`linalg` — library for linear algebra

Table of contents

1 Introduction

An overview of all the available functions can be obtained by using the MuPAD function `info`. The output of the call:

```
>> info(linalg)
```

looks like (the output was cutted due to its size):

```
 Library 'linalg': the linear algebra package

 -- Interface:
 linalg::addCol,           linalg::addRow,
 linalg::adjoint,          linalg::angle,
 linalg::basis,            linalg::charmat,
 linalg::charpoly,         linalg::col,
 ...
```

After being exported, library functions can also be used by their short notation. The function call `export(linalg)` exports all functions of `linalg`. After that one can use the function name `gaussElim` instead of `linalg::gaussElim`, for example.

Please note, that user-defined procedures that use functions of the library `linalg` should always use the long notation `linalg::`*function*, in order to make sure that the unambiguity of the function name is guaranteed.

The most easiest way to define a matrix A is using the command `matrix`. The following defines a 2×2 matrix:

```
>> A := matrix([[1, 2], [3, 2]])

                                  +-      -+
                                  |  1, 2  |
                                  |        |
                                  |  3, 2  |
                                  +-      -+
```

Now you can add or multiply matrices using the standard arithmetical operators of MuPAD:

```
>> A * A, 2 * A, 1/A

            +-        -+  +-       -+  +-             -+
            |  7,  6   |  |  2, 4   |  |  -1/2,  1/2   |
            |          |, |         |, |               |
            |  9, 10   |  |  6, 4   |  |   3/4, -1/4   |
            +-        -+  +-       -+  +-             -+
```

or use functions of the `linalg` library:

```
>> linalg::det(A)
```

The domain type returned by `matrix` is `Dom::Matrix()`:

```
>> domtype(A)

                          Dom::Matrix()
```

which is introduced in the following section.

2 Data Types for Matrices and Vectors

The library `linalg` is based on the domain constructors `Dom::Matrix` and `Dom::SquareMatrix`. These constructors enable the user to define matrices and they offer matrix arithmetic and several functions for matrix manipulation.

A domain created by `Dom::Matrix` represents matrices of arbitrary rows and columns over a specified ring. The domain constructor `Dom::Matrix` expects a coefficient ring of category `Cat::Rng` (a ring without unit) as argument. If no argument is given, the domain of matrices is created, that represents matrices over the field of arithmetical expressions, i.e., the domain `Dom::ExpressionField()`.

Be careful with calculations with matrices over this coefficient domain, because their entries usually do not have a unique representation (e.g., there is more than one representation of zero). You can normalize the components of such a matrix `A` with the command `map( A,normal )`.

The library `Dom` offers standard coefficient domains, such as the field of rational numbers (`Dom::Rational`), the ring of integers (`Dom::Integer`), the residue classes of integers (`Dom::IntegerMod(n)`) for an integer `n`, or rings of polynomials (`Dom::DistributedPolynomial(ind, R)` or `Dom::Polynomial(R)`, where `ind` is the list of variables and `R` is the coefficient ring).

A domain created by the domain constructor `Dom::SquareMatrix` represents the ring of square matrices over a specified coefficient domain. `Dom::SquareMatrix` expects the number of rows of the square matrices and optionally a coefficient ring of category `Cat::Rng`.

There are several possibilities to define matrices of a domain created by `Dom::Matrix` or `Dom::SquareMatrix`. A matrix can be created by giving a two-dimensional array, a list of the matrix components, or a function that generates the matrix components:

```
>> delete a, b, c, d:
   A := matrix([[a, b], [c, d]])

                          +-       -+
                          |  a, b   |
                          |         |
                          |  c, d   |
                          +-       -+
```

The command `matrix` actually is an abbreviation for the domain `Dom::Matrix()`.

To create diagonal matrices one should use the option `Diagonal` (the third argument of `matrix` is either a function or a list):

```
>> B := matrix(2, 2, [2, -2], Diagonal)

                           +-       -+
                           |  2,  0  |
                           |         |
                           |  0, -2  |
                           +-       -+
```

The following two examples show the meaning of the third argument:

```
>> delete x: matrix(2, 2, () -> x), matrix(2, 2, x)

               +-      -+  +-                  -+
               |  x, x  |  |  x(1, 1), x(1, 2)  |
               |        |, |                    |
               |  x, x  |  |  x(2, 1), x(2, 2)  |
               +-      -+  +-                  -+
```

The arithmetical operators of MuPAD are used to perform matrix arithmetic:

```
>> A * B - 2 * B

                    +-                    -+
                    |  2 a - 4,    -2 b    |
                    |                      |
                    |    2 c,   - 2 d + 4  |
                    +-                    -+
```

```
>> 1/A

               +-                              -+
               |         d             b        |
               |  - -----------,  -----------   |
               |    - a d + b c   - a d + b c   |
               |                                |
               |        c               a       |
               |   -----------,  - -----------  |
               |   - a d + b c     - a d + b c  |
               +-                              -+
```

Next we create the 2 × 2 generalized Hilbert matrix (see also `linalg::hilbert`) as a matrix of the ring of 2-dimensional square matrices:

```
>> MatQ2 := Dom::SquareMatrix(2, Dom::Rational)
```

```
                     Dom::SquareMatrix(2, Dom::Rational)
```

```
>> H2 := MatQ2((i, j) -> 1/(i + j - 1))

                                +-           -+
                                |   1,  1/2   |
                                |             |
                                |  1/2, 1/3   |
                                +-           -+
```

A *vector* with n components is a $1 \times n$ matrix (a row vector) or a $n \times 1$ matrix (a column vector).

The components of a matrix or a vector are accessed using the index operator, i.e., `A[i,j]` returns the component of the row with index `i` and column with index `j`.

The input `A[i, j]:= x` sets the (i, j)-th component of the matrix `A` to the value of `x`.

The index operator can also be used to extract submatrices by giving ranges of integers as its arguments:

```
>> A := Dom::Matrix(Dom::Integer)(
     [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
   )

                                 +-          -+
                                 |  1, 2, 3   |
                                 |            |
                                 |  4, 5, 6   |
                                 |            |
                                 |  7, 8, 9   |
                                 +-          -+
```

```
>> A[1..3, 1..2], A[3..3, 1..3]

                            +-       -+
                            |  1, 2   |
                            |         |  +-         -+
                            |  4, 5   |, | 7, 8, 9 |
                            |         |  +-         -+
                            |  7, 8   |
                            +-       -+
```

See also the function `linalg::submatrix`.

