Numerical Linear Algebra for Computational Sciences and Information Engineering
CITHN2006

Locally Optimal Block Preconditioned Conjugate Gradient

by Nicolas Venkovic

Chair of Computational Mathematics
School of Computation, Information and Technology (CIT)
Technical University of Munich, Germany

1. *Implement LOBPCG.*

[1]:
```
using LinearAlgebra, Printf, SparseArrays
using MatrixMarket: mmread
```

[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])
  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]:
```
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]

  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]:
```
function 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;
```

2. *Implement BLOPEX.*

[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]:
```
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)
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;
```

3. *Implement Ortho_LOPBCG.*

[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},
                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;
```

[13]:
```
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
  Z .= D * Z * diagm(Theta.^(-.5))
  U .= U * Z
end;
```

[14]:
```
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

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

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

[15]:
```
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;
```

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;
```

4. *Use the matrices* `bcsstk12` *and* `bcsstm12` *from the SuiteSparse collection for A and B, respectively. Then, using $m = 200$, benchmark each algorithm by monitoring the convergence of the 10 smallest generalized eigenvalues, first with $\tau = 10^{-5}$, then with $\tau = 10^{-6}$. Use a block Jacobi preconditioner with 10 diagonal blocks based on A. What do you observe?*

[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]:
```
A = mmread("matrices/bcsstk12.mtx");
B = mmread("matrices/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 LOBPCG(A, B, X0, nev, T=T, tol=tol);
```

```
it = 0, k = 0
extrema(res) = (0.33607947834742297, 1.5393299467290997)
it = 1, k = 0
extrema(res) = (0.40376771844474013, 20.72425270413949)
it = 2, k = 0
extrema(res) = (0.13951984528566502, 7.329917245994145)
it = 3, k = 0
extrema(res) = (0.04731417724554892, 10.48922173582916)
it = 4, k = 0
extrema(res) = (0.01892768712542413, 24.05552115029743)
it = 5, k = 0
extrema(res) = (0.0060489971601822585, 45.342861872023846)
it = 6, k = 0
extrema(res) = (0.001723525436572569, 37.31314660803577)
it = 7, k = 0
extrema(res) = (0.0006031177573116454, 12.383406040361592)
it = 8, k = 0
extrema(res) = (0.00019627314816687548, 3.554451686524213)
it = 9, k = 0
extrema(res) = (6.364749486514656e-5, 0.9631513048655007)
it = 10, k = 0
extrema(res) = (2.1251732826825454e-5, 0.25739489320049314)
it = 11, k = 0
extrema(res) = (5.732146767416478e-6, 0.07338081438821058)
it = 12, k = 0
extrema(res) = (1.6436028087410895e-6, 0.032646367272165765)
it = 13, k = 0
extrema(res) = (5.86462587118075e-7, 0.024672578984468455)
it = 14, k = 0
extrema(res) = (1.6171771903722684e-7, 0.019631801804468583)
it = 15, k = 0
extrema(res) = (4.825675152430617e-8, 0.014792592057024635)
it = 16, k = 0
extrema(res) = (1.6147650242220444e-8, 0.012006025434441738)
it = 17, k = 0
extrema(res) = (6.015661173379572e-9, 0.01039303704494304)
it = 18, k = 0
extrema(res) = (2.3284858826513074e-9, 0.008430285669504822)
```

```
it = 19, k = 0
extrema(res) = (8.644826726572888e-10, 0.006102160310926589)
  7.607693 seconds (14.54 M allocations: 1.654 GiB, 13.70% gc time, 59.87%
compilation time)
```

