[ < ] [ > ] [ << ] [ Up ] [ >> ] [Top] [Contents] [Index] [ ? ]

# 61. linearalgebra



---

[ < ] [ > ] [ << ] [ Up ] [ >> ] [Top] [Contents] [Index] [ ? ]

## 61.1 Introduction to linearalgebra

`linearalgebra` is a collection of functions for linear algebra.

Example:

```
(%i1) M : matrix ([1, 2], [1, 2]);
                            [ 1  2 ]
(%o1)                       [      ]
                            [ 1  2 ]
(%i2) nullspace (M);
                               [  1  ]
                               [     ]
(%o2)                     span([   1 ])
                               [ - - ]
                               [   2 ]
(%i3) columnspace (M);
                                [ 1 ]
(%o3)                      span([   ])
                                [ 1 ]
(%i4) ptriangularize (M - z*ident(2), z);
                         [ 1   2 - z   ]
(%o4)                    [             ]
                         [           2 ]
                         [ 0  3 z - z  ]
(%i5) M : matrix ([1, 2, 3], [4, 5, 6], [7, 8, 9]) - z*ident(3);
                     [ 1 - z    2      3   ]
                     [                     ]
(%o5)                [   4    5 - z    6   ]
                     [                     ]
                     [   7      8    9 - z ]
(%i6) MM : ptriangularize (M, z);
              [ 4  5 - z            6           ]
              [                                 ]
              [                2                ]
              [     66        z    102 z   132  ]
              [ 0   --      - -- + ----- + ---  ]
(%o6)         [     49        7     49     49   ]
              [                                 ]
              [               3        2        ]
              [           49 z    245 z    147 z ]
              [ 0    0    ----- - ------ - ----- ]
              [            264      88      44   ]
(%i7) algebraic : true;
(%o7)                          true
(%i8) tellrat (MM [3, 3]);
                         3       2
(%o8)                  [z  - 15 z  - 18 z]
(%i9) MM : ratsimp (MM);
               [ 4  5 - z           6           ]
```

```
               [                                 ]
               [                2                ]
(%o9)          [     66      7 z  - 102 z - 132 ]
               [ 0   --    - ------------------ ]
               [     49              49          ]
               [                                 ]
               [ 0    0             0            ]
(%i10) nullspace (MM);
                        [        1         ]
                        [                  ]
                        [   2              ]
                        [  z  - 14 z - 16  ]
                        [  --------------  ]
(%o10)             span([        8         ])
                        [                  ]
                        [    2             ]
                        [   z  - 18 z - 12 ]
                        [ - -------------- ]
                        [         12       ]
(%i11) M : matrix ([1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12],
                   [13, 14, 15, 16]);
                       [ 1   2   3   4  ]
                       [                ]
                       [ 5   6   7   8  ]
(%o11)                 [                ]
                       [ 9   10  11  12 ]
                       [                ]
                       [ 13  14  15  16 ]
(%i12) columnspace (M);
                           [ 1  ]  [ 2  ]
                           [    ]  [    ]
                           [ 5  ]  [ 6  ]
(%o12)                span([    ], [    ])
                           [ 9  ]  [ 10 ]
                           [    ]  [    ]
                           [ 13 ]  [ 14 ]
(%i13) apply ('orthogonal_complement, args (nullspace (transpose (M))));
                           [ 0 ]  [   1  ]
                           [   ]  [      ]
                           [ 1 ]  [   0  ]
(%o13)                span([   ], [      ])
                           [ 2 ]  [ - 1  ]
                           [   ]  [      ]
                           [ 3 ]  [ - 2  ]
```

.

@ref{Category: Linear algebra} · @ref{Category: Share packages} · @ref{Category: Package linearalgebra}

[ < ] [ > ]   [ << ] [ Up ] [ >> ]         [Top] [Contents] [Index] [ ? ]

## 61.2 Functions and Variables for linearalgebra

Function: **addmatrices** *(f, M_1, ..., M_n)*

Using the function *f* as the addition function, return the sum of the matrices *M_1*, ..., *M_n*. The function *f* must accept any number of arguments (a Maxima nary function).

Examples:

```
(%i1) m1 : matrix([1,2],[3,4])$
(%i2) m2 : matrix([7,8],[9,10])$
(%i3) addmatrices('max,m1,m2);
(%o3) matrix([7,8],[9,10])
(%i4) addmatrices('max,m1,m2,5*m1);
(%o4) matrix([7,10],[15,20])
```

.