3 Remarks on Improving Runtime

The runtime of user-defined procedures that use functions of the `linalg` library and methods of the constructors `Dom::Matrix` and `Dom::SquareMatrix` can be considerably improved in certain cases.

1. Some of the functions of the `linalg` library correspond to certain methods of the domain constructor `Dom::Matrix` in their name and functionality. These functions are implemented by calling relevant methods of the domain to that they belong, apart from additional argument checking. These functions enable an user-friendly usage on the interactive level after exporting.

   However, in user-defined procedures the methods of the corresponding domain should be used directly to avoid additionally calls of procedures.

   For example standard matrix manipulation functions such as deleting, extracting or swapping of rows and columns are defined as methods of the domain constructors `Dom::Matrix` and `Dom::SquareMatrix`.

   The method `"gaussElim"` offers a Gaussian elimination process for each domain created by these constructors.

2. When creating a new matrix the method `"new"` is called. It converts each component of the matrix explicitly into a component the component ring, which may be time consuming.

   However, matrices and vectors are often the results of computations, whose components already are elements of the component ring. Thus, the conversion of the entries is not necessary. To take this into account, the domain constructors `Dom::Matrix` and `Dom::SquareMatrix` offer a method `"create"` to define matrices in the usual way but without the conversion of the components.

   Please note that this method does not test its arguments. Thus it should be used with caution.

3. A further possibility of achieving better runtimes using functions of `linalg` or methods of the constructor `Dom::Matrix` is to store functions and methods that are called more than once in local variables. This enables a faster access of these functions and methods.

The following example shows how a user-defined procedure using functions of `linalg` and methods of the domain constructor `Dom::Matrix` may look like. It computes the adjoint of a square matrix defined over a commutative ring (see `Cat::CommutativeRing`):

```
>> adjoint := proc(A)
       local n, R, i, j, a, Ai, Mat,
             // local variables to store often used methods
             det, delRow, delCol, Rnegate;
   begin
       if args(0) <> 1 then
           error("wrong number of arguments")
       end_if;
```

```
        Mat := A::dom;       // the domain of A
        R := Mat::coeffRing; // the component ring of A
        n := Mat::matdim(A); // the dimension of A; faster than calling
                             // 'linalg::matdim'!
        if testargs() then
            if Mat::hasProp(Cat::Matrix) <> TRUE then
                error("expecting a matrix")
            elif not R::hasProp( Cat::CommutativeRing ) then
                error("expecting matrix over a 'Cat::CommutativeRing'")
            elif n[1] <> n[2] then
                error("expecting a square matrix")
            end_if
        end_if;

        // store often used methods in local variables:
        det := linalg::det;
        delRow := Mat::delRow;  // faster than calling 'linalg::delRow'!
        delCol := Mat::delCol;  // faster than calling 'linalg::delCol'!
        Rnegate := R::_negate;  // faster than using the '-
' operator!

        n := Mat::matdim(A)[1]; // faster than calling 'linalg::nrows'!
        a := array(1..n, 1..n);

        for i from 1 to n do
            Ai := delCol(A, i);
            for j from 1 to n do
                a[i, j] := det(delRow(Ai, j));
                if i + j mod 2 = 1 then
                    a[i, j] := Rnegate(a[i, j])
                end_if
            end_for
        end_for;

        // create a new matrix: use Mat::create instead of Mat::new
        // because the entries of the array are already el-
ements of R
        return(Mat::create(a))
    end_proc:
```

We give an example:

```
>> MatZ6 := Dom::Matrix(Dom::IntegerMod(6)):
   adjoint(MatZ6([[1, 5], [2, 4]]))

                      +-                  -+
                      |  4 mod 6, 1 mod 6  |
                      |                    |
                      |  4 mod 6, 1 mod 6  |
                      +-                  -+
```

4 Changes since Version 1.4

We have changed the names of some functions in the `linalg` library:

Old Name	New Name
`cholesky`	`factorCholesky`
`dimen`	`matdim`
`eigenValues`	`eigenvalues`
`eigenVectors`	`eigenvectors`
`extractMatrix`	`submatrix`
`isHermitian`	`isHermitean`
`ogSystem`	`orthog`
`onSystem`	see `ogSystem`
`linearSolve`	`matlinsolve`
`linearSolveLU`	`matlinsolveLU`
`vectorDimen`	`vecdim`

The following functions are new in the `linalg` library:

New Function	Description
`companion`	Companion matrix for univariate polynomials.
`factorLU`	LU-decomposition of a matrix.
`frobeniusForm`	Frobenius form of a matrix.
`hessenberg`	Hessenberg matrix of a matrix.
`hilbert`	Hilbert matrix.
`inverseLU`	Inverse of a square matrix by LU-decomposition.
`invhilbert`	Inverse of a Hilbert matrix.
`minpoly`	Minimal polynomial of a matrix.
`permanent`	Permanent of a matrix.
`pseudoInverse`	Moore-Penrose inverse of a matrix.
`smithForm`	Smith normal form of a matrix.
`substitute`	Replace a part of a matrix by another matrix.
`vandermondeSolve`	Solver for Vandermode systems.
`wiedemann`	Solving of linear systems using Wiedemann's algorithm.

`linalg::addCol` – **linear combination of matrix columns**

`linalg::addCol(A, c1, c2, s)` returns a copy of the matrix A in which column c_2 of A is replaced by $s \cdot \mathrm{col}(A, c_1) + \mathrm{col}(A, c_2)$.

Call(s):

- `linalg::addCol(A, c1, c2, s)`

Parameters:

`A`	— an $m \times n$ matrix of a domain of category `Cat::Matrix`
`c1, c2`	— the column indices: positive integers $\leq n$
`s`	— an expression that can be converted into the component ring of `A`

Return Value: a matrix of the same domain type as `A`.

