

Package ‘sanic’

October 25, 2026

Type Package

Title Solving $Ax = b$ Nimbly in C++

Version 0.1.0

Date 2026-10-21

Author Nikolas Kuschnig [aut, cre] (ORCID:

<<https://orcid.org/0000-0002-6642-2543>>),

Lukas Vashold [ctb] (ORCID: <<https://orcid.org/0000-0002-3562-3414>>),

Dirk Eddelbuettel [ctb] (ORCID:

<<https://orcid.org/0000-0001-6419-907X>>),

Yixuan Qiu [ctb]

Maintainer Nikolas Kuschnig <nikolas.kuschnig@wu.ac.at>

Description

Direct and iterative methods for solving linear systems and computing eigenvalue and singular value decompositions using the Eigen C++ library in R. Supports dense and sparse linear systems with Cholesky, LU, QR, conjugate gradient, stabilized biconjugate gradient, and least-squares conjugate gradient methods. Provides reusable factorizations for repeated solves.

URL <https://github.com/nk027/sanic>

BugReports <https://github.com/nk027/sanic/issues>

Depends R (>= 3.3.0)

Imports Rcpp (>= 1.0.5), Matrix, methods

Suggests knitr, rmarkdown, microbenchmark, tinytest

VignetteBuilder knitr

License GPL-3 | file LICENSE

Encoding UTF-8

LinkingTo Rcpp, RcppEigen

Config/roxygen2/version 8.1.0

NeedsCompilation yes

Repository CRAN

Date/Publication 2026-10-25 14:50:02 UTC

Contents

arnoldi	2
eigen2	3
factorize_lu	4
sanic	5
solve2	6
solve_cg	7
solve_chol	9
sparsify	10
svd2	11
Index	13

arnoldi	<i>Krylov Subspace Spectral Decomposition</i>
---------	---

Description

Arnoldi iteration and Lanczos method to iteratively approximate the Hessenberg or tridiagonal form of a matrix A and find its eigenvalues.

Usage

```
arnoldi(  
  a,  
  b,  
  symmetric,  
  iter = nrow(a),  
  tol = .Machine$double.eps,  
  eigen = TRUE,  
  orthogonalise = TRUE  
)  
  
lanczos(  
  a,  
  b,  
  iter = nrow(a),  
  tol = .Machine$double.eps,  
  eigen = TRUE,  
  orthogonalise = TRUE  
)
```

Arguments

- a Square numeric matrix.
- b Arbitrary numeric non-zero vector used to construct the basis.

<code>symmetric</code>	Logical scalar indicating whether 'a' is symmetric. By default symmetry is checked up to machine precision, which may take a long time for symmetric matrices.
<code>iter</code>	Integer scalar with the maximum number of iterations. Defaults to the theoretical maximum, i.e. the number of columns in 'a'.
<code>tol</code>	Numeric scalar with the desired tolerance. Defaults to the machine precision.
<code>eigen</code>	Logical scalar indicating whether to compute eigenvalues from the decomposition.
<code>orthogonalise</code>	Logical scalar indicating whether to use plain Lanczos or full reorthogonalisation. Defaults to reorthogonalisation.

Value

Returns a list with slots "H" for the Hessenberg form of 'a' or slots "diagonal" and "subdiagonal" for its triangular form, slot "Q" with the orthonormal basis, and, if requested, eigenvalues in the slot "values".

Examples

```
set.seed(42)
# Compute Hessenberg of a square matrix
A <- matrix(rnorm(9), nrow = 3, ncol = 3)
ks <- arnoldi(A, symmetric = FALSE)

# Compute tridiagonal of a symmetric matrix
A <- crossprod(matrix(rnorm(9), nrow = 3, ncol = 3))
ks <- lanczos(A)
ks <- arnoldi(A, symmetric = TRUE) # Short-hand
```

eigen2	<i>Spectral Decomposition</i>
--------	-------------------------------

Description

Solvers for eigenproblems around the matrix A . Compute eigenvalues λ and eigenvectors v of A , such that $Av = \lambda v$.

Usage

```
eigen2(a, symmetric, vectors = TRUE)
```

Arguments

<code>a</code>	Square numeric matrix.
<code>symmetric</code>	Logical scalar indicating whether 'a' is symmetric. By default symmetry is checked up to machine precision, which may take a long time for symmetric matrices.
<code>vectors</code>	Logical scalar indicating whether eigenvectors should be computed and returned.

Value

Solves the eigenproblem and returns a list with eigenvalues in the "values" slot and, if requested, eigenvectors in the slot "vectors".