@ref{Category: Package linearalgebra}

Function: **blockmatrixp** *(M)*

Return true if and only if *M* is a matrix and every entry of *M* is a matrix.

.

@ref{Category: Package linearalgebra} · @ref{Category: Predicate functions}

Function: **columnop** *(M, i, j, theta)*

If *M* is a matrix, return the matrix that results from doing the column operation `C_i <- C_i - theta * C_j`. If *M* doesn't have a row *i* or *j*, signal an error.

.

@ref{Category: Package linearalgebra}

Function: **columnswap** *(M, i, j)*

If *M* is a matrix, swap columns *i* and *j*. If *M* doesn't have a column *i* or *j*, signal an error.

.

@ref{Category: Package linearalgebra}

Function: **columnspace** *(M)*

If *M* is a matrix, return `span (v_1, ..., v_n)`, where the set `{v_1, ..., v_n}` is a basis for the column space of *M*. The span of the empty set is `{0}`. Thus, when the column space has only one member, return `span ()`.

.

@ref{Category: Package linearalgebra}

Function: **copy** *(e)*

Return a copy of the Maxima expression *e*. Although *e* can be any Maxima expression, the copy function is the most useful when *e* is either a list or a matrix; consider:

```
(%i1) m : [1,[2,3]]$
(%i2) mm : m$
(%i3) mm[2][1] : x$
(%i4) m;
(%o4)                      [1,[x,3]]
(%i5) mm;
(%o5)                      [1,[x,3]]
```

Let's try the same experiment, but this time let *mm* be a copy of *m*

```
(%i6) m : [1,[2,3]]$
(%i7) mm : copy(m)$
(%i8) mm[2][1] : x$
(%i9) m;
(%o9)                     [1,[2,3]]
(%i10) mm;
(%o10)                    [1,[x,3]]
```

This time, the assignment to *mm* does not change the value of *m*.

·

@ref{Category: Package linearalgebra}

Function: **cholesky** *(M)*
Function: **cholesky** *(M, field)*

Return the Cholesky factorization of the matrix selfadjoint (or hermitian) matrix *M*. The second argument defaults to 'generalring.' For a description of the possible values for *field*, see `lu_factor`.

·

@ref{Category: Matrix decompositions} · @ref{Category: Package linearalgebra}

Function: **ctranspose** *(M)*

Return the complex conjugate transpose of the matrix *M*. The function `ctranspose` uses `matrix_element_transpose` to transpose each matrix element.

·

@ref{Category: Package linearalgebra}

Function: **diag_matrix** *(d_1, d_2,...,d_n)*

Return a diagonal matrix with diagonal entries *d_1*, *d_2*,...,*d_n*. When the diagonal entries are matrices, the zero entries of the returned matrix are zero matrices of the appropriate size; for example:

```
(%i1) diag_matrix(diag_matrix(1,2),diag_matrix(3,4));

                            [ [ 1  0 ]  [ 0  0 ] ]
                            [ [      ]  [      ] ]
                            [ [ 0  2 ]  [ 0  0 ] ]
(%o1)                       [                    ]
                            [ [ 0  0 ]  [ 3  0 ] ]
                            [ [      ]  [      ] ]
                            [ [ 0  0 ]  [ 0  4 ] ]
(%i2) diag_matrix(p,q);

                                   [ p  0 ]
(%o2)                              [      ]
                                   [ 0  q ]
```

.

@ref{Category: Package linearalgebra}

Function: **dotproduct** *(u, v)*

Return the dotproduct of vectors *u* and *v*. This is the same as `conjugate (transpose (u)) . v`. The arguments *u* and *v* must be column vectors.

.

@ref{Category: Package linearalgebra}

Function: **eigens_by_jacobi** *(A)*
Function: **eigens_by_jacobi** *(A, field_type)*

Computes the eigenvalues and eigenvectors of *A* by the method of Jacobi rotations. *A* must be a symmetric matrix (but it need not be positive definite nor positive semidefinite). *field_type* indicates the computational field, either `floatfield` or `bigfloatfield`. If *field_type* is not specified, it defaults to `floatfield`.

The elements of *A* must be numbers or expressions which evaluate to numbers via `float` or `bfloat` (depending on *field_type*).

