Package ‘PRIMME’

July 25, 2019

Type Package
Title Eigenvalues and Singular Values and Vectors from Large Matrices
Version 3.0-0
Date 2019-07-25
Maintainer Eloy Romero <eloy@cs.wm.edu>
Description R interface to PRIMME, a C library for computing a few
eigenvalues and their corresponding eigenvectors of a real symmetric or complex
Hermitian matrix. It can also compute singular values and vectors of a square
or rectangular matrix. It can find largest, smallest, or interior
singular/eigenvalues and can use preconditioning to accelerate convergence.

URL https://github.com/primme/primme
BugReports https://github.com/primme/primme/issues
Imports Rcpp
LinkingTo Rcpp, Matrix
Suggests Matrix
SystemRequirements A POSIX system. Currently Linux and OS X are known
to work. GNU make.
NeedsCompilation yes
License GPL-3
Encoding UTF-8
RoxygenNote 6.1.1
Author Eloy Romero [aut, cre],
Andreas Stathopoulos [aut],
College of William & Mary [cph]

Repository CRAN
Date/Publication 2019-07-25 09:50:02 UTC

R topics documented:

eigs_sym ................................................................. 2
svds ................................................................. 5
eigs_sym

Find a few eigenvalues and vectors on large, sparse Hermitian matrix

Description

Compute a few eigenpairs from a specified region (the largest, the smallest, the closest to a point) on a symmetric/Hermitian matrix using PRIMME [1]. Generalized symmetric/Hermitian problem is also supported. Only the matrix-vector product of the matrix is required. The used method is usually faster than a direct method (such as `eigen`) if seeking a few eigenpairs and the matrix-vector product is cheap. For accelerating the convergence consider to use preconditioning and/or educated initial guesses.

Usage

```r
eigs_sym(A, NEig = 1, which = "LA", targetShifts = NULL,
        tol = 1e-06, x0 = NULL, ortho = NULL, prec = NULL,
        isreal = NULL, B = NULL, ...)
```

Arguments

- **A**: symmetric/Hermitian matrix or a function with signature f(x) that returns `A %*% x`.
- **NEig**: number of eigenvalues and vectors to seek.
- **which**: which eigenvalues to find:
  - "LA" the largest (rightmost) values;
  - "SA" the smallest (leftmost) values;
  - "LM" the farthest from `targetShifts`;
  - "SM" the closest to `targetShifts`;
  - "CLT" the closest but left to `targetShifts`;
  - "CGT" the closest but greater than `targetShifts`;
- **targetShifts**: vector of numbers the closest values to these points.
- **tol**: the convergence tolerance: \(|Ax - x\lambda| \leq tol\|A\|.
- **x0**: matrix whose columns are educated guesses of the eigenvectors to to find.
- **ortho**: find eigenvectors orthogonal to the space spanned by the columns of this matrix; useful to avoid finding some eigenvalues or to find new solutions.
- **prec**: preclassifier used to accelerated the convergence; usually it is an approximation of the inverse of \(A - \sigma I\) if finding the closest eigenvalues to \(\sigma\). If it is a matrix it is used as \(prec \times x\); otherwise it is used as \(prec(x)\).
- **isreal**: whether \(A \times x\) always returns real number and not complex.
- **B**: symmetric/Hermitian positive definite matrix or a function with signature f(x) that returns \(B \times x\). If given, the function returns the eigenpairs of \((A,B)\).
- **...**: other PRIMME options (see details).
Optional arguments to pass to PRIMME eigensolver (see further details at [2]):