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

```
it = 0, k = 0
extrema(res) = (0.3360794783474341, 1.539329946729174)
it = 1, k = 0
```

```
ArgumentError: matrix contains Infs or NaNs

Stacktrace:
  [1] chkuplofinite(A::Matrix{Float64}, uplo::Char)
    @ LinearAlgebra.LAPACK ~/.julia/juliaup/julia-1.12.1+0.x64.linux.gnu/share/
↪julia/stdlib/v1.12/LinearAlgebra/src/lapack.jl:118
  [2] syevr!(jobz::Char, range::Char, uplo::Char, A::Matrix{Float64}, vl::
↪Float64, vu::Float64, il::Int64, iu::Int64, abstol::Float64)
    @ LinearAlgebra.LAPACK ~/.julia/juliaup/julia-1.12.1+0.x64.linux.gnu/share/
↪julia/stdlib/v1.12/LinearAlgebra/src/lapack.jl:5388
  [3] #eigen!#252
    @ ~/.julia/juliaup/julia-1.12.1+0.x64.linux.gnu/share/julia/stdlib/v1.12/
↪LinearAlgebra/src/symmetriceigen.jl:24 [inlined]
  [4] eigen!
    @ ~/.julia/juliaup/julia-1.12.1+0.x64.linux.gnu/share/julia/stdlib/v1.12/
↪LinearAlgebra/src/symmetriceigen.jl:18 [inlined]
  [5] #_eigen#254
    @ ~/.julia/juliaup/julia-1.12.1+0.x64.linux.gnu/share/julia/stdlib/v1.12/
↪LinearAlgebra/src/symmetriceigen.jl:61 [inlined]
  [6] _eigen
    @ ~/.julia/juliaup/julia-1.12.1+0.x64.linux.gnu/share/julia/stdlib/v1.12/
↪LinearAlgebra/src/symmetriceigen.jl:59 [inlined]
  [7] #eigen#253
    @ ~/.julia/juliaup/julia-1.12.1+0.x64.linux.gnu/share/julia/stdlib/v1.12/
↪LinearAlgebra/src/symmetriceigen.jl:55 [inlined]
  [8] eigen
    @ ~/.julia/juliaup/julia-1.12.1+0.x64.linux.gnu/share/julia/stdlib/v1.12/
↪LinearAlgebra/src/symmetriceigen.jl:54 [inlined]
  [9] RR_BLOPEX(X::Matrix{Float64}, Z::Matrix{Float64}, AX::Matrix{Float64}, AZ :
↪Matrix{Float64}, BZ::Matrix{Float64}, Lambda::Vector{Float64})
    @ Main ./In[7]:35
 [10] RR_BLOPEX
    @ ./In[7]:13 [inlined]
 [11] BLOPEX_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[9]:94
[12] macro expansion
   @ ./timing.jl:689 [inlined]
[13] top-level scope
   @ ./In[22]:0
[14] eval(m::Module, e::Any)
   @ Core ./boot.jl:489
```

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

```
it = 0, k = 0
extrema(res) = (0.33607947834742297, 1.5393299467290997)
it = 1, k = 0
extrema(res) = (0.40376771844492365, 20.72425270413976)
it = 2, k = 0
extrema(res) = (0.13951984528582134, 7.32991724599528)
it = 3, k = 0
extrema(res) = (0.047314177267022535, 10.48922173586115)
it = 4, k = 0
extrema(res) = (0.01892768713542251, 24.0555211510122)
it = 5, k = 0
extrema(res) = (0.006048997161806734, 45.3428618848359)
it = 6, k = 0
extrema(res) = (0.0017235254366534507, 37.31314660860034)
it = 7, k = 0
extrema(res) = (0.0006031177569128225, 12.383405991914115)
it = 8, k = 0
extrema(res) = (0.00019627314996056763, 3.5544516783715774)
it = 9, k = 0
extrema(res) = (6.364749404893702e-5, 0.9631513086451193)
it = 10, k = 0
extrema(res) = (2.1251734055866656e-5, 0.2573948879504246)
it = 11, k = 0
extrema(res) = (5.732146549300889e-6, 0.07338082243226848)
it = 12, k = 0
extrema(res) = (1.6436029313097627e-6, 0.03264637990383193)
it = 13, k = 0
extrema(res) = (5.864650898366509e-7, 0.024672462867950006)
it = 14, k = 0
extrema(res) = (1.6170890496842498e-7, 0.019631461573792426)
it = 15, k = 0
extrema(res) = (4.82601767478479e-8, 0.014794202870996231)
it = 16, k = 0
extrema(res) = (1.6050350570866934e-8, 0.011968468952121543)
it = 17, k = 0
extrema(res) = (6.05886239944976e-9, 0.010372500394596923)
it = 18, k = 0
```

```
extrema(res) = (2.4190417617332986e-9, 0.008469588271927437)
it = 19, k = 0
extrema(res) = (7.858661551793364e-10, 0.0062672684039373735)
it = 20, k = 1
extrema(res) = (2.755187265984946e-10, 0.005072077406598309)
  4.753264 seconds (4.78 M allocations: 1.097 GiB, 7.00% gc time, 38.13%
compilation time)
```