Examples

```
set.seed(42)
# Compute eigenvalues and eigenvectors for a square matrix
A <- matrix(rnorm(9), nrow = 3, ncol = 3)
ev <- eigen2(A, symmetric = FALSE)

# Compute eigenvalues and eigenvectors for a symmetric matrix
A <- crossprod(matrix(rnorm(9), nrow = 3, ncol = 3))
ev <- eigen2(A, symmetric = TRUE)
# Check reconstruction
norm(A %% ev$vectors - ev$vectors %% diag(ev$values))
```

factorize_lu

Reusable LU Factorization

Description

Compute an LU factorization once and reuse it for different right-hand sides.

Usage

```
factorize_lu(a, pivot = 1L, ordering = 1L)

## S3 method for class 'sanic_lu'
solve(a, b, ...)
```

Arguments

a	For factorize_lu(), a nonempty square numeric matrix, dense or sparse (dgCMatrix). For solve(), a sanc_lu factorization.
pivot	Pivoting scheme for dense matrices, as in solve_lu .
ordering	Ordering scheme for sparse matrices, as in solve_lu .
b	Numeric vector or matrix of right-hand sides. If missing, compute the inverse. Sparse right-hand sides are converted to dense matrices.
...	Unused; additional arguments are rejected.

Details

The factorization owns its data: changing or removing `a` does not affect subsequent solves. Dense LU uses partial pivoting by default; `pivot = 2` uses full pivoting and checks for singularity. As with `solve_lu()`, partial pivoting assumes an invertible matrix and does not perform a rank check. Sparse LU ignores `pivot`.

Factorizations are session-local external pointers. They cannot be reused after serialization (for example, with `saveRDS()`) or in another R process; recreate them from the original matrix instead.

Value

`factorize_lu()` returns an object of class `sanic_lu`. Its `solve()` method returns a numeric matrix, including for vector right-hand sides. Omitting `b` computes the inverse.

Examples

```
A <- matrix(c(4, 1, 2, 3), 2)
f <- factorize_lu(A)
solve(f, c(1, 2))
solve(f, cbind(c(1, 2), c(3, 4)))
solve(f)
```

sanic

Solving $Ax = b$ Nimbly in C++

Description

Routines for solving large systems of linear equations in R. Direct and iterative solvers from the Eigen C++ library are made available. Solvers include Cholesky, LU, QR, and Krylov subspace methods (Conjugate Gradient, BiCGSTAB). Both dense and sparse problems are supported.

Author(s)

Maintainer: Nikolas Kuschnig <nikolas.kuschnig@wu.ac.at> ([ORCID](#))

Authors:

- Nikolas Kuschnig <nikolas.kuschnig@wu.ac.at> ([ORCID](#))

Other contributors:

- Lukas Vashold ([ORCID](#)) [contributor]
- Dirk Eddelbuettel ([ORCID](#)) [contributor]
- Yixuan Qiu [contributor]

See Also

Useful links:

- <https://github.com/nk027/sanic>
- Report bugs at <https://github.com/nk027/sanic/issues>

solve2

Solve Systems of Equations

Description

Solve systems of equations $Ax = b$ using an automatically chosen direct method (see [solve_chol](#)). Methods are chosen for speed at reasonable accuracy. Please choose a suitable method manually if numerical stability is the main consideration.

Usage

```
solve2(a, b, ...)

## Default S3 method:
solve2(a, b, ...)

## S3 method for class 'sanic_lu'
solve2(a, b, ...)
```

Arguments

a	Numeric matrix with the coefficients of the linear system, dense or sparse (see sparsify), or a sanic_lu factorization from factorize_lu .
b	Numeric vector or matrix at the right-hand side of the linear system. If missing, 'b' is set to an identity matrix and 'a' is inverted.
...	Dispatched to methods in the solvers.

Value

Solves for x and returns a numeric matrix with the results.

Examples

```
set.seed(42)
x <- rnorm(3)

# Solve using a general matrix
A <- matrix(rnorm(9), nrow = 3, ncol = 3)
b <- A %*% x
norm(solve2(A, b) - x)

# Solve using a symmetric matrix
A <- crossprod(matrix(rnorm(9), nrow = 3, ncol = 3))
b <- A %*% x
norm(solve2(A, b) - x)

# Solve using a square matrix
A <- matrix(rnorm(12), nrow = 4, ncol = 3)
```

```

b <- A %*% x
norm(solve2(A, b) - x)

# Solve reusing an LU factorization
A <- matrix(rnorm(9), nrow = 3, ncol = 3)
f <- factorize_lu(A)
norm(solve2(f, A %*% x) - x)

```

Description

Iterative solvers using the Conjugate Gradient method for sparse systems of equations $Ax = b$. Three different types are available: (1) stabilized bi-conjugate gradient (BiCGSTAB) for square matrices, (2) conjugate gradient for rectangular least-squares (LSCG), and (3) classic conjugate gradient (CG) for symmetric positive definite matrices.

