Package ‘Rlinsolve’

August 21, 2021

Type Package

Title Iterative Solvers for (Sparse) Linear System of Equations

Version 0.3.2

Description Solving a system of linear equations is one of the most fundamental computational problems for many fields of mathematical studies, such as regression problems from statistics or numerical partial differential equations. We provide basic stationary iterative solvers such as Jacobi, Gauss-Seidel, Successive Over-Relaxation and SSOR methods. Nonstationary, also known as Krylov subspace methods are also provided. Sparse matrix computation is also supported in that solving large and sparse linear systems can be manageable using 'Matrix' package along with 'RcppArmadillo'. For a more detailed description, see a book by Saad (2003) <doi:10.1137/1.9780898718003>.

License MIT + file LICENSE

Encoding UTF-8

Imports Rcpp (>= 0.12.4), Matrix, Rdpack, stats, utils

LinkingTo Rcpp, RcppArmadillo

RoxygenNote 7.1.1

RdMacros Rdpack

NeedsCompilation yes

Author Kisung You [aut, cre] (<https://orcid.org/0000-0002-8584-459X>)

Maintainer Kisung You <kisungyou@outlook.com>

Repository CRAN

Date/Publication 2021-08-21 15:40:10 UTC

R topics documented:

aux.fisch ................................................................. 2
lsolve.bicg ............................................................ 3
lsolve.bicgstab ........................................................ 4
lsolve.cg ............................................................... 6
lsolve.cgs .............................................................. 7
Generate a 2-dimensional discrete Poisson matrix

Description

Poisson equation is one of most well-known elliptic partial differential equations. In order to give a concrete example, a discrete Poisson matrix is generated, assuming we have \( N \) number of grid points for each dimension under square domain. \( \text{fisch} \) is a German word for Poisson.

Usage

```
aux.fisch(N, sparse = FALSE)
```

Arguments

- \( N \) the number of grid points for each direction.
- `sparse` a logical; TRUE for returning sparse matrix, FALSE otherwise.

Value

an \( (N^2 \times N^2) \) matrix having block banded structure.

References


Examples

```r
## generate dense and sparse Poisson matrix of size 25 by 25.
A = aux.fisch(5, sparse=FALSE)
B = aux.fisch(5, sparse=TRUE)
(all(A==B)) # TRUE if two matrices are equal.
```
**lsolve.bicg**

*Description*

Biconjugate Gradient (BiCG) method is a modification of Conjugate Gradient for nonsymmetric systems using evaluations with respect to $A^T$ as well as $A$ in matrix-vector multiplications. For an overdetermined system where $\text{nr}(A) > \text{nc}(A)$, it is automatically transformed to the normal equation. Underdetermined system - $\text{nr}(A) < \text{nc}(A)$ - is not supported. Preconditioning matrix $M$, in theory, should be symmetric and positive definite with fast computability for inverse, though it is not limited until the solver level.

**Usage**

```r
lsolve.bicg(
  A,
  B,
  xinit = NA,
  reltol = 1e-05,
  maxiter = 10000,
  preconditioner = diag(ncol(A)),
  verbose = TRUE
)
```

**Arguments**

- **A**  
  an $(m \times n)$ dense or sparse matrix. See also `sparseMatrix`.
- **B**  
  a vector of length $m$ or an $(m \times k)$ matrix (dense or sparse) for solving $k$ systems simultaneously.
- **xinit**  
  a length-$n$ vector for initial starting point. NA to start from a random initial point near 0.
- **reltol**  
  tolerance level for stopping iterations.
- **maxiter**  
  maximum number of iterations allowed.
- **preconditioner**  
  an $(n \times n)$ preconditioning matrix; default is an identity matrix.
- **verbose**  
  a logical; TRUE to show progress of computation.