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

```
it = 0, k = 0
extrema(res) = (0.33607947834742297, 1.5393299467290997)
it = 1, k = 0, skipOrtho = true
extrema(res) = (0.40376771844474013, 20.72425270413949)
it = 2, k = 0, skipOrtho = true
extrema(res) = (0.1395198452857611, 7.3299172459965245)
it = 3, k = 0, skipOrtho = true
extrema(res) = (0.04731417726909077, 10.489221735863138)
it = 4, k = 0, skipOrtho = true
extrema(res) = (0.018927687133066886, 24.055521151056478)
it = 5, k = 0, skipOrtho = true
extrema(res) = (0.006048997157275548, 45.34286188348852)
it = 6, k = 0, skipOrtho = true
extrema(res) = (0.0017235254331841791, 37.31314660900157)
it = 7, k = 0, skipOrtho = true
extrema(res) = (0.0006031177535866891, 12.38340599190203)
it = 8, k = 0, skipOrtho = true
extrema(res) = (0.00019627315012762453, 3.554451678498725)
it = 9, k = 0, skipOrtho = true
extrema(res) = (6.36474943740306e-5, 0.9631513085519576)
it = 10, k = 0, skipOrtho = true
extrema(res) = (2.125173397166655e-5, 0.25739488961654183)
it = 11, k = 0, skipOrtho = true
extrema(res) = (5.732146377149203e-6, 0.07338082048459042)
it = 12, k = 0, skipOrtho = true
extrema(res) = (1.6436028997015333e-6, 0.0326463778224355)
it = 13, k = 0, skipOrtho = true
extrema(res) = (5.864654422441583e-7, 0.02467245990113184)
it = 14, k = 0, skipOrtho = true
extrema(res) = (1.6170879960829366e-7, 0.019631454748245878)
it = 15, k = 0, skipOrtho = true
extrema(res) = (4.825943691934804e-8, 0.014794214542708576)
it = 16, k = 0, skipOrtho = true
extrema(res) = (1.6052039551553434e-8, 0.011964397799745811)
it = 17, k = 0, skipOrtho = true
extrema(res) = (6.061059789858141e-9, 0.010371418751684047)
it = 18, k = 0, skipOrtho = true
extrema(res) = (2.4194318259944435e-9, 0.008467025645307167)
```

```
it = 19, k = 0, skipOrtho = true
extrema(res) = (7.877784818462774e-10, 0.006362546241276976)
it = 20, k = 1, skipOrtho = false
extrema(res) = (2.8635770597872997e-10, 0.005066277483205903)
  5.325768 seconds (3.05 M allocations: 1.530 GiB, 9.46% gc time, 24.18%
compilation time)
```