Usage

```

solve_cg(
  a,
  b,
  x0,
  type = c("BiCGSTAB", "LSCG", "CG"),
  iter,
  tol,
  precondition = 1L,
  verbose = FALSE
)

```

Arguments

<code>a</code>	Square numeric matrix with the coefficients of the linear system. Dense and sparse matrices are supported, but the format must be sparse (see sparsify). Dense matrices are coerced automatically.
<code>b</code>	Numeric vector or matrix at the right-hand side of the linear system. If missing, 'b' is set to an identity matrix and 'a' is inverted.
<code>x0</code>	Numeric vector or matrix with an initial guess. Must be of the same dimension as 'b'.
<code>type</code>	Character scalar. Whether to use the BiCGSTAB, least squares CG or classic CG method.
<code>iter</code>	Integer scalar with the maximum number of iterations. Defaults to the theoretical maximum, i.e. the number of columns in 'a'.
<code>tol</code>	Numeric scalar with the desired tolerance. Defaults to the machine precision.

precond	Integer scalar indicating the type of preconditioner to be used. Defaults to diagonal preconditioning. See the Details for further information.
verbose	Logical scalar. Whether to print iterations and tolerance.

Details

Preconditioners can be set to 0 for no / identity preconditioning, 1 (default) for Jacobi / diagonal preconditioning, or 2 for incomplete factorisation. Not all schemes are available for every type:

* type = "BiCGSTAB" The default is precondition = 1 for diagonal preconditioning. Set precondition = 0 for no preconditioning, or precondition = 2 for an incomplete LUT preconditioner. * type = "LSCG" The default is precondition = 1 for diagonal least squares preconditioning. Set precondition = 0 for no preconditioning. * type = "CG" The default is precondition = 1 for diagonal preconditioning. Set precondition = 0 for no preconditioning, or precondition = 2 for an incomplete Cholesky preconditioner.

Value

Solves for x and returns a numeric matrix with the results. The matrix has converged, iterations, and error attributes containing the convergence status, number of iterations, and Eigen's estimated error.

Examples

```
set.seed(42)
x <- rnorm(3)

# Solve via BiCGSTAB for square matrices
A <- matrix(rnorm(9), nrow = 3, ncol = 3)
b <- A %*% x
norm(solve_cg(A, b, type = "B") - x)

# Solve via LSCG for rectangular matrices
A <- matrix(rnorm(12), nrow = 4, ncol = 3)
b <- A %*% x
norm(solve_cg(A, b, type = "LS") - x)

# Solve via classic CG for symmetric matrices
A <- crossprod(matrix(rnorm(9), nrow = 3, ncol = 3))
b <- A %*% x
norm(solve_cg(A, b, type = "CG") - x)

# The input matrix A should always be in sparse format
A <- sparsify(crossprod(matrix(rnorm(9), nrow = 3, ncol = 3)))
# The right-hand side should be a dense matrix
b <- as.matrix(A %*% x)

# We can check the speed of convergence and quality directly
solve_cg(A, b, verbose = TRUE)
# And provide guesses as starting value
solve_cg(A, b, x0 = x, verbose = TRUE)
```

Description

Direct solvers using Cholesky, LU, or QR decompositions for systems of equations $Ax = b$. Dense or sparse methods are used depending on the format of the input matrix (see [sparsify](#)).

Usage

```
solve_chol(a, b, pivot = 1L, ordering = 0L)
```

```
solve_lu(a, b, pivot = 1L, ordering = 1L)
```

```
solve_qr(a, b, pivot = 1L, ordering = 1L)
```

Arguments

a	Square numeric matrix with the coefficients of the linear system. Both dense and sparse matrices are supported (see sparsify).
b	Numeric vector or matrix at the right-hand side of the linear system. If missing, 'b' is set to an identity matrix and 'a' is inverted.
pivot	Integer scalar indicating the pivoting scheme to be used. Defaults to partial pivoting. See the Details for further information.
ordering	Integer scalar indicating the ordering scheme to be used. See the Details for further information.

Details

Pivoting schemes for dense matrices can be set to 0 for no pivoting, 1 (default) for partial pivoting, or 2 for full pivoting. Not all schemes are available for every decomposition:

* `solve_chol()` The default is `pivot = 1` for the robust LDLT decomposition of A , such that $A = P'LDL^*P$. For the LDLT A needs to be positive or negative semidefinite. Set `pivot = 0` for the plain LLT decomposition of A , such that $A = LL^* = U^*U$. For the LLT A needs to be positive definite and preferably numerically stable. * `solve_lu()` The default is `pivot = 1` for the partial pivoting LU decomposition of A , such that $A = PLU$. For this scheme A needs to be invertible and preferably numerically stable. Set `pivot = 2` for the complete pivoting LU decomposition of A , such that $A = P^{-1}LUQ^{-1}$. This scheme is applicable to square matrices, rank-revealing, and stable. `solve_qr()` The default is `pivot = 1` for the column pivoting Householder QR decomposition of A , such that $AP = QR$. This scheme is generally applicable, rank-revealing, and stable. Set `pivot = 2` for the full pivoting Householder QR decomposition of A , such that $PAP' = QR$. This scheme is generally applicable, rank-revealing, and optimally stable. Set `pivot = 0` for an unpivoted Householder QR decomposition of A , such that $A = QR$. This scheme is generally applicable, but not as stable as pivoted variants.