Related Functions: `linalg::addRow`, `linalg::col`, `linalg::multCol`, `linalg::multRow`, `Dom::Matrix`

Example 1. The following defines a 3×3 matrix over the integers:

```
>> A := Dom::Matrix(Dom::Integer)(
     [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
   )

                          +-         -+
                          |  1, 2, 3  |
                          |           |
                          |  4, 5, 6  |
                          |           |
                          |  7, 8, 9  |
                          +-         -+
```

We replace the 2nd column by $-\mathrm{col}(A, 1) + \mathrm{col}(A, 2)$, i.e., we subtract the first column from the second:

```
>> linalg::addCol(A, 1, 2, -1)

                          +-         -+
                          |  1, 1, 3  |
                          |           |
                          |  4, 1, 6  |
                          |           |
                          |  7, 1, 9  |
                          +-         -+
```

Example 2. The following defines a 2 × 3 matrix over the reals:

```
>> B := Dom::Matrix(Dom::Real)(
     [[sin(2), 0, 1], [1, PI, 0]]
   )

                              +-                -+
                              |  sin(2),  0, 1  |
                              |                 |
                              |     1,   PI, 0  |
                              +-                -+
```

If `s` is an expression that does not represent a real number then an error message is reported. The following tries to replace the 1st column by $x \cdot \mathrm{col}(B, 3) + \mathrm{col}(B, 1)$, where x is an identifier which cannot be converted into the component ring `Dom::Real` of B:

```
>> delete x: linalg::addCol(B, 3, 1, x)

 Error: unable to convert x [linalg::addCol]
```

Example 3. If symbolic expressions are involved, then one may define matrices over a component ring created by `Dom::ExpressionField`. The following example defines a matrix over this default component ring:

```
>> delete a11, a12, a21, a22, x:
   C := matrix([[a11, a12], [a21, a22]])

                                +-          -+
                                |  a11, a12  |
                                |            |
                                |  a21, a22  |
                                +-          -+
```

We retry the input from the previous example:

```
>> linalg::addCol(C, 2, 1, x)

                            +-                  -+
                            |  a11 + x a12, a12  |
                            |                    |
                            |  a21 + x a22, a22  |
                            +-                  -+
```

`linalg::addRow` – linear combination of matrix rows

`linalg::addRow(A, r1, r2, s)` returns a copy of the matrix A in which row r_2 of A is replaced by $s \cdot \text{row}(A, r_1) + \text{row}(A, r_2)$.

Call(s):

`linalg::addRow(A, r1, r2, s)`

Parameters:

`A` — an $m \times n$ matrix of a domain of category `Cat::Matrix`
`r1, r2` — the row indices: positive integers $\leq m$
`s` — an expression that can be converted into the component ring of `A`

Return Value: a matrix of the same domain type as `A`.

Related Functions: `linalg::addCol`, `linalg::row`, `linalg::multCol`, `linalg::multRow`

Example 1. The following defines a 3×3 matrix over the integers:

```
>> A := Dom::Matrix(Dom::Integer)(
     [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
   )
```

```
                          +-         -+
                          |  1, 2, 3  |
                          |           |
                          |  4, 5, 6  |
                          |           |
                          |  7, 8, 9  |
                          +-         -+
```

We replace the 2nd row by $-\text{row}(A, 1) + \text{row}(A, 2)$, i.e., we subtract the first row from the second:

```
>> linalg::addRow(A, 1, 2, -1)
```

```
                          +-         -+
                          |  1, 2, 3  |
                          |           |
                          |  3, 3, 3  |
                          |           |
                          |  7, 8, 9  |
                          +-         -+
```

Example 2. The following defines a 2 × 3 matrix over the reals:

```
>> B := Dom::Matrix(Dom::Real)(
     [[sin(2), 0, 1], [1, PI, 0]]
   )
```

```
                                  +-                 -+
                                  |  sin(2),  0, 1  |
                                  |                  |
                                  |     1,   PI, 0  |
                                  +-                 -+
```

If `s` is an expression that does not represent a real number then an error message is reported. The following tries to replace the 1st row by $x \cdot \operatorname{row}(B, 2) + \operatorname{row}(B, 1)$, where x is an identifier which cannot be converted into the component ring `Dom::Real` of B:

```
>> delete x: linalg::addRow(B, 2, 1, x)
```

```
 Error: unable to convert x [linalg::addRow]
```

Example 3. If symbolic expressions are involved, then one may define matrices over the component ring created by `Dom::ExpressionField`. The following example defines a matrix over this default component ring:

```
>> delete a11, a12, a21, a22, x:
   C := matrix([[a11, a12], [a21, a22]])
```

```
                                    +-           -+
                                    |  a11, a12  |
                                    |             |
                                    |  a21, a22  |
                                    +-           -+
```

We retry the input from the previous example:

```
>> linalg::addRow(C, 2, 1, x)
```

```
                            +-                             -+
                            |  a11 + x a21, a12 + x a22  |
                            |                             |
                            |      a21,         a22      |
                            +-                             -+
```

`linalg::adjoint` – **Adjoint of a matrix**

`linalg::adjoint(A)` computes the adjoint Adj(A) of the $n \times n$ matrix A. The adjoint matrix satisfies the equation $A \cdot \mathrm{Adj}(A) = \det(A) \cdot I_n$, where I_n is the $n \times n$ identity matrix.

Call(s):

- `linalg::adjoint(A)`

Parameters:

`A` — a square matrix of a domain of category `Cat::Matrix`

Return Value: a matrix of the same domain type as `A`.

Related Functions: `linalg::det`

Details:

- The component ring of `A` must be of category `Cat::CommutativeRing`.

Example 1. We define a matrix over the rationals:

```
>> MatQ := Dom::Matrix( Dom::Rational ):
   A := MatQ( [[0, 2, 1], [2, 1, 0], [1, 0, 2]] )

                          +-         -+
                          |  0, 2, 1  |
                          |           |
                          |  2, 1, 0  |
                          |           |
                          |  1, 0, 2  |
                          +-         -+
```

Then the adjoint matrix of `A` is given by:

```
>> Ad := linalg::adjoint(A)

                        +-             -+
                        |   2, -4, -1   |
                        |               |
                        |  -4, -1,  2   |
                        |               |
                        |  -1,  2, -4   |
                        +-             -+
```

We check the property of the adjoint matrix `Ad` mentioned above:

```
>> A * Ad = linalg::det(A)*MatQ::identity(3)
```

```
+-              -+   +-              -+
|  -9,  0,  0   |   |  -9,  0,  0   |
|                |   |                |
|   0, -9,  0   | = |   0, -9,  0   |
|                |   |                |
|   0,  0, -9   |   |   0,  0, -9   |
+-              -+   +-              -+
```

Background:

- The adjoint of a square matrix A is the matrix whose (i, j)-th entry is the (j, i)-th cofactor of A.

  The (j, i)-th *cofactor* of A is defined by $a'_{ij} = (-1)^{i+j} \det A(i|j)$, where $A(i|j)$ is the submatrix of A obtained from A by deleting the i-th row and j-th column.