Examples:

```
(%i1) S: matrix([1/sqrt(2), 1/sqrt(2)],[-1/sqrt(2), 1/sqrt(2)]);
                     [     1         1    ]
                     [  -------   ------- ]
                     [  sqrt(2)   sqrt(2) ]
(%o1)                [                    ]
                     [      1        1    ]
                     [ - -------  ------- ]
                     [   sqrt(2)  sqrt(2) ]
(%i2) L : matrix ([sqrt(3), 0], [0, sqrt(5)]);
                      [ sqrt(3)     0    ]
```

```
(%o2)                 [                   ]
                      [    0     sqrt(5) ]
(%i3) M : S . L . transpose (S);
            [ sqrt(5)   sqrt(3)  sqrt(5)   sqrt(3) ]
            [ ------- + -------  ------- - ------- ]
            [    2         2        2         2    ]
(%o3)       [                                      ]
            [ sqrt(5)   sqrt(3)  sqrt(5)   sqrt(3) ]
            [ ------- - -------  ------- + ------- ]
            [    2         2        2         2    ]
(%i4) eigens_by_jacobi (M);
The largest percent change was 0.1454972243679
The largest percent change was 0.0
number of sweeps: 2
number of rotations: 1
(%o4) [[1.732050807568877, 2.23606797749979],
                        [  0.70710678118655   0.70710678118655 ]
                        [                                      ]]
                        [ - 0.70710678118655  0.70710678118655 ]
(%i5) float ([[sqrt(3), sqrt(5)], S]);
(%o5) [[1.732050807568877, 2.23606797749979],
                        [  0.70710678118655   0.70710678118655 ]
                        [                                      ]]
                        [ - 0.70710678118655  0.70710678118655 ]
(%i6) eigens_by_jacobi (M, bigfloatfield);
The largest percent change was 1.454972243679028b-1
The largest percent change was 0.0b0
number of sweeps: 2
number of rotations: 1
(%o6) [[1.732050807568877b0, 2.23606797749979b0],
                [  7.071067811865475b-1   7.071067811865475b-1 ]
                [                                              ]]
                [ - 7.071067811865475b-1  7.071067811865475b-1 ]
```

.

@ref{Category: Matrix decompositions} · @ref{Category: Package linearalgebra}

Function: **get_lu_factors** *(x)*

When `x = lu_factor (A)`, then `get_lu_factors` returns a list of the form `[P, L, U]`, where *P* is a permutation matrix, *L* is lower triangular with ones on the diagonal, and *U* is upper triangular, and `A = P L U`.

.

@ref{Category: Package linearalgebra}

Function: **hankel** *(col)*
Function: **hankel** *(col, row)*

Return a Hankel matrix *H*. The first column of *H* is *col*; except for the first entry, the last row of *H* is *row*. The default for *row* is the zero vector with the same length as *col*.

.

@ref{Category: Package linearalgebra}

Function: **hessian** *(f, x)*

Returns the Hessian matrix of *f* with respect to the list of variables *x*. The `(i, j)`-th element of the Hessian matrix is `diff(f, x[i], 1, x[j], 1)`.

Examples:

```
(%i1) hessian (x * sin (y), [x, y]);
                     [   0       cos(y)   ]
(%o1)                [                    ]
                     [ cos(y)  - x sin(y) ]
(%i2) depends (F, [a, b]);
(%o2)                       [F(a, b)]
(%i3) hessian (F, [a, b]);
                        [   2      2   ]
                        [  d F    d F  ]
                        [  ---   ----- ]
                        [    2   da db ]
                        [  da          ]
(%o3)                   [              ]
                        [   2      2   ]
                        [  d F    d F  ]
                        [ -----    --- ]
                        [ da db     2  ]
                        [         db   ]
```

·

@ref{Category: Differential calculus} · @ref{Category: Package linearalgebra}

Function: **hilbert_matrix** *(n)*

Return the *n* by *n* Hilbert matrix. When *n* isn't a positive integer, signal an error.

·

@ref{Category: Package linearalgebra}

Function: **identfor** *(M)*
Function: **identfor** *(M, fld)*

Return an identity matrix that has the same shape as the matrix *M*. The diagonal entries of the identity matrix are the multiplicative identity of the field *fld*; the default for *fld* is *generalring*.