Ordering schemes for sparse matrices can be set to 0 for approximate minimum degree (AMD) ordering, 1 for column approximate minimum degree (COLAMD) ordering, or 2 for natural ordering. Not all orderings are available for every decomposition:

* `solve_chol()` The default is `ordering = 0` for AMD ordering. Set `ordering = 2` for natural ordering. * `solve_lu()` The default is `ordering = 1` for COLAMD ordering. Set `ordering = 0` for AMD or `ordering = 2` for natural ordering. * `solve_qr()` The default is `ordering = 1` for COLAMD ordering. Set `ordering = 0` for AMD or `ordering = 2` for natural ordering.

Value

Solves for x and returns a numeric matrix with the results.

Examples

```
set.seed(42)
x <- rnorm(3)

# Solve via QR for general matrices
A <- matrix(rnorm(12), nrow = 4, ncol = 3)
b <- A %*% x
norm(solve_qr(A, b) - x)

# Solve via LU for square matrices
A <- matrix(rnorm(9), nrow = 3, ncol = 3)
b <- A %*% x
norm(solve_lu(A, b) - x)

# Solve via Cholesky for symmetric matrices
A <- crossprod(matrix(rnorm(9), nrow = 3, ncol = 3))
b <- A %*% x
norm(solve_chol(A, b) - x)

# Sparse methods are available for the 'dgCMatrix' class from Matrix
A <- crossprod(matrix(rnorm(9), nrow = 3, ncol = 3))
b <- A %*% x
norm(solve_qr(sparsify(A), b))
norm(solve_lu(sparsify(A), b))
norm(solve_chol(sparsify(A), b))
```

sparsify

Transform a Matrix to Be Sparse.

Description

Concise function to transform dense to sparse matrices of class `dgCMatrix` (see [sparseMatrix](#)).

Usage

```
sparsify(x)
```

Arguments

`x` Numeric matrix to transform to a sparse 'dgCMatrix'.

Value

Returns 'x' as dgCMatrix.

Examples

```
sparsify(matrix(rnorm(9L), 3L))
```

svd2

Singular Value Decomposition

Description

Solvers for generalized eigenproblems around the matrix A . Compute singular values Σ , left singular vectors U and right singular vectors V of A , such that $A = U\Sigma V^*$. Two different types are available: (1) bidiagonal divide and conquer strategy (BDC) SVD, and (2) two-sided Jacobi SVD for small matrices (<16) and high accuracy.

Usage

```
svd2(a, type = c("BDC", "Jacobi"), vectors = TRUE, thin = TRUE)
```

Arguments

`a` Numeric matrix.

`type` Character scalar. Whether to use BDC or Jacobi SVD.

`vectors` Logical scalar indicating whether singular vectors should be computed and returned.

`thin` Logical scalar indicating whether singular vectors should be returned in thinned or full format.

Value

Solves the generalised eigenproblem and returns a list with singular values in the "d" component and, if requested, singular vectors in the components "u" and "v".

Examples

```
set.seed(42)
# Compute singular values and vectors using BDC
A <- matrix(rnorm(9), nrow = 3, ncol = 3)
sv <- svd2(A)

# Compute singular values using Jacobi
A <- matrix(rnorm(9), nrow = 3, ncol = 3)
sv <- svd2(A, type = "J", vectors = FALSE)

# Compute singular values and full vectors using BDC
A <- matrix(rnorm(12), nrow = 4, ncol = 3)
sv <- svd2(A, type = "B", thin = FALSE)
A <- matrix(rnorm(12), nrow = 3, ncol = 4)
sv <- svd2(A, type = "B", thin = FALSE)
```

Index

arnoldi, [2](#)

eigen2, [3](#)

factorize_lu, [4](#), [6](#)

lanczos (arnoldi), [2](#)

sanic, [5](#)

sanic-package (sanic), [5](#)

solve.sanic_lu (factorize_lu), [4](#)

solve2, [6](#)

solve_cg, [7](#)

solve_chol, [6](#), [9](#)

solve_lu, [4](#)

solve_lu (solve_chol), [9](#)

solve_qr (solve_chol), [9](#)

sparseMatrix, [10](#)

sparsify, [6](#), [7](#), [9](#), [10](#)

svd2, [11](#)