- **method** used by the solver, one of:
  - "DYNAMIC" switches dynamically between DEFAULT_MIN_TIME and DEFAULT_MIN_MATVECS
  - "DEFAULT_MIN_TIME" best method for light matrix-vector product
  - "DEFAULT_MIN_MATVECS" best method for heavy matrix-vector product or precon-
  - "Arnoldi" an Arnoldi not implemented efficiently
  - "GD" classical block Generalized Davidson
  - "GD_plusK" GD+k block GD with recurrence restarting
  - "GD_Olsen_plusK" GD+k with approximate Olsen preconditioning
  - "JD_Olsen_plusK" GD+k, exact Olsen (two preconditioner applications per step)
  - "RQI" Rayleigh Quotient Iteration, also Inverse Iteration if targetShifts is provided
  - "JDQR" original block, Jacobi Davidson
  - "JDQMR" our block JDQMR method (similar to JDCG)
  - "JDQMR_ETol" slight, but efficient JDQMR modification
  - "STEEPEST_DESCENT" equivalent to GD(maxBlockSize,2*maxBlockSize)
  - "LOBPCG_OrthoBasis" equivalent to GD(neig,3*neig)+neig
  - when neig>maxBlockSize
  - "LOBPCG_OrthoBasis_Window" equivalent to GD(maxBlockSize,3*maxBlockSize)+maxBlockSize

- **aNorm** estimation of norm-2 of A, used in convergence test (if not provided, it is estimated as the
  largest eigenvalue in magnitude seen).

- **maxBlockSize** maximum block size (like in subspace iteration or LOBPCG).

- **printLevel** message level reporting, from 0 (no output) to 5 (show all).

- **locking** 1, hard locking; 0, soft locking.

- **maxBasisSize** maximum size of the search subspace.

- **minRestartSize** minimum Ritz vectors to keep in restarting.

- **maxMatveces** maximum number of matrix vector multiplications.

- **maxit** maximum number of outer iterations.

- **scheme** the restart scheme (thick restart by default).

- **maxPrevRetain** number of approximate eigenvectors retained from previous iteration, that are kept after restart.

- **robustShifts** set to true to avoid stagnation.

- **maxInnerIterations** maximum number of inner QMR iterations.

- **LeftQ** use the locked vectors in the left projector.

- **LeftX** use the approx. eigenvector in the left projector.

- **RightQ** use the locked vectors in the right projector.

- **RightX** use the approx. eigenvector in the right projector.

- **SkewQ** use the preconditioned locked vectors in the right projector.

- **SkewX** use the preconditioned approximate eigenvector in the right projector.

- **relTolBase** a legacy from classical JDQR (recommend not use).

- **iseed** an array of four numbers used as a random seed.
Value

   list with the next elements
values  the eigenvalues $\lambda_i$
  vectors  the eigenvectors $x_i$
  rnorms  the residual vector norms $\|Ax_i - \lambda_i Bx_i\|$.  

stats$numMatvecs number of matrix-vector products performed
stats$numPreconds number of preconditioner applications performed
stats$elapsedTime time expended by the eigensolver
stats$timeMatvec time expended in the matrix-vector products
stats$timePrecond time expended in applying the preconditioner
stats$timeOrtho time expended in orthogonalizing
stats$estimateMinEval estimation of the smallest eigenvalue of A
stats$estimateMaxEval estimation of the largest eigenvalue of A
stats$estimateANorm estimation of the norm of A

References

[1] A. Stathopoulos and J. R. McCombs PRIMME: PReconditioned Iterative MultiMethod Eigen-
solver: Methods and software description, ACM Transaction on Mathematical Software Vol. 37,


See Also

eigen for computing all values; svds for computing a few singular values

Examples

A <- diag(1:10)  # the eigenvalues of this matrix are 1:10 and the
    # eigenvectors are the columns of diag(10)
r <- eigs_sym(A, 3);
r$values # the three largest eigenvalues on diag(1:10)
r$vectors # the corresponding approximate eigenvectors
r$rnorms # the corresponding residual norms
r$stats$numMatvecs # total matrix-vector products spend

r <- eigs_sym(A, 3, 'SA') # compute the three smallest values

r <- eigs_sym(A, 3, 2.5) # compute the three closest values to 2.5

r <- eigs_sym(A, 3, 2.5, tol=1e-3);  # compute the values with
    # residual norm <= 1e-3*||A||

