2 Solution of linear algebraic equations

Functions:

gaussj: matrix inversion/linear equation solution using Gauss-Jordan
ludcmp: linear equation solution, LU decomposition
lubksb: linear equation solution, backsubstitution
tridag: linear equation solution, tridiagonal equations
mprove: linear equation solution, iterative improvement
vander: linear equation solution, Vandermonde matrices
toeplz: linear equation solution, Toeplitz matrices
svdcmp: singular value decomposition of a matrix
svbksb: singular value backsubstitution
sparse: linear equation solution, sparse matrix

---

gaussj

Description:

This function performs matrix inversion and the solution to a system of n by n linear equations $A.x = b$ using the Gauss-Jordan algorithm.

gaussj(a, b) $\longrightarrow$ **[a, b]**

Where:

a on input is a real square matrix representing the left hand sides A of the linear equations to be solved and on output the inverse matrix,
b on input is a real matrix representing the right hand sides and on output the matrix of solutions.

Example:

The equation `a*x = b` is solved using `gaussj` and the Senac function `matrix`. The return list is stored in `res`. The result is then checked.

```
-> a : matrix([1.0, 0.0, 0.0],
              [0.0, 2.0, 0.0],
              [0.0, 0.0, 3.0]);
-> Right-hand side vectors: #
-> b : matrix([1.0, 1.0],
              [0.0, 1.0],
              [0.0, 1.0]);
-> Inverse of matrix a and the solution to the system: #
-> res : gaussj(a, b)?
 | 1.000000000  0.000000000  0.000000000 |  | 1.000000000  1.000000000 |
 |                                       |  |                          |
[| 0.000000000  0.500000000  0.000000000 |, | 0.000000000  0.500000000 |]
 |                                       |  |                          |
 | 0.000000000  0.000000000  0.333333333 |  | 0.000000000  0.333333333 |

 -> a*first(res)?
 | 1.000000000  0.000000000  0.000000000 |
 |                                       |
 | 0.000000000  1.000000000  0.000000000 |
 |                                       |
 | 0.000000000  0.000000000  1.000000000 |
 -> a*last(res)?
 | 1.000000000  1.000000000 |
 |                          |
 | 0.000000000  1.000000000 |
 |                          |
 | 0.000000000  1.000000000 |
```

ludcmp

Description:

This function computes the LU decomposition of a square matrix.

ludcmp(a) $\longrightarrow$ [a, indx, d]

Where:

a on input is a real square matrix and on output the LU decomposition,
indx is a list of integers recording the row permutation effected by partial pivoting,
d is a list of values being either +1 or -1 depending on whether the corresponding permutation was even or odd respectively.

Pre-variables:

ludcmp_tiny [1.0d-20] a small floating point number.

Example:

For an example of the use of `ludcmp` see the example following `lubksb` below.

---

lubksb

Description:

This function performs the backsubstitution step in the solution to a system of linear equations $A.x = b$.

lubksb(a, indx, b) $\longrightarrow$ b

Where:

a is a real square matrix being the LU decomposition of the original matrix A returned by *ludcmp*,
indx is an integer permution vector returned by *ludcmp*,
b on input is a real list representing the right-hand side vector and on output the solution vector.

Example:

In this example first a matrix `a` is decomposed using `ludcmp`. Backsubstitution is then performed to solve the problem `a*x = b`, where `b` is a matrix.

```
-> First perform LU decomp.: #
-> a : matrix([1.0, 2.0, 3.0],
               [2.0, 2.0, 3.0],
               [3.0, 3.0, 3.0]);
-> Right-hand side vectors: #
-> b : matrix([1.0, 1.0],
               [1.0, 2.0],
               [1.0, 3.0]);
-> res : ludcmp(a)?
 | 3.000000000  3.000000000  3.000000000 |
 |                                       |
[| 0.333333333  1.000000000  2.000000000 |, [2, 2, 2], 1.000000000]
 |                                       |
 | 0.666666667  0.000000000  1.000000000 |
-> a : first(res);
-> indx : proj(res,2);
-> res1 : lubksb(a,indx,column(1,b))?
[0.000000000, 0.000000000, 0.333333333]
-> res2 : lubksb(a,indx,column(2,b))?
[1.000000000, 0.000000000, 0.000000000]
-> Test results with original matrix (should get columns of b): #
-> list(a*transpose(matrix(res1)),a*transpose(matrix(res2)))?
 | 1.000000000 |  | 1.000000000 |
 |             |  |             |
[| 1.000000000 |, | 2.000000000 |]
 |             |  |             |
 | 1.000000000 |  | 3.000000000 |
```

tridag

Description:

This function solves a system of tridiagonal linear equations.

tridag(a, b, c, r) $\longrightarrow$ u

Where:

a is a real list of length n representing the first lower diagonal of the input matrix with a_1 not being used,
b is a real list of length n representing the leading diagonal of the input matrix,
c is a real list of length n representing the first upper diagonal of the input matrix with c_n not being used,
r a real list representing the right hand side of the system of equations,
u a floating point list representing the solution vector.

Example:

This example solves a tridiagonal system of equations.

```
-> Diagonal elements #
-> diag1 : [1.0, 2.0, 3.0];
-> Super-diagonal elements #
-> super1 : [2.0, 3.0, 0.0]; final 0.0 being a dummy element here #
-> Sub-diagonal elements #
-> sub1 : [0.0, 2.0, 3.0]; initial 0.0 being a dummy element here #
-> Right-hand side vector #
-> right1 : [1.0, 2.0, 3.0];
-> res : tridag(sub1,diag1,super1,right1)?
[-0.200000000, 0.600000000, 0.400000000]
-> Test solution (should get right1): #
-> ans : listn(1,length(diag1));
-> res : join([0],res,[0]);
-> for k from 1 by 1 to length(diag1) do
  ans : put(ans,k,proj(sub1,k)*proj(res,k)
                  + proj(diag1,k)*proj(res,k+1)
                  + proj(super1,k)*proj(res,k+2));
-> ans?
[1.000000000, 2.000000000, 3.000000000]
```

mprove

Description:

This function computes an iterative improvement for the solution to a system of linear equations $A.x = b$.

mprove(a, alud, indx, b, x) $\longrightarrow$ x

Where:

a is a real square matrix of dimension n,
alud is a real square matrix being the LU decomposition of a as returned by *ludcmp*,
indx is an integer list of permutions as returned by *ludcmp*,
b is a real list representing the right hand side of the system of equations,
x on input is a real list representing an approximation to the solution to the system of equations and on output an improved solution.

Example:

In this example `ludcmp` and `lubksb` are first used to solve the system of equations `a*x = b`. This solution is then altered and `mprove` used to return it to its original state.

```
-> a : matrix([1.0, 2.0, 1.0, 4.0, 5.0],
              [2.0, 3.0, 1.0, 5.0, 1.0],
              [3.0, 4.0, 1.0, 1.0, 2.0],
              [4.0, 5.0, 1.0, 2.0, 3.0],
              [5.0, 1.0, 1.0, 3.0, 4.0]);
-> b : [1.0, 1.0, 1.0, 1.0, 1.0];
-> aa : ludcmp(a);
-> Solution vector for the equations #
-> x : lubksb(first(aa),proj(aa,2),b)?
[0.000000000, 0.000000000, 1.000000000, 0.000000000, 0.000000000]
-> Now alter this solution and let mprove fix it #
-> xx : [1.0, 1.0, 1.0, 1.0, 1.0];
-> Solution vector recovered by mprove (should be x)#
-> mprove(a,first(aa),proj(aa,2),b,xx)?
[0.000000000, 0.000000000, 1.000000000, 0.000000000, 0.000000000]
```

vander

Description:

This function solves a system of linear equations represented by a Vandermonde matrix i.e. the system

$$\sum_{i=1}^{N} x_i^{k-1} w_i = q_k, \quad 1 \le k \le N.$$

vander(x, q) $\longrightarrow$ w

Where:

x is a real list,
q is a real list having the same length as x,
w is a floating point list representing the solution to the Vandermonde system.