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

[25]:
```
Lambda, X, res = @time LOBPCG(A, B, X0, nev, T=T, tol=tol);
```

```
it = 0, k = 0
extrema(res) = (0.33607947834742297, 1.5393299467290997)
it = 1, k = 0
extrema(res) = (0.40376771844474013, 20.72425270413949)
it = 2, k = 0
extrema(res) = (0.13951984528566502, 7.329917245994145)
it = 3, k = 0
extrema(res) = (0.04731417724554892, 10.48922173582916)
it = 4, k = 0
extrema(res) = (0.01892768712542413, 24.05552115029743)
it = 5, k = 0
extrema(res) = (0.0060489971601822585, 45.342861872023846)
it = 6, k = 0
extrema(res) = (0.001723525436572569, 37.31314660803577)
it = 7, k = 0
extrema(res) = (0.0006031177573116454, 12.383406040361592)
it = 8, k = 0
extrema(res) = (0.00019627314816687548, 3.554451686524213)
it = 9, k = 0
extrema(res) = (6.364749486514656e-5, 0.9631513048655007)
it = 10, k = 0
extrema(res) = (2.1251732826825454e-5, 0.25739489320049314)
it = 11, k = 0
extrema(res) = (5.732146767416478e-6, 0.07338081438821058)
it = 12, k = 0
extrema(res) = (1.6436028087410895e-6, 0.032646367272165765)
it = 13, k = 0
extrema(res) = (5.86462587118075e-7, 0.024672578984468455)
it = 14, k = 0
extrema(res) = (1.6171771903722684e-7, 0.019631801804468583)
it = 15, k = 0
extrema(res) = (4.825675152430617e-8, 0.014792592057024635)
it = 16, k = 0
extrema(res) = (1.6147650242220444e-8, 0.012006025434441738)
it = 17, k = 0
extrema(res) = (6.015661173379572e-9, 0.01039303704494304)
it = 18, k = 0
```

```
extrema(res) = (2.3284858826513074e-9, 0.008430285669504822)
it = 19, k = 0
extrema(res) = (8.644826726572888e-10, 0.006102160310926589)
  2.505907 seconds (10.05 k allocations: 1012.008 MiB, 14.39% gc time)
```

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

```
it = 0, k = 0
extrema(res) = (0.3360794783474341, 1.539329946729174)
it = 1, k = 0
extrema(res) = (8.574940554564954, 4.88966342064059e6)
it = 2, k = 0
extrema(res) = (0.3172996624348516, 17.751776077080827)
it = 3, k = 0
extrema(res) = (0.08406327486325064, 5.9983899363167055)
it = 4, k = 0
extrema(res) = (0.0419653955912113, 16.43113010749454)
it = 5, k = 0
extrema(res) = (0.013715790049867184, 32.29748153772814)
it = 6, k = 0
extrema(res) = (0.003590865037086672, 41.39482499438359)
it = 7, k = 0
extrema(res) = (0.0009417040288032838, 21.811074196057614)
it = 8, k = 0
extrema(res) = (0.00032961463788262155, 7.620916921020385)
it = 9, k = 0
extrema(res) = (0.00012444528792675375, 2.091170371997852)
it = 10, k = 0
extrema(res) = (3.934527530860459e-5, 0.7208570425274843)
it = 11, k = 0
extrema(res) = (1.2748202525496579e-5, 0.19407429773651905)
it = 12, k = 0
extrema(res) = (3.3770902044442593e-6, 0.050871868588969964)
it = 13, k = 0
extrema(res) = (1.1730058834939912e-6, 0.016830646705195827)
it = 14, k = 0
extrema(res) = (4.402529838315922e-7, 0.010638447204842864)
it = 15, k = 0
extrema(res) = (1.358436323220159e-7, 0.006679627532702876)
it = 16, k = 0
extrema(res) = (4.4312679468663106e-8, 0.004829423796525399)
it = 17, k = 0
extrema(res) = (1.3402615387198135e-8, 0.003867311138669768)
it = 18, k = 0
extrema(res) = (5.114023324998047e-9, 0.003410412086830769)
it = 19, k = 0
extrema(res) = (2.0410622634019814e-9, 0.00326776864433167)
it = 20, k = 0
```

```
extrema(res) = (6.294546198258045e-10, 0.0022570693709404893)
  1.942680 seconds (9.63 k allocations: 749.585 MiB, 10.82% gc time)
```