**Value**

a named list containing

- **x**  
  solution; a vector of length $n$ or a matrix of size $(n \times k)$.
- **iter**  
  the number of iterations required.
- **errors**  
  a vector of errors for stopping criterion.
References


Examples

```r
## Overdetermined System
set.seed(100)
A = matrix(rnorm(10*5),nrow=10)
x = rnorm(5)
b = A%*%x

out1 = lsolve.cg(A,b)
out2 = lsolve.bicg(A,b)
matout = cbind(matrix(x),out1$x, out2$x);
colnames(matout) = c("true x","CG result", "BiCG result")
print(matout)
```

lsolve.bicgstab

*Biconjugate Gradient Stabilized Method*

Description

Biconjugate Gradient Stabilized(BiCGSTAB) method is a stabilized version of Biconjugate Gradient method for nonsymmetric systems using evaluations with respect to $A^T$ as well as $A$ in matrix-vector multiplications. For an overdetermined system where $\text{nrow}(A) > \text{ncol}(A)$, it is automatically transformed to the normal equation. Underdetermined system - $\text{nrow}(A) < \text{ncol}(A)$ - is not supported. Preconditioning matrix $M$, in theory, should be symmetric and positive definite with fast computability for inverse, though it is not limited until the solver level.

Usage

```r
lsolve.bicgstab(
  A,
  B,
  xinit = NA,
  reltol = 1e-05,
  maxiter = 1000,
  preconditioner = diag(ncol(A)),
  verbose = TRUE
)
```
Arguments

- **A**: an \( (m \times n) \) dense or sparse matrix. See also `sparseMatrix`.
- **B**: a vector of length \( m \) or an \( (m \times k) \) matrix (dense or sparse) for solving \( k \) systems simultaneously.
- **xinit**: a length-\( n \) vector for initial starting point. `NA` to start from a random initial point near 0.
- **reltol**: tolerance level for stopping iterations.
- **maxiter**: maximum number of iterations allowed.
- **preconditioner**: an \( (n \times n) \) preconditioning matrix; default is an identity matrix.
- **verbose**: a logical; `TRUE` to show progress of computation.

Value

A named list containing

- **x**: solution; a vector of length \( n \) or a matrix of size \( (n \times k) \).
- **iter**: the number of iterations required.
- **errors**: a vector of errors for stopping criterion.

References


Examples

```r
## Overdetermined System
set.seed(100)
A = matrix(rnorm(10*5),nrow=10)
x = rnorm(5)
b = A%x

out1 = lsolve.cg(A,b)
out2 = lsolve.bicg(A,b)
out3 = lsolve.bicgstab(A,b)
matout = cbind(matrix(x),out1$x, out2$x, out3$x);
colnames(matout) = c("true x","CG result", "BiCG result", "BiCGSTAB result")
print(matout)
```
lsolve.cg

Description

Conjugate Gradient(CG) method is an iterative algorithm for solving a system of linear equations where the system is symmetric and positive definite. For a square matrix $A$, it is required to be symmetric and positive definite. For an overdetermined system where $\text{nr}(A) > \text{nc}(A)$, it is automatically transformed to the normal equation. Underdetermined system - $\text{nr}(A) < \text{nc}(A)$ - is not supported. Preconditioning matrix $M$, in theory, should be symmetric and positive definite with fast computability for inverse, though it is not limited until the solver level.

Usage

```r
lsolve.cg(
  A,
  B,
  xinit = NA,
  reltol = 1e-05,
  maxiter = 10000,
  preconditioner = diag(ncol(A)),
  adjsym = TRUE,
  verbose = TRUE
)
```

Arguments

- **A**: an $(m \times n)$ dense or sparse matrix. See also `sparseMatrix`.
- **B**: a vector of length $m$ or an $(m \times k)$ matrix (dense or sparse) for solving $k$ systems simultaneously.
- **xinit**: a length-$n$ vector for initial starting point. NA to start from a random initial point near 0.
- **reltol**: tolerance level for stopping iterations.
- **maxiter**: maximum number of iterations allowed.
- **preconditioner**: an $(n \times n)$ preconditioning matrix; default is an identity matrix.
- **adjsym**: a logical; TRUE to symmetrize the system by transforming the system into normal equation, FALSE otherwise.
- **verbose**: a logical; TRUE to show progress of computation.