B <- diag(rev(1:10));
r <- eigs_sym(A, 3, B=B);  # compute the 3 largest eigenpairs of
    # the generalized problem (A,B)
# Build a Jacobi preconditioner (too convenient for a diagonal matrix!)
# and see how reduce the number matrix-vector products
A <- diag(1:1000)  # we use a larger matrix to amplify the difference
P <- diag(diag(A) - 2.5)
eigs_sym(A, 3, 2.5, tol=1e-3)$stats$numMatvecs
eigs_sym(A, 3, 2.5, tol=1e-3, prec=P)$stats$numMatvecs

# Passing A and the preconditioner as functions
Af <- function(x) (1:100) * x;  # = diag(1:100) %*% x
Pf <- function(x) x / (1:100 - 2.5);  # = solve(diag(1:100 - 2.5), x)
r <- eigs_sym(Af, 3, 2.5, tol=1e-3, prec=Pf, n=100)

# Passing initial guesses
A <- diag(1:1000)  # we use a larger matrix to amplify the difference
x0 <- diag(1,1000,4) + matrix(rnorm(4000), 1000, 4)/100;
eigs_sym(A, 4, "SA", tol=1e-3)$stats$numMatvecs
eigs_sym(A, 4, "SA", tol=1e-3, x0=x0)$stats$numMatvecs

# Passing orthogonal constrain, in this case, already compute eigenvectors
r <- eigs_sym(A, 4, "SA", tol=1e-3); r$values
eigs_sym(A, 4, "SA", tol=1e-3, ortho=r$vectors)$values

---

svds

Find a few singular values and vectors on large, sparse matrix

Description

Compute a few singular triplets from a specified region (the largest, the smallest, the closest to a point) on a matrix using PRIMME [1]. Only the matrix-vector product of the matrix is required. The used method is usually faster than a direct method (such as svd) if seeking few singular values and the matrix-vector product is cheap. For accelerating the convergence consider to use preconditioning and/or educated initial guesses.

Usage

svds(A, NSvals, which = "L", tol = 1e-06, u0 = NULL, v0 = NULL,
orhoun = NULL, orhov = NULL, prec = NULL, isreal = NULL, ...)

Arguments

A matrix or a function with signature f(x, trans) that returns $A %*% x$ when trans == "n" and $t(Conj(A)) %*% x$ when trans == "c".

NSvals number of singular triplets to seek.

which which singular values to find:
"L" the largest values;
"S" the smallest values;
vector of numbers  the closest values to these points.

tol  a triplet \((\sigma, u, v)\) is marked as converged when \(\sqrt{\|Av - \sigma u\|^2 + \|A^*u - \sigma v\|^2} \leq tol\|A\|\) is smaller than \(tol \times \|A\|\), or close to the minimum tolerance that the selected method can achieve.

\(u_0\) matrix whose columns are educated guesses of the left singular vectors to find.

\(v_0\) matrix whose columns are educated guesses of the right singular vectors to find.

orthou  find left singular vectors orthogonal to the space spanned by the columns of this matrix; useful to avoid finding some triplets or to find new solutions.

orthov  find right singular vectors orthogonal to the space spanned by the columns of this matrix.

prec  preconditioner used to accelerated the convergence; it is a named list of matrices or functions such as \(\text{solve}(\text{prec}[\text{[mode]}], x)\) or \(\text{prec}[\text{[mode]}](x)\) return an approximation of \(OP^{-1}x\), where

\[
\begin{align*}
\text{mode} & \quad OP \\
"AHA" & \quad A^*A \\
"AAH" & \quad AA^* \\
"aug" & \quad [0A; A*0]
\end{align*}
\]

The three values haven’t to be set. It is recommended to set "AHA" for matrices with nrow > ncol; "AAH" for matrices with nrow < ncol; and additionally "aug" for tol < 1e-8.

isreal  whether \(A \times x\) always returns real number and not complex.

...  other PRIMME options (see details).