Example:

This example solves a Vandermond system of equations for a solution vector `w` and then tests the solution.

```
-> a : [1.0, 1.5, 2.0, 2.5, 3.0];
-> q : a;
-> Solution vector: #
-> w : vander(a,q)?
[-1.0833, 4.0000, -3.0000, 1.3333, -0.2500]
-> Vandermonde matrix corresponding to a: #
-> aa : matrix([1.0,     1.0,     1.0,     1.0,     1.0],
               [1.0,     1.5,     2.0,     2.5,     3.0],
               [1.0, (1.5)^2, (2.0)^2, (2.5)^2, (3.0)^2],
               [1.0, (1.5)^3, (2.0)^3, (2.5)^3, (3.0)^3],
               [1.0, (1.5)^4, (2.0)^4, (2.5)^4, (3.0)^4])?
| 1.0000  1.0000  1.0000   1.0000   1.0000  |
|                                           |
| 1.0000  1.5000  2.0000   2.5000   3.0000  |
|                                           |
| 1.0000  2.2500  4.0000   6.2500   9.0000  |
|                                           |
| 1.0000  3.3750  8.0000   15.6250  27.0000 |
|                                           |
| 1.0000  5.0625  16.0000  39.0625  81.0000 |
-> Test solution vector (should get matrix of q (or a)): #
-> transpose(aa*transpose(matrix(w)))?
| 1.0000  1.5000  2.0000  2.5000  3.0000 |
```

toeplz

Description:

This function solves a system of linear equations represented by a Toeplitz matrix i.e. the system

$$\sum_{j=1}^{N} R_{N+1-j} x_j = y_i, \quad 1 \le i \le N.$$

toeplz(r, y) $\longrightarrow$ x

Where:

r is a real list of length $2n-1$,
y is a real list of length n,
x is a floating point list representing the solution to the Toeplitz system.

Example:

In this example a Toeplitz system of equations is solved and then the solution is tested.

```
-> y : [0.1, 0.2, 0.3, 0.4, 0.5];
-> r : [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9];
-> Solution vector: #
-> x : toeplz(r,y)?
[0.000000002, -0.000000000, 0.000000000, -0.000000007, 1.000000006]
-> Toeplitz matrix corresponding to r: #
-> rr : matrix([0.5, 0.4, 0.3, 0.2, 0.1],
               [0.6, 0.5, 0.4, 0.3, 0.2],
               [0.7, 0.6, 0.5, 0.4, 0.3],
               [0.8, 0.7, 0.6, 0.5, 0.4],
               [0.9, 0.8, 0.7, 0.6, 0.5]);
-> Test of solution (should get matrix of y): #
-> transpose(rr*transpose(matrix(x)))?
| 0.100000001  0.200000003  0.300000012  0.400000006  0.500000000 |
```

svdcmp

Description:

This function performs a singular value decomposition of a matrix i.e. $A = U.W.V^T$.

svdcmp(a) $\longrightarrow$ [u, w, v]

Where:

a is a real matrix,
u is a floating point matrix representing U,
w is a floating point list representing the floating point diagonal matrix W,
v is a floating point matrix representing V.

Example:

In this example the singular value decomposition of a matrix `aa` is first computed using `svdcmp`. The solution is then tested reproducing `aa`.

```
-> aa : matrix([1.0, 2.0, 3.0, 4.0, 5.0],
               [2.0, 2.0, 3.0, 4.0, 5.0],
               [3.0, 3.0, 3.0, 4.0, 5.0],
               [4.0, 4.0, 4.0, 4.0, 5.0],
               [5.0, 5.0, 5.0, 5.0, 5.0]);
-> Decompose matrix a: #
-> res : svdcmp(aa)?
 | -0.363117162  -0.641960897  0.523446615   -0.377808941  0.198232637  |
 |                                                                       |
 | -0.381645782  -0.440409392  -0.174588651  0.601674216   -0.517583815 |
 |                                                                       |
[| -0.419648474  -0.100585952  -0.639803546  0.021295506   0.635590708  |,
 |                                                                       |
 | -0.479064385  0.270817683   -0.251816938  -0.614292563  -0.506347683 |
 |                                                                       |
 | -0.562925303  0.557194781   0.471976796   0.342694749   0.180131319  |
[19.597636627, 3.185096026, 0.749885629, 0.385722754, 0.276932217],
| -0.363117162  0.641960897   -0.523446615  0.377808941   -0.198232637 |
|                                                                       |
| -0.381645782  0.440409392   0.174588651   -0.601674216  0.517583815  |
|                                                                       |
| -0.419648474  0.100585952   0.639803546   -0.021295506  -0.635590708 |]
|                                                                       |
| -0.479064385  -0.270817683  0.251816938   0.614292563   0.506347683  |
|                                                                       |
| -0.562925303  -0.557194781  -0.471976796  -0.342694749  -0.180131319 |
```

```
-> uu : first(res); vv : last(res);
-> w : proj(res,2);
-> ww : matrix([proj(w,1), 0, 0, 0, 0],
               [0, proj(w,2), 0, 0, 0],
               [0, 0, proj(w,3), 0, 0],
               [0, 0, 0, proj(w,4), 0],
               [0, 0, 0, 0, proj(w,5)]);
-> test solution (should get aa): #
-> uu*ww*transpose(vv)?
| 1.000000000  2.000000000  3.000000000  4.000000000  5.000000000 |
|                                                                  |
| 2.000000000  2.000000000  3.000000000  4.000000000  5.000000000 |
|                                                                  |
| 3.000000000  3.000000000  3.000000000  4.000000000  5.000000000 |
|                                                                  |
| 4.000000000  4.000000000  4.000000000  4.000000000  5.000000000 |
|                                                                  |
| 5.000000000  5.000000000  5.000000000  5.000000000  5.000000000 |
```