Value

- a named list containing
  - **x**: solution; a vector of length $n$ or a matrix of size $(n \times k)$.
  - **iter**: the number of iterations required.
  - **errors**: a vector of errors for stopping criterion.
lsolve.cgs

References


Examples

```r
## Overdetermined System
set.seed(100)
A = matrix(rnorm(10*5),nrow=10)
x = rnorm(5)
b = A%*%x

out1 = lsolve.sor(A,b,w=0.5)
out2 = lsolve.cg(A,b)
matout = cbind(matrix(x),out1$x, out2$x);
colnames(matout) = c("true x","SSOR result", "CG result")
print(matout)
```

---

lsolve.cgs  
Conjugate Gradient Squared method

Description

Conjugate Gradient Squared (CGS) method is an extension of Conjugate Gradient method where the system is symmetric and positive definite. It aims at achieving faster convergence using an idea of contraction operator twice. For a square matrix $A$, it is required to be symmetric and positive definite. For an overdetermined system where $\text{nr} \text{ow}(A) > \text{ncol}(A)$, it is automatically transformed to the normal equation. Underdetermined system - $\text{nr} \text{ow}(A) < \text{ncol}(A)$ - is not supported. Preconditioning matrix $M$, in theory, should be symmetric and positive definite with fast computability for inverse, though it is not limited until the solver level.

Usage

```r
lsolve.cgs(
  A,
  B,
  xinit = NA,
  reltol = 1e-05,
  maxiter = 10000,
  preconditioner = diag(ncol(A)),
  adjsym = TRUE,
  verbose = TRUE
)
```
Arguments

A  an \((m \times n)\) dense or sparse matrix. See also \texttt{sparseMatrix}.
B  a vector of length \(m\) or an \((m \times k)\) matrix (dense or sparse) for solving \(k\) systems simultaneously.
\texttt{xinit}  a length-\(n\) vector for initial starting point. \texttt{NA} to start from a random initial point near 0.
\texttt{reltol}  tolerance level for stopping iterations.
\texttt{maxiter}  maximum number of iterations allowed.
\texttt{preconditioner}  an \((n \times n)\) preconditioning matrix; default is an identity matrix.
\texttt{adjsym}  a logical; \texttt{TRUE} to symmetrize the system by transforming the system into normal equation, \texttt{FALSE} otherwise.
\texttt{verbose}  a logical; \texttt{TRUE} to show progress of computation.

Value

a named list containing

\textbf{x}  solution; a vector of length \(n\) or a matrix of size \((n \times k)\).
\textbf{iter}  the number of iterations required.
\textbf{errors}  a vector of errors for stopping criterion.

References


Examples

```r
## Overdetermined System
set.seed(100)
A = matrix(rnorm(10*5),nrow=10)
x = rnorm(5)
b = A%*%x
out1 = lsolve.cg(A,b)
out2 = lsolve.cgs(A,b)
matout = cbind(matrix(x),out1$x, out2$x);
rownames(matout) = c("true x","CG result", "CGS result")
print(matout)
```
Description

Chebyshev method - also known as Chebyshev iteration - avoids computation of inner product, enabling distributed-memory computation to be more efficient at the cost of requiring a priori knowledge on the range of spectrum for matrix $A$.

