res) = (0.3711103795873485, 18.749168921901187)
it = 3, k = 0
extrema(res) = (0.09852559298854692, 6.625905865112871)
it = 4, k = 0
extrema(res) = (0.05026800364400092, 16.89217878895544)
it = 5, k = 0
extrema(res) = (0.014242500881280524, 28.993468697005998)
it = 6, k = 0
```

```
extrema(res) = (0.004520129533218506, 39.83958874783002)
it = 7, k = 0
extrema(res) = (0.0012449420832257626, 21.20008534463295)
it = 8, k = 0
extrema(res) = (0.00037167833369572325, 6.883051783160938)
it = 9, k = 0
extrema(res) = (0.00014279062602572977, 2.384738327253412)
it = 10, k = 0
extrema(res) = (4.535173043927379e-5, 0.8105860568995963)
it = 11, k = 0
extrema(res) = (1.2746859352015564e-5, 0.24888402766968515)
it = 12, k = 0
extrema(res) = (3.968067259637595e-6, 0.0704623925557094)
it = 13, k = 0
extrema(res) = (1.1908917622097085e-6, 0.03763833199403787)
it = 14, k = 0
extrema(res) = (4.055793303742991e-7, 0.02825674075647867)
it = 15, k = 0
extrema(res) = (1.314630868061625e-7, 0.022741493667800844)
it = 16, k = 0
extrema(res) = (4.3432607941561784e-8, 0.018202379251052527)
it = 17, k = 0
extrema(res) = (1.462573290710571e-8, 0.014379698834286837)
it = 18, k = 0
extrema(res) = (5.242354889532622e-9, 0.01436708838501901)
it = 19, k = 0
extrema(res) = (1.8671455356112838e-9, 0.012246850960368143)
it = 20, k = 0
extrema(res) = (5.739001254458116e-10, 0.009073836144584015)
it = 21, k = 0
extrema(res) = (2.0988740241554377e-10, 0.007699610238526502)
it = 22, k = 0
extrema(res) = (7.85530559889718e-11, 0.006748458924427619)
it = 23, k = 1
extrema(res) = (4.3082856859813924e-11, 0.00529295475436949)
  4.242623 seconds (14.34 k allocations: 922.205 MiB, 13.77% gc time)
```

[28]:
```
Lambda, X, res = @time Ortho_LOBPCG(A, B, X0, nev, T=T, tol=tol);
```

```
it = 0, k = 0
extrema(res) = (0.3300303655975059, 1.537769678077098)
it = 1, k = 0
extrema(res) = (0.40492829073269426, 21.221655035487384)
it = 2, k = 0
extrema(res) = (0.16200941298757182, 7.182750266996059)
it = 3, k = 0
extrema(res) = (0.07349862403046775, 11.437971535689487)
it = 4, k = 0
```

```
extrema(res) = (0.02049180712031476, 21.659467904094267)
it = 5, k = 0
extrema(res) = (0.006833383665072171, 40.78220661675316)
it = 6, k = 0
extrema(res) = (0.0022527568092916674, 40.159190790099515)
it = 7, k = 0
extrema(res) = (0.0006738751071814033, 13.56351530219227)
it = 8, k = 0
extrema(res) = (0.0001989173502197883, 4.093336298469632)
it = 9, k = 0
extrema(res) = (7.339932281283249e-5, 1.2290930325994853)
it = 10, k = 0
extrema(res) = (2.2291552638477075e-5, 0.30032423179723855)
it = 11, k = 0
extrema(res) = (6.202334290113706e-6, 0.07762887827692491)
it = 12, k = 0
extrema(res) = (1.8169929375997989e-6, 0.04169214824676901)
it = 13, k = 0
extrema(res) = (6.222485630247953e-7, 0.029629923556402375)
it = 14, k = 0
extrema(res) = (1.9252450500574176e-7, 0.021361611876170774)
it = 15, k = 0
extrema(res) = (6.044935730316275e-8, 0.01469704121625765)
it = 16, k = 0
extrema(res) = (1.957094362903719e-8, 0.010034446355838679)
it = 17, k = 0
extrema(res) = (6.0620105741208605e-9, 0.00726870571749568)
it = 18, k = 0
extrema(res) = (2.142250976118473e-9, 0.005103339471330485)
it = 19, k = 0
extrema(res) = (6.785992631756814e-10, 0.0037844693814731278)
it = 20, k = 0
extrema(res) = (2.52726160871652e-10, 0.0024346738828153486)
it = 21, k = 0
extrema(res) = (8.624309356095311e-11, 0.0021370131807835783)
it = 22, k = 0
extrema(res) = (3.4497106548853246e-11, 0.001787228845901778)
  5.507670 seconds (17.97 k allocations: 1.044 GiB, 11.00% gc time)
```

[29]:
```
Lambda, X, res = @time Skip_ortho_LOBPCG(A, B, X0, nev, T=T, tol=tol);
```

```
it = 0, k = 0
extrema(res) = (0.3300303655975059, 1.537769678077098)
it = 1, k = 0, skipOrtho = true
extrema(res) = (0.4049282907328257, 21.22165503549347)
it = 2, k = 0, skipOrtho = true
extrema(res) = (0.16200941298626606, 7.182750266996343)
it = 3, k = 0, skipOrtho = true
```