svbksb

Description:

This function performs a singular value backsubstitution for the system of equations $A.x = b$ where the matrix A is represented by its singular value decomposition as retured by *svdcmp*.

svbksb(u, w, v, b) —→ x

Where:

u is a real matrix as returned by *svdcmp*,
w is a real list as returned by *svdcmp*,
v is a real matrix as returned by *svdcmp*,
b is a real list representing the right hand side of the system of equations,
x a floating point list representing the solution vector.

Example:

In this example a system of equations `a*x = b` is solved by first computing a singluar value decomposition and then using `svbskb` to obtain a solution.

```
-> a : matrix([1.4, 2.1, 2.1, 7.4, 9.6],
              [1.6, 1.5, 1.1, 0.7, 5.0],
              [3.8, 8.0, 9.6, 5.4, 8.8],
              [4.6, 8.2, 8.4, 0.4, 8.0],
              [2.6, 2.9, 0.1, 9.6, 7.7]);
-> Solution vectors: #
-> b : [1.1, 1.6, 4.7, 9.1, 0.1];
-> Decompose matrix a: #
-> res : svdcmp(a);
-> uu : first(res); vv : last(res);
-> w : proj(res,2);
-> x : svbksb(uu,w,vv,b)?
[-5.313079544, 5.735671550, -2.507606432, -1.058741235, 0.999381217]
-> Test solution (should get elements of b): #
-> transpose(a*transpose(matrix(x)))?
| 1.100000024  1.600000024  4.699999809  9.100000381  0.100000001 |
```

sparse

Description:

This function solves a sparse system of linear $A.x = b$ equations using the conjugate-gradient method.

sparse(b, asub, atsub, x) $\longrightarrow$ [x, rsq]

Where:

b is a real list representing the right hand side of the system of equations,
asub is a Senac function of one real list which computes for each list x a list representing the value of $A.x$,
atsub is a Senac function of one real list which computes for each list x a list representing the value of $A^T.x$,
x on input is a real list being an approximation to the solution and on output the computed solution.
rsq is a floating point number being the sum of the squares of the components of the residual vector $A.x - b$.

Pre-variables:

sparse_eps [1.0d-8] a floating point number being the required RMS accuracy.

Example:

In this example a system of equations `aa*x = b` is solved. The functions `asub` and `atsub` in this case do not take into account the sparse structure of `aa`.

```
-> aa : matrix([ 1.0,  2.0,  0.0,  0.0, 0.0],
               [-2.0,  1.0,  2.0,  0.0, 0.0],
               [ 0.0, -2.0,  1.0,  2.0, 0.0],
               [ 0.0,  0.0, -2.0,  1.0, 2.0],
               [ 0.0,  0.0,  0.0, -2.0, 1.0]);
-> define asub & atsub as functions which compute aa*x
 and transpose(aa)*x for a given x #
-> asub(x) := row(1,transpose(aa*transpose(matrix(x))));
-> atsub(x) := row(1,transpose(transpose(aa)*transpose(matrix(x))));
-> x : [0.0, 0.0, 0.0, 0.0, 0.0];
-> b : [3.0, 1.0, 1.0, 1.0, -1.0];
-> res : sparse(b,asub,atsub,x)?
[[1.000000000, 1.000000000, 1.000000000, 1.000000000, 1.000000000],
0.000000000]
-> x : first(res);
-> Test solution (should get b): #
-> column(1, aa*transpose(matrix(x)))?
[3.000000000, 1.0000000000, 1.00000000000, -1.0000000000]
```