Package 'Rlinsolve'

September 22, 2025

Type Package

Title Iterative Solvers for (Sparse) Linear System of Equations

Version 0.3.3

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 7.3.2

RdMacros Rdpack

NeedsCompilation yes

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

Maintainer Kisung You <kisung.you@outlook.com>

Repository CRAN

Date/Publication 2025-09-22 05:10:56 UTC

Contents

`aux.fisch` *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. *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

Golub, G. H. and Van Loan, C. F. (1996) *Matrix Computations, 3rd Ed.*, pages 177–180.

Examples

```
## 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` *Biconjugate Gradient method*

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 `nrow(A)>ncol(A)`, it is automatically transformed to the normal equation. Underdetermined system - `nrow(A)<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

```
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

Fletcher R (1976). "Conjugate gradient methods for indefinite systems." In Watson GA (ed.), *Numerical Analysis*, volume 506, 73–89. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-540-07610-0 978-3-540-38129-7.

Voevodin VV (1983). "The question of non-self-adjoint extension of the conjugate gradients method is closed." *USSR Computational Mathematics and Mathematical Physics*, **23**(2), 143–144. ISSN 00415553.

Examples

```
## 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 `nrow(A)>ncol(A)`, it is automatically transformed to the normal equation. Underdetermined system - `nrow(A)<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

```
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

van der Vorst HA (1992). "Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems." *SIAM Journal on Scientific and Statistical Computing*, **13**(2), 631–644. ISSN 0196-5204, 2168-3417.

Examples

```
## 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` *Conjugate Gradient method*

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 `nrow(A)>ncol(A)`, it is automatically transformed to the normal equation. Underdetermined system - `nrow(A)<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

```
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.

References

Hestenes MR, Stiefel E (1952). "Methods of conjugate gradients for solving linear systems." *Journal of Research of the National Bureau of Standards*, **49**(6), 409. ISSN 0091-0635.

Examples

```
## 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 `nrow(A)>ncol(A)`, it is automatically transformed to the normal equation. Underdetermined system - `nrow(A)<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

```
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 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

Sonneveld P (1989). "CGS, A Fast Lanczos-Type Solver for Nonsymmetric Linear systems." *SIAM Journal on Scientific and Statistical Computing*, **10**(1), 36–52. ISSN 0196-5204, 2168-3417.

Examples

```
## 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);
colnames(matout) = c("true x","CG result", "CGS result")
print(matout)
```

`lsolve.cheby` *Chebyshev Method*

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

```
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

Gutknecht MH, Röllin S (2002). "The Chebyshev iteration revisited." *Parallel Computing*, **28**(2), 263–283. ISSN 01678191.

Examples

```
## 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

```
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.

`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

Saad Y, Schultz MH (1986). "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems." *SIAM Journal on Scientific and Statistical Computing*, **7**(3), 856–869. ISSN 0196-5204, 2168-3417.

Examples

```
## 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 `nrow(A)>ncol(A)`, it is automatically transformed to the normal equation. Underdetermined system - `nrow(A)<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 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.
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

Demmel JW (1997). *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics. ISBN 978-0-89871-389-3 978-1-61197-144-6.

Examples

```
## 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)
```

`lsolve.jacobi` *Jacobi method*

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 `nrow(A)>ncol(A)`, it is automatically transformed to the normal equation. Underdetermined system - `nrow(A)<ncol(A)` - is not supported.

Usage

```
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

Demmel JW (1997). *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics. ISBN 978-0-89871-389-3 978-1-61197-144-6.

Examples

```
## 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

Quasia-Minimal Resudial(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

```
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.

`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

Freund RW, Nachtigal NM (1991). "QMR: a quasi-minimal residual method for non-Hermitian linear systems." *Numerische Mathematik*, **60**(1), 315–339. ISSN 0029-599X, 0945-3245.

Examples

```
## 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.qmr(A,b)
matout = cbind(matrix(x),out1$x, out2$x, out3$x);
colnames(matout) = c("true x","CG result", "BiCG result", "QMR result")
print(matout)
```

---

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 `nrow(A)>ncol(A)`, it is automatically transformed to the normal equation. Underdetermined system - `nrow(A)<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 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.
w	a weight value in $(0, 2)$.; w=1 leads to Gauss-Seidel method.
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

Demmel JW (1997). *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics. ISBN 978-0-89871-389-3 978-1-61197-144-6.

Examples

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

```
out1 = lsolve.sor(A,b)
out2 = lsolve.sor(A,b,w=0.5)
out3 = lsolve.sor(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 `nrow(A)>ncol(A)`, it is automatically transformed to the normal equation. Underdetermined system - `nrow(A)<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 `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.
`w`	a weight value in $(0, 2)$.; `w=1` leads to Gauss-Seidel method.
`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

Demmel JW (1997). *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics. ISBN 978-0-89871-389-3 978-1-61197-144-6.

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","SSOR w=1", "SSOR w=0.5", "SSOR w=1.5")
print(matout)
```

Index