```
extrema(res) = (0.07349862403494663, 11.437971535686156)
it = 4, k = 0, skipOrtho = true
extrema(res) = (0.020491807116805875, 21.659467904033058)
it = 5, k = 0, skipOrtho = true
extrema(res) = (0.006833383659256521, 40.78220661697174)
it = 6, k = 0, skipOrtho = true
extrema(res) = (0.002252756809008092, 40.1591907896633)
it = 7, k = 0, skipOrtho = true
extrema(res) = (0.000673875106755757, 13.56351530186385)
it = 8, k = 0, skipOrtho = true
extrema(res) = (0.0001989173500304196, 4.0933362985239246)
it = 9, k = 0, skipOrtho = true
extrema(res) = (7.339932275009455e-5, 1.22909303225764)
it = 10, k = 0, skipOrtho = true
extrema(res) = (2.229155253313179e-5, 0.3003242337467159)
it = 11, k = 0, skipOrtho = true
extrema(res) = (6.202334473199743e-6, 0.07762887830347989)
it = 12, k = 0, skipOrtho = true
extrema(res) = (1.816992535656678e-6, 0.04169215654096534)
it = 13, k = 0, skipOrtho = true
extrema(res) = (6.222489849144442e-7, 0.02962989856092638)
it = 14, k = 0, skipOrtho = true
extrema(res) = (1.9252399130548884e-7, 0.021361588389590423)
it = 15, k = 0, skipOrtho = true
extrema(res) = (6.044877589218183e-8, 0.014696430609415658)
it = 16, k = 0, skipOrtho = true
extrema(res) = (1.9565880239618814e-8, 0.010033382403694587)
it = 17, k = 0, skipOrtho = true
extrema(res) = (6.062828533337184e-9, 0.0072624650212738204)
it = 18, k = 0, skipOrtho = true
extrema(res) = (2.1370139176285207e-9, 0.005080564087687539)
it = 19, k = 0, skipOrtho = true
extrema(res) = (6.702640376390765e-10, 0.003866767420569932)
it = 20, k = 0, skipOrtho = true
extrema(res) = (2.3970281316847327e-10, 0.002487395693003118)
it = 21, k = 0, skipOrtho = true
extrema(res) = (9.639764908705398e-11, 0.0021781498850046385)
it = 22, k = 0, skipOrtho = true
extrema(res) = (7.516165751988193e-11, 0.0018536707094801788)
it = 23, k = 0, skipOrtho = true
extrema(res) = (7.332662597985692e-11, 0.001118580544218256)
it = 24, k = 0, skipOrtho = true
extrema(res) = (5.3037991479784636e-11, 0.0007628397397518135)
it = 25, k = 0, skipOrtho = false
extrema(res) = (4.6127297814116035e-11, 0.0006862361562914559)
it = 26, k = 0, skipOrtho = false
extrema(res) = (2.918962449932645e-11, 0.0006107231243253201)
it = 27, k = 0, skipOrtho = false
```

```
extrema(res) = (2.346800334406554e-11, 0.0004529809696884631)
it = 28, k = 0, skipOrtho = false
extrema(res) = (1.9321083924334917e-11, 0.00033844135890871924)
it = 29, k = 0, skipOrtho = false
extrema(res) = (1.6962802188002774e-11, 0.00020071015411547512)
it = 30, k = 0, skipOrtho = false
extrema(res) = (1.6326451460657985e-11, 0.00010446356421312829)
it = 31, k = 0, skipOrtho = false
extrema(res) = (9.74924592797299e-12, 5.9503490311807725e-5)
it = 32, k = 0, skipOrtho = false
extrema(res) = (8.263255262543558e-12, 2.0796109734948676e-5)
it = 33, k = 0, skipOrtho = false
extrema(res) = (7.566604867111807e-12, 9.307156380922056e-6)
it = 34, k = 0, skipOrtho = false
extrema(res) = (6.8875408080080094e-12, 4.1886783669034105e-6)
it = 35, k = 0, skipOrtho = false
extrema(res) = (6.774777095808773e-12, 2.6862551495523123e-6)
it = 36, k = 0, skipOrtho = false
extrema(res) = (6.844410843603399e-12, 2.315240763608643e-6)
it = 37, k = 0, skipOrtho = false
extrema(res) = (6.592859638931927e-12, 2.2283354537901137e-6)
it = 38, k = 0, skipOrtho = false
extrema(res) = (7.835259278130832e-12, 2.159192440807991e-6)
it = 39, k = 0, skipOrtho = false
extrema(res) = (7.891745117948363e-12, 2.120649759095954e-6)
it = 40, k = 0, skipOrtho = false
extrema(res) = (7.350157696378084e-12, 2.0731551301673995e-6)
it = 41, k = 0, skipOrtho = false
extrema(res) = (6.951919072800876e-12, 2.0078199398924886e-6)
it = 42, k = 2, skipOrtho = false
extrema(res) = (6.3969325378675725e-12, 1.9722616338717263e-6)
 15.848513 seconds (37.77 k allocations: 3.128 GiB, 10.84% gc time)
```