`linalg::angle` – The angle between two vectors

`linalg::angle(u,v)` computes the angle φ between the two vectors `u` and `v`, defined by

$$\varphi = \arccos\left(\frac{\vec{u} * \vec{v}}{|\vec{u}|\,|\vec{v}|}\right).$$

Call(s):

- `linalg::angle(u, v)`

Parameters:

`u, v` — vectors of the same dimension; a vector is a $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`

Return Value: an arithmetical expression.

Related Functions: `arccos`, `linalg::scalarProduct`, `linalg::vecdim`

Details:

- For the equation

$$\varphi = \arccos\left(\frac{\vec{u} * \vec{v}}{|\vec{u}|\,|\vec{v}|}\right)$$

the product $\vec{u} * \vec{v}$ denotes the scalar product of two vectors given by `linalg::scalarProduct`, and $|\cdot|$ the 2-norm of a vector, i.e., $|\vec{u}| = \sqrt{\vec{u} * \vec{u}}$.

- `linalg::angle` does not check if the computation is defined in the corresponding component ring. This can lead to an error message, as shown in Example 2.
- The following relationship between the angle between $\vec{u}$ and $\vec{v}$ and the angle between $\vec{u}$ and $-\vec{v}$ holds: $\varphi(\vec{u}, \vec{v}) = \pi - \varphi(\vec{u}, \vec{v})$.
- An error message is returned if the vectors are not defined over the same component ring.

Example 1. We compute the angle between the two vectors $\begin{pmatrix} 2 \\ 5 \end{pmatrix}$ and $\begin{pmatrix} -3 \\ 3 \end{pmatrix}$:

```
>> phi := linalg::angle(
     matrix([2, 5]), matrix([-3, 3])
   )

                                  /   1/2   1/2 \
                                  | 18    29    |
                            arccos| ----------- |
                                  \     58      /
```

We use the function `float` to get a floating-point approximation of this number:

```
>> float(phi)

                                  1.165904541
```

We give two further examples:

```
>> linalg::angle(
     matrix([1, -1]), matrix([1, 1])
   )

                                      PI
                                      --
                                      2

>> linalg::angle(
     matrix([1, 1]), matrix([-1, -1])
   )

                                      PI
```

Example 2. `linalg::angle` does not check whether the term $\frac{\vec{u}*\vec{v}}{|\vec{u}||\vec{v}|}$ is defined in the corresponding component ring.

As an example, we try to compute the angle between two vectors with components in $\mathbb{Z}_7$:

```
>> MatZ7 := Dom::Matrix(Dom::IntegerMod(7))

                  Dom::Matrix(Dom::IntegerMod(7))
```

The following call leads to an error because the 2-norm cannot be computed:

```
>> linalg::angle(MatZ7([1, 1]), MatZ7([-1, -1]))

 Error: no integer exponent [(Dom::IntegerMod(7))::_power]
```

Note that the domain `Dom::IntegerMod(7)` does not implement the square root of an element, therefore in MuPAD you cannot compute the angle of any two vectors over $\mathbb{Z}_7$.

`linalg::basis` – **basis for a vector space**

`linalg::basis(S)` returns a basis for the vector space spanned by the vectors in the set or list *S*.

Call(s):

- `linalg::basis(S)`

Parameters:

`S` — a set or list of *n*-dimensional vectors; a vector is a $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`

Return Value: a set or a list of vectors, respectively.

Related Functions: `linalg::intBasis`, `linalg::sumBasis`, `lllint`

Details:

- `linalg::basis(S)` removes those vectors in `S` that are linearly dependent on other vectors in `S`. The result is a basis for the vector space spanned by the vectors in `S`.
- For an ordered basis of vectors, `S` should be a list of vectors.
- The vectors in `S` must be defined over the same component ring.
- The component ring of the vectors in `S` must be a field, i.e., it must be of category `Cat::Field`.

Example 1. We define the domain of matrices over $\mathbb{Q}$:

```
>> MatQ := Dom::Matrix(Dom::Rational):
```

and compute a basis for the vector space spanned by the vectors $\begin{pmatrix} 3 \\ -2 \end{pmatrix}$, $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 5 \\ -3 \end{pmatrix}$:

```
>> v1 := MatQ([3, -2]): v2 := MatQ([1, 0]): v3 := MatQ([5, -
3]):
   linalg::basis([v1, v2, v3])
```

```
                                -- +-     -+  +-    -+ --
                                |  |   3   |  |  1   |  |
                                |  |       |, |      |  |
                                |  |  -2   |  |  0   |  |
                                -- +-     -+  +-    -+ --
```

If not a list but a set of vectors is given, then the basis returned contains the same vectors. But the order of the vectors in the set depends on the internal order (see `sysorder` and `DOM_SET`), i.e., the order of the vectors appears to be random:

```
>> b := linalg::basis({v1, v2, v3}): op(b, 1)
```

```
                                       +-    -+
                                       |  1   |
                                       |      |
                                       |  0   |
                                       +-    -+
```

`linalg::charmat` – characteristic matrix

`linalg::charmat(A, x)` returns the characteristic matrix $xI_n - A$ of the $n \times n$ matrix A, where I_n denotes the $n \times n$ identity matrix.

Call(s):

- `linalg::charmat(A, x)`

Parameters:

`A` — a square matrix of a domain of category `Cat::Matrix`
`x` — an indeterminate

Return Value: a matrix of the domain `Dom::Matrix(Dom::DistributedPolynomial([x],R))`, where `R` is the component ring of `A`.

Related Functions: `linalg::charpoly`

Details:

- The component ring of `A` must be a commutative ring, i.e., a domain of category `Cat::CommutativeRing`.
- The characteristic matrix $M = xI_n - A$ of A can be evaluated at a point $x = u$ via `evalp(M, x = u)`. See example 2.

Example 1. We define a matrix over the rational numbers:

```
>> A := Dom::Matrix(Dom::Rational)([[1, 2], [3, 4]])

                          +-       -+
                          |  1, 2   |
                          |         |
                          |  3, 4   |
                          +-       -+
```

and compute the characteristic matrix of A in the variable x:

```
>> MA := linalg::charmat(A, x)

                      +-                -+
                      |  x - 1,    -2    |
                      |                  |
                      |    -3,  x - 4    |
                      +-                -+
```

The determinant of the matrix `MA` is a polynomial in x, the characteristic polynomial of the matrix A:

```
>> pA := linalg::det(MA)

                            2
                           x  - 5 x - 2

>> domtype(pA)

    Dom::DistributedPolynomial([x], Dom::Rational, LexOrder)
```

Of course, we can compute the characteristic polynomial of A directly via `linalg::charpoly`:

```
>> linalg::charpoly(A, x)

                            2
                           x  - 5 x - 2
```

The result is of the same domain type as the polynomial `pA`.