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

```
it = 0, k = 0
extrema(res) = (0.33607947834742297, 1.5393299467290997)
it = 1, k = 0
extrema(res) = (0.40376771844492365, 20.72425270413976)
it = 2, k = 0
extrema(res) = (0.13951984528582134, 7.32991724599528)
it = 3, k = 0
extrema(res) = (0.047314177267022535, 10.48922173586115)
it = 4, k = 0
extrema(res) = (0.01892768713542251, 24.0555211510122)
it = 5, k = 0
extrema(res) = (0.006048997161806734, 45.3428618848359)
it = 6, k = 0
extrema(res) = (0.0017235254366534507, 37.31314660860034)
it = 7, k = 0
extrema(res) = (0.0006031177569128225, 12.383405991914115)
it = 8, k = 0
extrema(res) = (0.00019627314996056763, 3.5544516783715774)
it = 9, k = 0
extrema(res) = (6.364749404893702e-5, 0.9631513086451193)
it = 10, k = 0
extrema(res) = (2.1251734055866656e-5, 0.2573948879504246)
it = 11, k = 0
extrema(res) = (5.732146549300889e-6, 0.07338082243226848)
it = 12, k = 0
extrema(res) = (1.6436029313097627e-6, 0.03264637990383193)
it = 13, k = 0
extrema(res) = (5.864650898366509e-7, 0.024672462867950006)
it = 14, k = 0
extrema(res) = (1.6170890496842498e-7, 0.019631461573792426)
it = 15, k = 0
extrema(res) = (4.82601767478479e-8, 0.014794202870996231)
it = 16, k = 0
extrema(res) = (1.6050350570866934e-8, 0.011968468952121543)
it = 17, k = 0
extrema(res) = (6.05886239944976e-9, 0.010372500394596923)
it = 18, k = 0
extrema(res) = (2.4190417617332986e-9, 0.008469588271927437)
it = 19, k = 0
extrema(res) = (7.858661551793364e-10, 0.0062672684039373735)
it = 20, k = 1
extrema(res) = (2.755187265984946e-10, 0.005072077406598309)
  3.114156 seconds (13.97 k allocations: 892.570 MiB, 9.63% gc time)
```

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

```
it = 0, k = 0
extrema(res) = (0.33607947834742297, 1.5393299467290997)
it = 1, k = 0, skipOrtho = true
extrema(res) = (0.40376771844474013, 20.72425270413949)
it = 2, k = 0, skipOrtho = true
extrema(res) = (0.1395198452857611, 7.3299172459965245)
it = 3, k = 0, skipOrtho = true
extrema(res) = (0.04731417726909077, 10.489221735863138)
it = 4, k = 0, skipOrtho = true
extrema(res) = (0.018927687133066886, 24.055521151056478)
it = 5, k = 0, skipOrtho = true
extrema(res) = (0.006048997157275548, 45.34286188348852)
it = 6, k = 0, skipOrtho = true
extrema(res) = (0.0017235254331841791, 37.31314660900157)
it = 7, k = 0, skipOrtho = true
extrema(res) = (0.0006031177535866891, 12.38340599190203)
it = 8, k = 0, skipOrtho = true
extrema(res) = (0.00019627315012762453, 3.554451678498725)
it = 9, k = 0, skipOrtho = true
extrema(res) = (6.36474943740306e-5, 0.9631513085519576)
it = 10, k = 0, skipOrtho = true
extrema(res) = (2.125173397166655e-5, 0.25739488961654183)
it = 11, k = 0, skipOrtho = true
extrema(res) = (5.732146377149203e-6, 0.07338082048459042)
it = 12, k = 0, skipOrtho = true
extrema(res) = (1.6436028997015333e-6, 0.0326463778224355)
it = 13, k = 0, skipOrtho = true
extrema(res) = (5.864654422441583e-7, 0.02467245990113184)
it = 14, k = 0, skipOrtho = true
extrema(res) = (1.6170879960829366e-7, 0.019631454748245878)
it = 15, k = 0, skipOrtho = true
extrema(res) = (4.825943691934804e-8, 0.014794214542708576)
it = 16, k = 0, skipOrtho = true
extrema(res) = (1.6052039551553434e-8, 0.011964397799745811)
it = 17, k = 0, skipOrtho = true
extrema(res) = (6.061059789858141e-9, 0.010371418751684047)
it = 18, k = 0, skipOrtho = true
extrema(res) = (2.4194318259944435e-9, 0.008467025645307167)
it = 19, k = 0, skipOrtho = true
extrema(res) = (7.877784818462774e-10, 0.006362546241276976)
it = 20, k = 1, skipOrtho = false
extrema(res) = (2.8635770597872997e-10, 0.005066277483205903)
  3.885528 seconds (13.47 k allocations: 1.391 GiB, 10.52% gc time)
```