Usage

```r
lsolve.cheby(
  A, B,
  xinit = NA, reltol = 1e-05, maxiter = 10000,
  preconditioner = diag(ncol(A)),
  adjsym = TRUE,
  verbose = TRUE
)
```

Arguments

- **A**: an $(m \times n)$ dense or sparse matrix. See also `sparseMatrix`.
- **B**: a vector of length $m$ or an $(m \times k)$ matrix (dense or sparse) for solving $k$ systems simultaneously.
- **xinit**: a length-$n$ vector for initial starting point. NA to start from a random initial point near 0.
- **reltol**: tolerance level for stopping iterations.
- **maxiter**: maximum number of iterations allowed.
- **preconditioner**: an $(n \times n)$ preconditioning matrix; default is an identity matrix.
- **adjsym**: a logical; TRUE to symmetrize the system by transforming the system into normal equation, FALSE otherwise.
- **verbose**: a logical; TRUE to show progress of computation.

Value

A named list containing:

- **x**: solution; a vector of length $n$ or a matrix of size $(n \times k)$.
- **iter**: the number of iterations required.
- **errors**: a vector of errors for stopping criterion.
References


Examples

```r
## Overdetermined System
set.seed(100)
A = matrix(rnorm(10*5), nrow=10)
x = rnorm(5)
b = A %*% x

out1 = lsolve.sor(A, b, w=0.5)
out2 = lsolve.cheby(A, b)
matout = cbind(x, out1$x, out2$x);
colnames(matout) = c("original x", "SOR result", "Chebyshev result")
print(matout)
```

lsolve.gmres

**Generalized Minimal Residual method**

Description

GMRES is a generic iterative solver for a nonsymmetric system of linear equations. As its name suggests, it approximates the solution using Krylov vectors with minimal residuals.

Usage

```r
lsolve.gmres(
  A,
  B,
  xinit = NA,
  reltol = 1e-05,
  maxiter = 1000,
  preconditioner = diag(ncol(A)),
  restart = (ncol(A) - 1),
  verbose = TRUE
)
```

Arguments

- **A**: an \((m \times n)\) dense or sparse matrix. See also \sparseMatrix{.}
- **B**: a vector of length \(m\) or an \((m \times k)\) matrix (dense or sparse) for solving \(k\) systems simultaneously.
- **xinit**: a length-\(n\) vector for initial starting point. NA to start from a random initial point near 0.
**solve.gs**

- **reltol**: tolerance level for stopping iterations.
- **maxiter**: maximum number of iterations allowed.
- **preconditioner**: an \((n \times n)\) preconditioning matrix; default is an identity matrix.
- **restart**: the number of iterations before restart.
- **verbose**: a logical; `TRUE` to show progress of computation.

**Value**

A named list containing:

- **x**: solution; a vector of length \(n\) or a matrix of size \((n \times k)\).
- **iter**: the number of iterations required.
- **errors**: a vector of errors for stopping criterion.

**References**


**Examples**

```r
## Overdetermined System
set.seed(100)
A = matrix(rnorm(10*5), nrow=10)
x = rnorm(5)
b = A %*% x

out1 = lsolve.cg(A, b)
out3_1 = lsolve.gmres(A, b, restart=2)
out3_2 = lsolve.gmres(A, b, restart=3)
out3_3 = lsolve.gmres(A, b, restart=4)
matout = cbind(matrix(x), out1$x, out3_1$x, out3_2$x, out3_3$x);
colnames(matout) = c("true x", "CG", "GMRES(2)", "GMRES(3)", "GMRES(4)"")
print(matout)
```

---

**lsolve.gs**

*Gauss-Seidel method*

**Description**

Gauss-Seidel(GS) method is an iterative algorithm for solving a system of linear equations, with a decomposition \(A = D + L + U\) where \(D\) is a diagonal matrix and \(L\) and \(U\) are strictly lower/upper triangular matrix respectively. For a square matrix \(A\), it is required to be diagonally dominant or symmetric and positive definite. For an overdetermined system where \(\text{nrow}(A) > \text{ncol}(A)\), it is automatically transformed to the normal equation. Underdetermined system - \(\text{nrow}(A) < \text{ncol}(A)\) - is not supported.
Usage

lsolve.gs(
    A,
    B,
    xinit = NA,
    reltol = 1e-05,
    maxiter = 1000,
    adjsym = TRUE,
    verbose = TRUE
)