Example 2. We define a matrix over the complex numbers:

```
>> B := Dom::Matrix(Dom::Complex)([[1 + I, 1], [1, 1 - I]])

                      +-              -+
                      |  1 + I,   1    |
                      |                |
                      |    1,   1 - I  |
                      +-              -+
```

The characteristic matrix of `B` in the variable z is:

```
>> MB := linalg::charmat(B, z)

                +-                              -+
                |  z - (1 + I),       -1         |
                |                                |
                |       -1,      z - (1 - I)     |
                +-                              -+
```

We evaluate `MB` at $z = i$ and get the matrix:

```
>> evalp(MB, z = I)

                      +-                -+
                      |  -1,      -1     |
                      |                  |
                      |  -1, - 1 + 2 I   |
                      +-                -+
```

Note that this is a matrix of the domain type `Dom::Matrix(Dom::Complex)`:

```
>> domtype(%)

                   Dom::Matrix(Dom::Complex)
```

Changes:

- `linalg::charmat` used to be `linalg::charMatrix`.

`linalg::charpoly` – characteristic polynomial of a matrix

`linalg::charpoly(A, x)` computes the characteristic polynomial of the matrix A. The characteristic polynomial of a $n \times n$ matrix is defined by $p_A(x) := \det(xI_n - A)$, where I_n denotes the $n \times n$ identity matrix.

Call(s):

- `linalg::charpoly(A, x)`

Parameters:

`A` — a square matrix of a domain of category `Cat::Matrix`
`x` — an indeterminate

Return Value: a polynomial of the domain `Dom::DistributedPolynomial([x],R)`, where `R` is the component ring of `A`.

Related Functions: `linalg::charmat`, `linalg::det`, `linalg::hessenberg`, `linalg::minpoly`

Details:

- The component ring of `A` must be a commutative ring, i.e., a domain of category `Cat::CommutativeRing`.

Example 1. We define a matrix over the rational numbers:

```
>> A := Dom::Matrix(Dom::Rational)([[1, 2], [3, 4]])

                          +-       -+
                          |  1, 2   |
                          |         |
                          |  3, 4   |
                          +-       -+
```

Then the characteristic polynomial $p_A(x)$ is given by:

```
>> linalg::charpoly(A, x)

                           2
                          x  - 5 x - 2
```

It is of the domain type:

```
>> domtype(%)

    Dom::DistributedPolynomial([x], Dom::Rational, LexOrder)
```

Example 2. We define a matrix over $\mathbb{Z}_7$:

```
>> B := Dom::Matrix(Dom::IntegerMod(7))([[1, 2], [3, 0]])

                         +-                    -+
                         |  1 mod 7, 2 mod 7  |
                         |                      |
                         |  3 mod 7, 0 mod 7  |
                         +-                    -+
```

The characteristic polynomial $p_B(x)$ of `B` is given by:

```
>> p := linalg::charpoly(B, x)

                  2
    (1 mod 7) x  + (6 mod 7) x + (1 mod 7)
```

We compute the zeros of $p_B(x)$, i.e., the eigenvalues of the matrix `B`:

```
>> solve(p)

                         x in {3 mod 7, 5 mod 7}
```

Background:

- `linalg::charpoly` implements Hessenberg's algorithm to compute the characteristic polynomial of a square matrix A. See: Henri Cohen: *A Course in Computational Algebraic Number Theory*, GTM 138, Springer Verlag.

  This algorithm works for any field and requires only $O(n^3)$ field operations, in contrast to $O(n^4)$ when computing the determinant of the characteristic matrix of A.

  Since the size of the components of A in intermediate computations of Hessenberg's algorithm can swell extremely, it is only applied for matrices over `Dom::Float` and `Dom::IntegerMod`.

  For any other component ring, the characteristic polynomial is computed using the Berkowitz algorithm. Reference: A. Jounaidi: *The Berkowitz Algorithm, Maple and Computing the Characteristic Polynomial in an Arbitrary Commutative Ring*. Equipe de Mathèmatiques de Besancon, Universitè de Franche - Comtè, 25030 Besancon Cedex, May 1996.

Changes:

- `linalg::charpoly` used to be `linalg::charPolynomial`.

`linalg::col` – extract columns of a matrix

`linalg::col(A, c)` extracts the c-th column vector of the matrix A.

Call(s):

- `linalg::col(A, c)`
- `linalg::col(A, c1..c2)`
- `linalg::col(A, list)`

Parameters:

`A` — an $m \times n$ matrix of a domain of category `Cat::Matrix`
`c` — the column index: a positive integer $\leq n$
`c1..c2` — a range of column indices (positive integers $\leq n$)
`list` — a list of column indices (positive integers $\leq n$)

Return Value: a single column vector or a list of column vectors; a column vector is an $m \times 1$ matrix of category `Cat::Matrix(R)`, where R is the component ring of `A`.

Related Functions: `linalg::row`, `linalg::delCol`, `linalg::delRow`, `linalg::setCol`, `linalg::setRow`

Details:

- `linalg::col(A, c1..c2)` returns a list of column vectors whose indices are in the range `c1..c2`. If `c2 < c1` then the empty list `[]` is returned.
- `linalg::col(A, list)` returns a list of column vectors whose indices are contained in `list` (in the same order).

Example 1. We define a matrix over $\mathbb{Q}$:

```
>> A := Dom::Matrix(Dom::Rational)(
     [[1, 1/5, 2], [-3/2, 0, 5]]
   )

                       +-                -+
                       |    1,  1/5, 2   |
                       |                 |
                       |  -3/2,   0,  5  |
                       +-                -+
```

and illustrate the three different input formats for `linalg::col`:

```
>> linalg::col(A, 2)

                            +-      -+
                            |  1/5   |
                            |        |
                            |   0    |
                            +-      -+
>> linalg::col(A, [2, 1, 3])

              -- +-      -+  +-      -+  +-    -+ --
              |  |  1/5  |  |    1   |  |  2  |  |
              |  |       |, |        |, |     |  |
              |  |   0   |  |  -3/2  |  |  5  |  |
              -- +-      -+  +-      -+  +-    -+ --
```

```
>> linalg::col(A, 2..3)

                    -- +-      -+  +-    -+ --
                    |  |  1/5  |  |  2  |  |
                    |  |       |, |     |  |
                    |  |   0   |  |  5  |  |
                    -- +-      -+  +-    -+ --
```

`linalg::companion` – Companion matrix of a univariate polynomial

`linalg::companion(p)` returns the companion matrix associated with the polynomial p.