Details

Optional arguments to pass to PRIMME eigensolver (see further details at [2]):

\(a\text{Norm}\) estimation of norm-2 of \(A\), used in convergence test (if not provided, it is estimated as the largest eigenvalue in magnitude seen)

\(\text{maxBlockSize}\) maximum block size (like in subspace iteration or LOBPCG)

\(\text{printLevel}\) message level reporting, from 0 (no output) to 5 (show all)

\(\text{locking}\) 1, hard locking; 0, soft locking

\(\text{maxBasisSize}\) maximum size of the search subspace

\(\text{minRestartSize}\) minimum Ritz vectors to keep in restarting

\(\text{maxMatvecs}\) maximum number of matrix vector multiplications

\(\text{iseed}\) an array of four numbers used as a random seed

\(\text{method}\) which equivalent eigenproblem to solve

"primme_svds_normal_equation"  \(A^*A\) or \(AA^*\)

"primme_svds_augmented"  \([0A^*; A0]\)

"primme_svds_hybrid"  first normal equations and then augmented (default)

\(\text{locking}\) 1, hard locking; 0, soft locking
**svds**

**primmeStage1, primmeStage2** list with options for the first and the second stage solver; see **eigs_sym**

If `method` is "primme_svds_normalequation", the minimum tolerance that can be achieved is \( \|A\|/\sigma \), where \( \epsilon \) is the machine precision. If `method` is "primme_svds_augmented" or "primme_svds_hybrid", the minimum tolerance is \( \|A\|/\epsilon \). However it may not return triplets with singular values smaller than \( \|A\|/\epsilon \).

**Value**

- list with the next elements
  - `d` the singular values \( \sigma_i \)
  - `u` the left singular vectors \( u_i \)
  - `v` the right singular vectors \( v_i \)
  - `rnorms` the residual vector norms \( \sqrt{\|Av - \sigma u\|^2 + \|A^*u - \sigma v\|^2} \)
  - `stats$numMatvecs` matrix-vector products performed
  - `stats$numPreconds` number of preconditioner applications performed
  - `stats$elapsedTime` time expended by the eigensolver
  - `stats$timeMatvec` time expended in the matrix-vector products
  - `stats$timePrecond` time expended in applying the preconditioner
  - `stats$timeOrtho` time expended in orthogonalizing
  - `stats$estimateANorm` estimation of the norm of A

**References**


**See Also**

- `svd` for computing all singular triplets; **eigs_sym** for computing a few eigenvalues and vectors from a symmetric/Hermitian matrix.

**Examples**

```r
A <- diag(1:5,10,5) # the singular values of this matrix are 1:10 and the
# left and right singular vectors are the columns of
# diag(1,100,10) and diag(10), respectively
r <- svds(A, 3);
r$d # the three largest singular values on A
r$u # the corresponding approximate left singular vectors
r$v # the corresponding approximate right singular vectors
r$rnorms # the corresponding residual norms
r$stats$numMatvecs # total matrix-vector products spend
```
r <- svds(A, 3, "S")  # compute the three smallest values

r <- svds(A, 3, 2.5)  # compute the three closest values to 2.5

A <- diag(1:500,500,100)  # we use a larger matrix to amplify the difference
r <- svds(A, 3, 2.5, tol=1e-3);  # compute the values with
r$rnorms  # residual norm <= 1e-3*||A||

# Build the diagonal squared preconditioner
# and see how reduce the number matrix-vector products
P <- diag(colSums(A^2));
svds(A, 3, "S", tol=1e-3)$stats$numMatvecs
svds(A, 3, "S", tol=1e-3, prec=list(AHA=P))$stats$numMatvecs

# Passing A and the preconditioner as functions
Af <- function(x,mode) if (mode == "n") A%*%x else crossprod(A,x);
P = colSums(A^2);
PAHAf <- function(x) x / P;
r <- svds(Af, 3, "S", tol=1e-3, prec=list(AHA=PAHAf), m=500, n=100)

# Passing initial guesses
v0 <- diag(1,100,4) + matrix(rnorm(400), 100, 4)/100;
svds(A, 4, "S", tol=1e-3)$stats$numMatvecs
svds(A, 4, "S", tol=1e-3, v0=v0)$stats$numMatvecs

# Passing orthogonal constrain, in this case, already compute singular vectors
r <- svds(A, 4, "S", tol=1e-3); r$d
svds(A, 4, "S", tol=1e-3, orthov=r$v)$d