Arguments

A an \((m \times n)\) dense or sparse matrix. See also \texttt{sparsMatrix}.
B a vector of length \(m\) or an \((m \times k)\) matrix (dense or sparse) for solving \(k\) systems simultaneously.

xinit a length-\(n\) vector for initial starting point. \texttt{NA} to start from a random initial point near 0.

reltol tolerance level for stopping iterations.
maxiter maximum number of iterations allowed.
adjsym a logical; \texttt{TRUE} to symmetrize the system by transforming the system into normal equation, \texttt{FALSE} otherwise.

verbose a logical; \texttt{TRUE} to show progress of computation.

Value

a named list containing

\(x\) solution; a vector of length \(n\) or a matrix of size \((n \times k)\).
iter the number of iterations required.
errors a vector of errors for stopping criterion.

References


Examples

```r
## Overdetermined System
set.seed(100)
A = matrix(rnorm(10*5),nrow=10)
x = rnorm(5)
b = A%*%x

out = lsolve.gs(A,b)
matout = cbind(matrix(x),out$x); colnames(matout) = c("true x","est from GS")
print(matout)
```
Description

Jacobi method is an iterative algorithm for solving a system of linear equations, with a decomposition $A = D + R$ where $D$ is a diagonal matrix. For a square matrix $A$, it is required to be diagonally dominant. For an overdetermined system where $\text{nrow}(A) > \text{ncol}(A)$, it is automatically transformed to the normal equation. Underdetermined system - $\text{nrow}(A) < \text{ncol}(A)$ - is not supported.

Usage

```r
lsolve.jacobi(
  A,
  B,
  xinit = NA,
  reltol = 1e-05,
  maxiter = 1000,
  weight = 2/3,
  adjsym = TRUE,
  verbose = TRUE
)
```

Arguments

- **A**: an $(m \times n)$ dense or sparse matrix. See also `sparseMatrix`.
- **B**: a vector of length $m$ or an $(m \times k)$ matrix (dense or sparse) for solving $k$ systems simultaneously.
- **xinit**: a length-$n$ vector for initial starting point. NA to start from a random initial point near 0.
- **reltol**: tolerance level for stopping iterations.
- **maxiter**: maximum number of iterations allowed.
- **weight**: a real number in $(0, 1]$; 1 for native Jacobi.
- **adjsym**: a logical; TRUE to symmetrize the system by transforming the system into normal equation, FALSE otherwise.
- **verbose**: a logical; TRUE to show progress of computation.

Value

a named list containing

- **x**: solution; a vector of length $n$ or a matrix of size $(n \times k)$.
- **iter**: the number of iterations required.
- **errors**: a vector of errors for stopping criterion.
References


Examples

```r
## Overdetermined System
set.seed(100)
A = matrix(rnorm(10*5),nrow=10)
x = rnorm(5)
b = A %*% x

out1 = lsolve.jacobi(A,b,weight=1,verbose=FALSE) # unweighted
out2 = lsolve.jacobi(A,b,verbose=FALSE)       # weight of 0.66
out3 = lsolve.jacobi(A,b,weight=0.5,verbose=FALSE) # weight of 0.50
print("* lsolve.jacobi : overdetermined case example")
print(paste("* error for unweighted Jacobi case : ",norm(out1$x-x)))
print(paste("* error for 0.66 weighted Jacobi case : ",norm(out2$x-x)))
print(paste("* error for 0.50 weighted Jacobi case : ",norm(out3$x-x)))
```

lsolve.qmr

Quasi Minimal Residual Method

Description

Quasi-Minimal Residual(QMR) method is another remedy of the BiCG which shows rather irregular convergence behavior. It adapts to solve the reduced tridiagonal system in a least squares sense and its convergence is known to be quite smoother than BiCG.