Call(s):

- `linalg::companion(p<, x>)`

Parameters:

`p` — an univariate polynomial, or a polynomial expression
`x` — an indeterminate

Return Value: a matrix of the domain `Dom::Matrix(R)`.

Details:

- `p` must be monic and of degree one at least.
- If `p` is a polynomial, i.e., an object of type `DOM_POLY`, then specifying `x` has no effect.
- If `p` is a polynomial, then the component ring of the returned matrix is the coefficient ring of `p`, except in two cases for built-in coefficient rings: if the coefficient ring of `p` is `Expr` then the domain `Dom::ExpressionField()` is the component ring of the companion matrix. If it is `IntMod(m)` then the companion matrix is defined over the ring `Dom::IntegerMod(m)` (see example 2).

- If `p` is a polynomial expression, then the companion matrix is defined over `Dom::ExpressionField()`.
- If `p` is a polynomial expression containing several symbolic indeterminates then `x` must be specified and distinguishes the indeterminate `x` from the other symbolic parameters.

Example 1. We start with the following polynomial expression:

```
>> delete a0, a1, a2, a3:
   p := x^4 + a3*x^3 + a2*x^2 + a1*x + a0

                                4     2        3
                 a0 + x a1 + x   + x   a2 + x   a3
```

To compute the companion matrix of p with respect to x we must specify the second parameter x, because the expression `p` contains the indeterminates a_0, a_1, a_2, a_3 and x:

```
>> linalg::companion(p)

 Error: multivariate expression [linalg::companion]
```

```
>> linalg::companion(p, x)

                        +-               -+
                        |  0, 0, 0, -a0  |
                        |                |
                        |  1, 0, 0, -a1  |
                        |                |
                        |  0, 1, 0, -a2  |
                        |                |
                        |  0, 0, 1, -a3  |
                        +-               -+
```

Of course, we can compute the companion matrix of p with respect to a_0 as well:

```
>> linalg::companion(p, a0)

                 +-           4     2        3    -+
                 | - x a1 - x   - x   a2 - x   a3 |
                 +-                               -+
```

The following fails with an error message, because the polynomial p is not monic with respect to a_1:

```
>> linalg::companion(p, a1)

 Error: polynomial is not monic [linalg::companion]
```

Example 2. If we enter a polynomial over the built-in coefficient domain `Expr`, then the companion matrix is defined over the standard component ring for matrices (the domain `Dom::ExpressionField()`):

```
>> C := linalg::companion(poly(x^2 + 10*x + PI, [x]))

                                    +-         -+
                                    |  0, -PI   |
                                    |           |
                                    |  1, -10   |
                                    +-         -+

>> domtype(C)

                                    Dom::Matrix()
```

If we define a polynomial over the build-in coefficient domain `IntMod(m)`, then the companion matrix is defined over the corresponding component ring `Dom::IntegerMod(m)`, as shown in the next example:

```
>> p := poly(x^2 + 10*x + 7, [x], IntMod(3))

                             2
                     poly(x  + x + 1, [x], IntMod(3))

>> C := linalg::companion(p)

                               +-                  -+
                               |  0 mod 3, 2 mod 3  |
                               |                    |
                               |  1 mod 3, 2 mod 3  |
                               +-                  -+

>> domtype(C)

                      Dom::Matrix(Dom::IntegerMod(3))
```

Background:

- The companion matrix of the polynomial $x^n + a_{n_1}x^{n-1} + \ldots + a_1 x + a_0$ is the matrix:

$$C = \begin{pmatrix} 0 & 0 & \cdots & 0 & -a_0 \\ 1 & 0 & \cdots & 0 & -a_1 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & -a_{n-1} \\ 0 & 0 & \cdots & 1 & -a_n \end{pmatrix}.$$

- The companion matrix of a univariate polynomial p of degree n is an $n \times n$ matrix C with $p_C = p$, where p_C is the characteristic polynomial of C.

Changes:

- `linalg::companion` is a new function.

`linalg::concatMatrix` – join matrices horizontally

`linalg::concatMatrix(A, B1, <B2, ...>)` returns the matrix formed by joining the matrices $A, B_1, B_2, \ldots$ horizontally.

Call(s):

- `linalg::concatMatrix(A, B1<, B2, ...>)`

Parameters:

`A, B1, B2, ...` — matrices of a domain of category `Cat::Matrix`

Return Value: a matrix of the domain type `Dom::Matrix(R)`, where `R` is the component ring of `A`.

Related Functions: `linalg::stackMatrix`

Details:

- The matrices `B1, B2, ...` are converted into the matrix domain `Dom::Matrix(R)`, where `R` is the component ring of `A`.

  An error message is raised if one of these conversions fails, or if the matrices do not have the same number of rows as the matrix `A`.

- A short form of `linalg::concatMatrix` is available through the dot operator `.`, i.e., instead of `linalg::concatMatrix(A, B)` one may use the short form `A . B`.

Example 1. We define the matrix:

```
>> A := matrix([[sin(x), x], [-x, cos(x)]])

                    +-                 -+
                    |  sin(x),    x     |
                    |                   |
                    |    -x,    cos(x)  |
                    +-                 -+
```

and append the 2×2 identity matrix to the right of `A`:

```
>> I2 := matrix::identity(2):
   linalg::concatMatrix(A, I2)
```

```
+-                        -+
|  sin(x),    x,   1, 0  |
|                          |
|    -x,   cos(x), 0, 1  |
+-                        -+
```

The short form for this operation is:

```
>> A . I2
```

```
+-                        -+
|  sin(x),    x,   1, 0  |
|                          |
|    -x,   cos(x), 0, 1  |
+-                        -+
```

Example 2. We define a matrix from the ring of 2×2 square matrices:

```
>> SqMatQ := Dom::SquareMatrix(2, Dom::Rational):
   A := SqMatQ([[1, 2], [3, 4]])
```

```
+-      -+
|  1, 2  |
|        |
|  3, 4  |
+-      -+
```