The first argument *M* should be a square matrix or a non-matrix. When *M* is a matrix, each entry of *M* can be a square matrix - thus *M* can be a blocked Maxima matrix. The matrix can be blocked to any (finite) depth.

See also `zerofor`

·

@ref{Category: Package linearalgebra}

Function: **invert_by_lu** *(M, (rng generalring))*

Invert a matrix *M* by using the LU factorization. The LU factorization is done using the ring *rng*.

·

@ref{Category: Package linearalgebra}

Function: **jacobian** *(f, x)*

Returns the Jacobian matrix of the list of functions *f* with respect to the list of variables *x*. The `(i, j)`-th element of the Jacobian matrix is `diff(f[i], x[j])`.

Examples:

```
(%i1) jacobian ([sin (u - v), sin (u * v)], [u, v]);
                  [ cos(v - u)  - cos(v - u) ]
(%o1)             [                          ]
                  [ v cos(u v)   u cos(u v)  ]
(%i2) depends ([F, G], [y, z]);
(%o2)                  [F(y, z), G(y, z)]
(%i3) jacobian ([F, G], [y, z]);
                           [ dF  dF ]
                           [ --  -- ]
                           [ dy  dz ]
(%o3)                      [        ]
                           [ dG  dG ]
                           [ --  -- ]
                           [ dy  dz ]
```

·

@ref{Category: Differential calculus} · @ref{Category: Package linearalgebra}

Function: **kronecker_product** *(A, B)*

Return the Kronecker product of the matrices *A* and *B*.

·

@ref{Category: Package linearalgebra}

Function: **listp** *(e, p)*
Function: **listp** *(e)*

Given an optional argument *p*, return `true` if *e* is a Maxima list and *p* evaluates to `true` for every list element. When `listp` is not given the optional argument, return `true` if *e* is a Maxima list. In all other cases, return `false`.

·

@ref{Category: Package linearalgebra} · @ref{Category: Predicate functions}

<u>Function:</u> **locate_matrix_entry** *(M, r_1, c_1, r_2, c_2, f, rel)*

The first argument must be a matrix; the arguments *r_1* through *c_2* determine a sub-matrix of *M* that consists of rows *r_1* through *r_2* and columns *c_1* through *c_2*.

Find a entry in the sub-matrix *M* that satisfies some property. Three cases:

(1) `rel = 'bool` and *f* a predicate:

Scan the sub-matrix from left to right then top to bottom, and return the index of the first entry that satisfies the predicate *f*. If no matrix entry satisfies *f*, return `false`.

(2) `rel = 'max` and *f* real-valued:

Scan the sub-matrix looking for an entry that maximizes *f*. Return the index of a maximizing entry.

(3) `rel = 'min` and *f* real-valued:

Scan the sub-matrix looking for an entry that minimizes *f*. Return the index of a minimizing entry.

·

@ref{Category: Package linearalgebra}

<u>Function:</u> **lu_backsub** *(M, b)*

When `M = lu_factor (A, field)`, then `lu_backsub (M, b)` solves the linear system `A x = b`.

·

@ref{Category: Package linearalgebra}

<u>Function:</u> **lu_factor** *(M, field)*

Return a list of the form `[LU, perm, fld]`, or `[LU, perm, fld, lower-cnd upper-cnd]`, where

(1) The matrix *LU* contains the factorization of *M* in a packed form. Packed form means three things: First, the rows of *LU* are permuted according to the list *perm*. If, for example, *perm* is the list `[3,2,1]`, the actual first row of the *LU* factorization is the third row of the matrix *LU*. Second, the lower triangular factor of m is the lower triangular part of *LU* with the diagonal entries replaced by all ones. Third, the upper triangular factor of *M* is the upper triangular part of *LU*.

(2) When the field is either `floatfield` or `complexfield`, the numbers *lower-cnd* and *upper-cnd* are lower and upper bounds for the infinity norm condition number of *M*. For all fields, the condition number might not be estimated; for such fields, `lu_factor` returns a two item list. Both the lower and upper bounds can differ from their true values by arbitrarily large factors. (See also `mat_cond`.)

The argument *M* must be a square matrix.