Usage

```r
lsolve.qmr(
  A,
  B,
  xinit = NA,
  reltol = 1e-05,
  maxiter = 1000,
  preconditioner = diag(ncol(A)),
  verbose = TRUE
)
```

Arguments

- **A**: an \((m \times n)\) dense or sparse matrix. See also `sparseMatrix`
- **B**: a vector of length \(m\) or an \((m \times k)\) matrix (dense or sparse) for solving \(k\) systems simultaneously.
lsolve.sor

Successive Over-Relaxation method

Description

Successive Over-Relaxation (SOR) method is a variant of Gauss-Seidel method for solving a system of linear equations, with a decomposition $A = D + L + U$ where $D$ is a diagonal matrix and $L$ and $U$ are strictly lower/upper triangular matrix respectively. For a square matrix $A$, it is required to be diagonally dominant or symmetric and positive definite like GS method. For an overdetermined system where $\text{nrow}(A) > \text{ncol}(A)$, it is automatically transformed to the normal equation. Underdetermined system - $\text{nrow}(A) < \text{ncol}(A)$ - is not supported.
Usage

lsolve.sor(
    A,
    B,
    xinit = NA,
    reltol = 1e-05,
    maxiter = 1000,
    w = 1,
    adjsym = TRUE,
    verbose = TRUE
)

Arguments

A an \((m \times n)\) dense or sparse matrix. See also \code{sparseMatrix}.

B a vector of length \(m\) or an \((m \times k)\) matrix (dense or sparse) for solving \(k\) systems simultaneously.

xinit a length-\(n\) vector for initial starting point. \code{NA} to start from a random initial point near 0.

reltol tolerance level for stopping iterations.

maxiter maximum number of iterations allowed.

w a weight value in \((0, 2)\); \(w=1\) leads to Gauss-Seidel method.

adjsym a logical; \code{TRUE} to symmetrize the system by transforming the system into normal equation, \code{FALSE} otherwise.

verbose a logical; \code{TRUE} to show progress of computation.

Value

a named list containing

\code{x} solution; a vector of length \(n\) or a matrix of size \((n \times k)\).

\code{iter} the number of iterations required.

\code{errors} a vector of errors for stopping criterion.

References


Examples

## Overdetermined System
set.seed(100)
A = matrix(rnorm(10*5),nrow=10)
x = rnorm(5)
b = A%*%x
out1 = lsolve.ssor(A,b)
out2 = lsolve.ssor(A,b,w=0.5)
out3 = lsolve.ssor(A,b,w=1.5)
matout = cbind(matrix(x),out1$x, out2$x, out3$x);
colnames(matout) = c("true x","SOR 1 = GS", "SOR w=0.5", "SOR w=1.5")
print(matout)

lsolve.ssor  
Symmetric Successive Over-Relaxation method

Description
Symmetric Successive Over-Relaxation (SSOR) method is a variant of Gauss-Seidel method for solving a system of linear equations, with a decomposition \( A = D + L + U \) where \( D \) is a diagonal matrix and \( L \) and \( U \) are strictly lower/upper triangular matrix respectively. For a square matrix \( A \), it is required to be diagonally dominant or symmetric and positive definite like GS method. For an overdetermined system where \( \text{nrow}(A) > \text{ncol}(A) \), it is automatically transformed to the normal equation. Underdetermined system - \( \text{nrow}(A) < \text{ncol}(A) \) - is not supported.

Usage
lsolve.ssor(
    A,  
    B,  
    xinit = NA,  
    reltol = 1e-05,  
    maxiter = 1000,  
    w = 1,  
    adjsym = TRUE,  
    verbose = TRUE
)

Arguments
A  an \((m \times n)\) dense or sparse matrix. See also \texttt{sparseMatrix}.  
B  a vector of length \( m \) or an \((m \times k)\) matrix (dense or sparse) for solving \( k \) systems simultaneously.  
xinit  a length-\(n\) vector for initial starting point. \texttt{NA} to start from a random initial point near 0.  
reltol  tolerance level for stopping iterations.  
maxiter  maximum number of iterations allowed.  
w  a weight value in \((0, 2)\); \( w = 1 \) leads to Gauss-Seidel method.  
adjsym  a logical; \texttt{TRUE} to symmetrize the system by transforming the system into normal equation, \texttt{FALSE} otherwise.  
verbose  a logical; \texttt{TRUE} to show progress of computation.
Value

- a named list containing
  - \( \mathbf{x} \) solution; a vector of length \( n \) or a matrix of size \( (n \times k) \).
  - \( \text{iter} \) the number of iterations required.
  - \( \text{errors} \) a vector of errors for stopping criterion.