Note the following operation:

```
>> AA := A . A
```

```
+-            -+
|  1, 2, 1, 2  |
|              |
|  3, 4, 3, 4  |
+-            -+
```

returns a matrix of a different domain type as the input matrix:

```
>> domtype(AA)
```

```
Dom::Matrix(Dom::Rational)
```

`linalg::crossProduct` – cross product of three-dimensional vectors

`linalg::crossProduct(u, v)` computes the cross product of the three-dimensional vectors $\vec{u}$ and $\vec{v}$. This is the vector

$$\vec{u} \times \vec{v} = \left(\begin{array}{c} u_2v_3 - u_3v_2 \\ u_3v_1 - u_1v_3 \\ u_1v_2 - u_2v_1 \end{array} \right).$$

Call(s):

- `linalg::crossProduct(u, v)`

Parameters:

`u, v` — 3-dimensional vectors, i.e., either two 3×1 or two 1×3 matrices of a domain of category `Cat::Matrix`

Return Value: a vector of the same domain type as `u`.

Related Functions: `linalg::scalarProduct`

Details:

- The vectors must be defined over the same component ring.

Example 1. We define two vectors:

```
>> a := matrix([[1, 2, 3]]); b := matrix([[-1, 0, 1]])

                           +-         -+
                           | 1, 2, 3 |
                           +-         -+

                          +-          -+
                          | -1, 0, 1 |
                          +-          -+
```

The cross product of these two vectors is a vector $\vec{c}$ which is orthogonal to $\vec{a}$ and $\vec{b}$:

```
>> c:= linalg::crossProduct(a, b)

                          +-          -+
                          | 2, -4, 2 |
                          +-          -+
```

```
>> linalg::scalarProduct(a, c), linalg::scalarProduct(b, c)

                             0, 0
```

`linalg::curl` – curl of a vector field

`linalg::curl(v, x)` computes the curl of the three-dimensional vector field $\vec{v}$ with respect to the three-dimensional vector $\vec{x}$ in Cartesian coordinates. This is the vector field

$$\operatorname{curl}(\vec{v}) = \left(\begin{array}{c} \frac{\partial}{\partial x_2} v_3 - \frac{\partial}{\partial x_3} v_2 \\ \frac{\partial}{\partial x_3} v_1 - \frac{\partial}{\partial x_1} v_3 \\ \frac{\partial}{\partial x_1} v_2 - \frac{\partial}{\partial x_2} v_1 \end{array} \right).$$

Call(s):

- `linalg::curl(v, x)`
- `linalg::curl(v, x, ogCoord)`

Parameters:

`v` — a list of three arithmetical expressions, or a 3-dimensional vector (i.e., a 3×1 or 1×3 matrix of a domain of category `Cat::Matrix`)

`x` — a list of three (indexed) identifiers

`ogCoord` — a list, or a name (identifier) of a predefined coordinate system

Return Value: a column vector.

Related Functions: `linalg::divergence`, `linalg::grad`, `linalg::ogCoordTab`

Details:

- `linalg::curl(v, x, ogCoord)` computes the curl of `v` with respect to `x` in the orthogonally curvilinear coordinate system specified by `ogCoord`.

  The scaling factors of the specified coordinate system must be the value of the index `ogCoord` of the table `linalg::ogCoordTab` (see example 2).

- If `ogCoord` is given as a list then the curl of `v` is computed in the orthogonal curvilinear coordinates, whose scaling factors are given in `ogCoord` (see example 3).

- If `v` is a vector then the component ring of `v` must be a field (i.e., a domain of category `Cat::Field`) for which differentiation with respect to `x` is defined.

- `linalg::curl` returns a vector of the domain `Dom::Matrix()` if `v` is given as a list of arithmetical expressions.

Example 1. We compute the curl of the vector field $\vec{v}(x, y, z) = \big(xy, 2y, z\big)$ in Cartesian coordinates:

```
>> delete x, y, z:
   linalg::curl([x*y, 2*y, z], [x, y, z])

                                     +-     -+
                                     |   0   |
                                     |       |
                                     |   0   |
                                     |       |
                                     |  -x   |
                                     +-     -+
```

Example 2. We compute the curl of the vector field $\vec{v}(r, \phi, z) = (r, \cos(\phi), z)$ $(0 \leq \phi \leq 2\pi)$ in cylindrical coordinates:

```
>> delete r, phi, z: V := matrix([r, cos(phi), z]):

>> linalg::curl(V, [r, phi, z], Cylindrical)

                                  +-           -+
                                  |      0      |
                                  |             |
                                  |      0      |
                                  |             |
                                  |  cos(phi)   |
                                  |  --------   |
                                  |     r       |
                                  +-           -+
```

The following relations between Cartesian and cylindrical coordinates hold:

$$x = r\cos(\phi),\ y = r\sin(\phi),\ z = z.$$

Other predefined orthogonal coordinate systems can be found in the table `linalg::ogCoordTab`.

Example 3. We want to compute the curl of the vector field $\vec{v}(r, \theta, \phi) = \big(0, r^2, 0\big)$ $(0 \leq \theta \leq \pi, 0 \leq \theta \leq 2\pi)$ in spherical coordinates.

The vectors

$$\vec{e_r} = \begin{pmatrix} \sin\theta\cos\phi \\ \sin\theta\sin\phi \\ \cos\theta \end{pmatrix}, \vec{e_\theta} = \begin{pmatrix} \cos\theta\cos\phi \\ \cos\theta\sin\phi \\ -\sin\theta \end{pmatrix}, \vec{e_\phi} = \begin{pmatrix} -\sin\phi \\ \cos\phi \\ 0 \end{pmatrix}$$

form an orthogonal system in spherical coordinates.

The scaling factors of the corresponding coordinate transformation (see `linalg::ogCoordTab`) are: $g_1 = |\vec{e_r}| = 1, g_2 = |\vec{e_\theta}| = r, g_3 = |\vec{e_\phi}| = r\sin\theta$, which we use in the following example to compute the curl of the above vector field in spherical coordinates:

```
>> delete r, theta, phi:
   linalg::curl([0, r^2, 0], [r, theta, phi], [1, r, r*sin(theta)])

                              +-      -+
                              |   0    |
                              |        |
                              |   0    |
                              |        |
                              |  3 r   |
                              +-      -+
```

Note that the spherical coordinates are already defined in `linalg::ogCoordTab`, i.e., the last result can also be achieved with the input `linalg::curl([0, r^2, 0], [r, theta, phi], Spherical)`.