The optional argument *fld* must be a symbol that determines a ring or field. The pre-defined fields and rings are:

(a) `generalring` - the ring of Maxima expressions, (b) `floatfield` - the field of floating point numbers of the type double, (c) `complexfield` - the field of complex floating point numbers of the type double, (d) `crering` - the ring of Maxima CRE expressions, (e) `rationalfield` - the field of rational numbers, (f) `runningerror` - track the all floating point rounding errors, (g) `noncommutingring` - the ring of Maxima expressions where multiplication is the non-commutative dot operator.

When the field is `floatfield`, `complexfield`, or `runningerror`, the algorithm uses partial pivoting; for all other fields, rows are switched only when needed to avoid a zero pivot.

Floating point addition arithmetic isn't associative, so the meaning of 'field' differs from the mathematical definition.

A member of the field `runningerror` is a two member Maxima list of the form `[x,n]`,where *x* is a floating point number and `n` is an integer. The relative difference between the 'true' value of `x` and `x` is approximately bounded by the machine epsilon times `n`. The running error bound drops some terms that of the order the square of the machine epsilon.

There is no user-interface for defining a new field. A user that is familiar with Common Lisp should be able to define a new field. To do this, a user must define functions for the arithmetic operations and functions for converting from the field representation to Maxima and back. Additionally, for ordered fields (where partial pivoting will be used), a user must define functions for the magnitude and for comparing field members. After that all that remains is to define a Common Lisp structure `mring`. The file `mring` has many examples.

To compute the factorization, the first task is to convert each matrix entry to a member of the indicated field. When conversion isn't possible, the factorization halts with an error message. Members of the field needn't be Maxima expressions. Members of the `complexfield`, for example, are Common Lisp complex numbers. Thus after computing the factorization, the matrix entries must be converted to Maxima expressions.

See also `get_lu_factors`.

Examples:

```
(%i1) w[i,j] := random (1.0) + %i * random (1.0);
(%o1)          w     := random(1.) + %i random(1.)
                i, j
(%i2) showtime : true$
Evaluation took 0.00 seconds (0.00 elapsed)
```

```
(%i3) M : genmatrix (w, 100, 100)$
Evaluation took 7.40 seconds (8.23 elapsed)
(%i4) lu_factor (M, complexfield)$
Evaluation took 28.71 seconds (35.00 elapsed)
(%i5) lu_factor (M, generalring)$
Evaluation took 109.24 seconds (152.10 elapsed)
(%i6) showtime : false$

(%i7) M : matrix ([1 - z, 3], [3, 8 - z]);
                        [ 1 - z    3   ]
(%o7)                   [              ]
                        [   3    8 - z ]
(%i8) lu_factor (M, generalring);
          [ 1 - z         3        ]
          [                        ]
(%o8)    [[   3            9       ], [1, 2], generalring]
          [ -----  - z - ----- + 8 ]
          [ 1 - z        1 - z     ]
(%i9) get_lu_factors (%);
                  [   1    0 ]  [ 1 - z         3        ]
        [ 1  0 ]  [          ]  [                        ]
(%o9)  [[      ], [   3      ], [                9       ]]
        [ 0  1 ]  [ -----  1 ]  [   0    - z - ----- + 8 ]
                  [ 1 - z    ]  [              1 - z     ]
(%i10) %[1] . %[2] . %[3];
                        [ 1 - z    3   ]
(%o10)                  [              ]
                        [   3    8 - z ]
```

·

@ref{Category: Matrix decompositions} · @ref{Category: Package linearalgebra}

Function: **mat_cond** *(M, 1)*
Function: **mat_cond** *(M, inf)*

Return the *p*-norm matrix condition number of the matrix *m*. The allowed values for *p* are 1 and *inf*. This function uses the LU factorization to invert the matrix *m*. Thus the running time for `mat_cond` is proportional to the cube of the matrix size; `lu_factor` determines lower and upper bounds for the infinity norm condition number in time proportional to the square of the matrix size.

·

@ref{Category: Package linearalgebra}

Function: **mat_norm** *(M, 1)*
Function: **mat_norm** *(M, inf)*
Function: **mat_norm** *(M, frobenius)*

Return the matrix *p*-norm of the matrix *M*. The allowed values for *p* are 1, `inf`, and `frobenius` (the Frobenius matrix norm). The matrix *M* should be an unblocked matrix.

·

@ref{Category: Package linearalgebra}

Function: **matrixp** *(e, p)*
Function: **matrixp** *(e)*

Given an optional argument *p*, return `true` if *e* is a matrix and *p* evaluates to `true` for every matrix element. When `matrixp` is not given an optional argument, return `true` if `e` is a matrix. In all other cases, return `false`.