References


Examples

```r
## Overdetermined System
set.seed(100)
A = matrix(rnorm(10*5),nrow=10)
x = rnorm(5)
b = A%*%x

out1 = lsolve.ssor(A,b)
out2 = lsolve.ssor(A,b,w=0.5)
out3 = lsolve.ssor(A,b,w=1.5)
matout = cbind(matrix(x),out1$x, out2$x, out3$x);
colnames(matout) = c("true x","SSOR w=1", "SSOR w=0.5", "SSOR w=1.5")
print(matout)
```

Rlinsolve

* A Collection of Iterative Solvers for (Sparse) Linear System of Equations

Description

Solving a system of linear equations is one of the most fundamental computational problems for many fields of mathematical studies, such as regression from statistics or numerical partial differential equations. We provide a list of both stationary and nonstationary solvers. Sparse matrix class from *Matrix* is also supported for large sparse system.

Non-square matrix

For a matrix \( A \) of size \( (m \times n) \), we say the system is **overdetermined** if \( m > n \), **underdetermined** if \( m < n \), or **squared** if \( m = n \). In the current version, underdetermined system is not supported; it will later appear with sparse constraints. For an overdetermined system, it automatically transforms the problem into **normal equation**, i.e.,

\[
Ax = b \rightarrow A^T Ax = A^T b
\]

even though if suffers from worse condition number while having desirable property of a system to be symmetric and positive definite.
**Sparsity**

*RcppArmadillo* is extensively used in the package. In order for bullet-proof transition between dense and sparse matrix, only 3 of 12 RcppArmadillo-supported sparse matrix formats have access to our algorithms; "dgCMatrix", "dtCMatrix" and "dsCMatrix". Please see the vignette on sparse matrix support from RcppArmadillo. If either of two inputs A or b is sparse, all matrices involved are automatically transformed into sparse matrices.

**Composition of the Package**

Following is a list of stationary methods,

- `lsolve.jacobi` Jacobi method
- `lsolve.gs` Gauss-Seidel method
- `lsolve.sor` Successive Over-Relaxation method
- `lsolve.ssor` Symmetric Successive Over-Relaxation method

as well as nonstationary (or, Krylov subspace) methods,

- `lsolve.bicg` Bi-Conjugate Gradient method
- `lsolve.bicgstab` Bi-Conjugate Gradient Stabilized method
- `lsolve.cg` Conjugate Gradient method
- `lsolve.cgs` Conjugate Gradient Squared method
- `lsolve.cheby` Chebyshev method
- `lsolve.gmres` Generalized Minimal Residual method
- `lsolve.qmr` Quasi-Minimal Residual method

Also, `aux.fisch` is provided to generate a sparse system of discrete Poisson matrix from finite difference approximation scheme of Poisson equation on 2-dimensional square domain.

**References**


Index

aux.fisch, 2, 19
lsolve.bicg, 3, 19
lsolve.bicgstab, 4, 19
lsolve.cg, 6, 19
lsolve.cgs, 7, 19
lsolve.cheby, 9, 19
lsolve.gmres, 10, 19
lsolve.gs, 11, 19
lsolve.jacobi, 13, 19
lsolve.qmr, 14, 19
lsolve.sor, 15, 19
lsolve.ssor, 17, 19
Rlinsolve, 18

sparseMatrix, 3, 5, 6, 8–10, 12–14, 16, 17