See also `blockmatrixp`

·

@ref{Category: Package linearalgebra} · @ref{Category: Predicate functions}

Function: **matrix_size** *(M)*

Return a two member list that gives the number of rows and columns, respectively of the matrix *M*.

·

@ref{Category: Package linearalgebra}

Function: **mat_fullunblocker** *(M)*

If *M* is a block matrix, unblock the matrix to all levels. If *M* is a matrix, return *M*; otherwise, signal an error.

·

@ref{Category: Package linearalgebra}

Function: **mat_trace** *(M)*

Return the trace of the matrix *M*. If *M* isn't a matrix, return a noun form. When *M* is a block matrix, `mat_trace(M)` returns the same value as does `mat_trace(mat_unblocker(m))`.

·

@ref{Category: Package linearalgebra}

Function: **mat_unblocker** *(M)*

If *M* is a block matrix, unblock *M* one level. If *M* is a matrix, `mat_unblocker (M)` returns *M*; otherwise, signal an error.

Thus if each entry of *M* is matrix, `mat_unblocker (M)` returns an unblocked matrix, but if each entry of *M* is a block matrix, `mat_unblocker (M)` returns a block matrix with one less level of blocking.

If you use block matrices, most likely you'll want to set `matrix_element_mult` to `"."` and `matrix_element_transpose` to `'transpose`. See also `mat_fullunblocker`.

Example:

```
(%i1) A : matrix ([1, 2], [3, 4]);
                            [ 1  2 ]
(%o1)                       [      ]
                            [ 3  4 ]
(%i2) B : matrix ([7, 8], [9, 10]);
                            [ 7  8  ]
(%o2)                       [       ]
                            [ 9  10 ]
(%i3) matrix ([A, B]);
                     [ [ 1  2 ]  [ 7  8  ] ]
(%o3)                [ [      ]  [       ] ]
                     [ [ 3  4 ]  [ 9  10 ] ]
(%i4) mat_unblocker (%);
                         [ 1  2  7  8  ]
(%o4)                    [             ]
                         [ 3  4  9  10 ]
```

·

@ref{Category: Package linearalgebra}

Function: **nonnegintegerp** *(n)*

Return `true` if and only if *n* `>= 0` and *n* is an integer.

·

@ref{Category: Package linearalgebra} · @ref{Category: Predicate functions}

Function: **nullspace** *(M)*

If *M* is a matrix, return `span (v_1, ..., v_n)`, where the set `{v_1, ..., v_n}` is a basis for the nullspace of *M*. The span of the empty set is `{0}`. Thus, when the nullspace has only one member, return `span ()`.

·

@ref{Category: Package linearalgebra}

Function: **nullity** *(M)*

If *M* is a matrix, return the dimension of the nullspace of *M*.

·

@ref{Category: Package linearalgebra}

Function: **orthogonal_complement** *(v_1, ..., v_n)*

Return `span (u_1, ..., u_m)`, where the set `{u_1, ..., u_m}` is a basis for the orthogonal complement of the set `(v_1, ..., v_n)`.

Each vector *v_1* through *v_n* must be a column vector.

.

@ref{Category: Package linearalgebra}

<u>Function:</u> **polynomialp** *(p, L, coeffp, exponp)*
<u>Function:</u> **polynomialp** *(p, L, coeffp)*
<u>Function:</u> **polynomialp** *(p, L)*

Return `true` if *p* is a polynomial in the variables in the list *L*, The predicate *coeffp* must evaluate to `true` for each coefficient, and the predicate *exponp* must evaluate to `true` for all exponents of the variables in *L*. If you want to use a non-default value for *exponp*, you must supply *coeffp* with a value even if you want to use the default for *coeffp*.

`polynomialp (p, L, coeffp)` is equivalent to `polynomialp (p, L, coeffp, 'nonnegintegerp)`.

`polynomialp (p, L)` is equivalent to `polynomialp (p, L, 'constantp, 'nonnegintegerp)`.

The polynomial needn't be expanded:

```
(%i1) polynomialp ((x + 1)*(x + 2), [x]);
(%o1)                          true
(%i2) polynomialp ((x + 1)*(x + 2)^a, [x]);
(%o2)                          false
```

An example using non-default values for coeffp and exponp:

```
(%i1) polynomialp ((x + 1)*(x + 2)^(3/2), [x], numberp, numberp);
(%o1)                          true
(%i2) polynomialp ((x^(1/2) + 1)*(x + 2)^(3/2), [x], numberp,
                                                        numberp);
(%o2)                          true
```

Polynomials with two variables:

```
(%i1) polynomialp (x^2 + 5*x*y + y^2, [x]);
(%o1)                          false
(%i2) polynomialp (x^2 + 5*x*y + y^2, [x, y]);
(%o2)                          true
```

.

@ref{Category: Package linearalgebra} · @ref{Category: Predicate functions}

Function: **polytocompanion** *(p, x)*

If *p* is a polynomial in *x*, return the companion matrix of *p*. For a monic polynomial *p* of degree *n*, we have `p = (-1)^n charpoly (polytocompanion (p, x))`.

When *p* isn't a polynomial in *x*, signal an error.

.

@ref{Category: Package linearalgebra}

Function: **ptriangularize** *(M, v)*

If *M* is a matrix with each entry a polynomial in *v*, return a matrix *M2* such that

(1) *M2* is upper triangular,

(2) `M2 = E_n ... E_1 M`, where *E_1* through *E_n* are elementary matrices whose entries are polynomials in *v*,

(3) `|det (M)| = |det (M2)|`,

Note: This function doesn't check that every entry is a polynomial in *v*.

.

@ref{Category: Package linearalgebra}

Function: **rowop** *(M, i, j, theta)*

If *M* is a matrix, return the matrix that results from doing the row operation `R_i <- R_i - theta * R_j`. If *M* doesn't have a row *i* or *j*, signal an error.

.

@ref{Category: Package linearalgebra}

Function: **rank** *(M)*

Return the rank of that matrix *M*. The rank is the dimension of the column space. Example:

```
(%i1) rank(matrix([1,2],[2,4]));
(%o1)                                  1
(%i2) rank(matrix([1,b],[c,d]));
Proviso:  {d - b c # 0}
(%o2)                                  2
```

.

@ref{Category: Package linearalgebra}

Function: **rowswap** *(M, i, j)*

If *M* is a matrix, swap rows *i* and *j*. If *M* doesn't have a row *i* or *j*, signal an error.

.

@ref{Category: Package linearalgebra}

Function: **toeplitz** *(col)*
Function: **toeplitz** *(col, row)*

Return a Toeplitz matrix *T*. The first first column of *T* is *col*; except for the first entry, the first row of *T* is *row*. The default for *row* is complex conjugate of *col*. Example:

```
(%i1)  toeplitz([1,2,3],[x,y,z]);

                                  [ 1  y  z ]
                                  [         ]
(%o1)                             [ 2  1  y ]
                                  [         ]
                                  [ 3  2  1 ]
(%i2)  toeplitz([1,1+%i]);

                              [   1     1 - %I ]
(%o2)                         [                ]
                              [ %I + 1    1    ]
```

.

@ref{Category: Package linearalgebra}

Function: **vandermonde_matrix** *([x_1, ..., x_n])*

Return a *n* by *n* matrix whose *i*-th row is `[1, x_i, x_i^2, ... x_i^(n-1)]`.

.

@ref{Category: Package linearalgebra}

Function: **zerofor** *(M)*
Function: **zerofor** *(M, fld)*

Return a zero matrix that has the same shape as the matrix *M*. Every entry of the zero matrix is the additive identity of the field *fld*; the default for *fld* is *generalring*.

The first argument *M* should be a square matrix or a non-matrix. When *M* is a matrix, each entry of *M* can be a square matrix - thus *M* can be a blocked Maxima matrix. The matrix can be blocked to any (finite) depth.

See also `identfor`

.

@ref{Category: Package linearalgebra}

Function: **zeromatrixp** *(M)*

If *M* is not a block matrix, return `true` if `is (equal (e, 0))` is true for each element *e* of the matrix *M*. If *M* is a block matrix, return `true` if `zeromatrixp` evaluates to `true` for each element of *e*.

·

@ref{Category: Package linearalgebra} · @ref{Category: Predicate functions}

---

[ << ] [ >> ] [Top] [Contents] [Index] [ ? ]

This document was generated by *root* on *July, 13 2009* using *texi2html 1.76*.