Changes:

- The result is a vector even if the vector field is given as a list of expressions.

`linalg::delCol` – delete matrix columns

`linalg::delCol(A, c)` returns a copy of the matrix A in which the column with index c is deleted.

Call(s):

- `linalg::delCol(A, c)`
- `linalg::delCol(A, c1..c2)`
- `linalg::delCol(A, list)`

Parameters:

`A` — an $m \times n$ matrix of a domain of category `Cat::Matrix`
`c` — the column index: a positive integer $\leq n$
`c1..c2` — a range of column indices (positive integers $\leq n$)
`list` — a list of column indices (positive integers $\leq n$)

Return Value: a matrix of a domain of category `Cat::Matrix(R)`, where `R` is the component ring of `A`, or the void object of type `DOM_NULL`.

Related Functions: `linalg::col`, `linalg::delRow`, `linalg::row`

Details:

- `linalg::delCol(A, c1..c2)` deletes those columns whose indices are in the range `c1..c2`. If `c2 < c1` then the input matrix `A` is returned.
- `linalg::delCol(A, list)` deletes those columns whose indices are contained in `list`.
- If all columns are deleted then the void object of type `DOM_NULL` is returned.

Example 1. We define the following matrix:

```
>> A := matrix([[1, 2, 3, 4], [5, 6, 7, 8]])

                        +-              -+
                        |  1, 2, 3, 4   |
                        |               |
                        |  5, 6, 7, 8   |
                        +-              -+
```

and demonstrate the three different input formats for `linalg::delCol`:

```
>> linalg::delCol(A, 2)

                          +-           -+
                          |  1, 3, 4   |
                          |            |
                          |  5, 7, 8   |
                          +-           -+
```

```
>> linalg::delCol(A, [1, 3])

                           +-        -+
                           |  2, 4   |
                           |         |
                           |  6, 8   |
                           +-        -+
>> linalg::delCol(A, 2..4)

                                    +-   -+
                                    |  1  |
                                    |     |
                                    |  5  |
                                    +-   -+
```

Example 2. We compute the inverse of the 2 × 2 matrix:

```
>> MatQ := Dom::Matrix(Dom::Rational):
   A := MatQ([[3, 2], [5, -4]])

                                  +-       -+
                                  |  3,  2  |
                                  |         |
                                  |  5, -4  |
                                  +-       -+
```

by appending the 2 × 2 identity matrix to the right side of A and applying the Gauss-Jordan algorithm provided by the function `linalg::gaussJordan`:

```
>> B := linalg::gaussJordan(A . MatQ::identity(2))

                            +-                     -+
                            |  1, 0, 2/11,  1/11   |
                            |                       |
                            |  0, 1, 5/22, -3/22   |
                            +-                     -+
```

We get the inverse of `A` by deleting the first two columns of the matrix `B`:

```
>> AI := linalg::delCol(B, 1..2)

                               +-               -+
                               |  2/11,  1/11   |
                               |                 |
                               |  5/22, -3/22   |
                               +-               -+
```

Finally, we check the result:

```
>> A * AI, AI * A

                            +-      -+  +-      -+
                            |  1, 0  |  |  1, 0  |
                            |        |, |        |
                            |  0, 1  |  |  0, 1  |
                            +-      -+  +-      -+
```

Note: The inverse of `A` can be computed directly by entering `1/A`.

Changes:

- If all rows are deleted then the void object of type `DOM_NULL` instead of the object `NIL` is returned.

`linalg::delRow` – delete matrix rows

`linalg::delRow(A, r)` returns a copy of the matrix A in which the row with index r is deleted.

Call(s):

- `linalg::delRow(A, r)`
- `linalg::delRow(A, r1..r2)`
- `linalg::delRow(A, list)`

Parameters:

`A`	— an $m \times n$ matrix of a domain of category `Cat::Matrix`
`r`	— the row index: a positive integer $\le m$
`r1..r2`	— a range of row indices (positive integers $\le m$)
`list`	— a list of row indices (positive integers $\le m$)

Return Value: a matrix of a domain of category `Cat::Matrix(R)`, where `R` is the component ring of `A`, or the void object of type `DOM_NULL`.

Related Functions: `linalg::col`, `linalg::delCol`, `linalg::row`

Details:

- `linalg::delRow(A, r1..r2)` deletes those rows whose indices are in the range `r1..r2`. If `r2 < r1` then the input matrix `A` is returned.
- `linalg::delRow(A, list)` deletes those rows whose indices are contained in `list`.
- If all rows are deleted then the void object of type `DOM_NULL` is returned.

Example 1. We define the following matrix:

```
>> A := matrix([[1, 2], [3, 4], [5, 6], [7, 8]])
```

```
+-       -+
|  1, 2   |
|         |
|  3, 4   |
|         |
|  5, 6   |
|         |
|  7, 8   |
+-       -+
```

and illustrate the three different input formats for `linalg::delRow`:

```
>> linalg::delRow(A, 2)

                                  +-       -+
                                  |  1, 2   |
                                  |         |
                                  |  5, 6   |
                                  |         |
                                  |  7, 8   |
                                  +-       -+

>> linalg::delRow(A, [1, 4])

                                  +-       -+
                                  |  3, 4   |
                                  |         |
                                  |  5, 6   |
                                  +-       -+

>> linalg::delRow(A, 2..4)

                                   +-     -+
                                   | 1, 2 |
                                   +-     -+
```

Changes:

- If all rows are deleted then the void object of type `DOM_NULL` instead of the object `NIL` is returned.

`linalg::det` – determinant of a matrix

`linalg::det(A)` computes the determinant of the square matrix *A*.

Call(s):

- `linalg::det(A)`

Parameters:

`A` — a square matrix of a domain of category `Cat::Matrix`.

Return Value: an element of the component ring of `A`.

Related Functions: `linalg::gaussElim`, `linalg::permanent`, `linalg::rank`, `numeric::det`

Details:

- A floating-point approximation of the determinant is computed with `numeric::det`, if `A` is defined over the component ring `Dom::Float`. In this case it is recommended to call `numeric::det` directly for a better efficiency.
- The component ring of `A` must be a commutative ring, i.e., a domain of category `Cat::CommutativeRing`.

Example 1. We compute the determinant of the following matrix:

```
>> A := matrix([[a11, a12], [a21, a22]])

                          +-            -+
                          |  a11, a12   |
                          |             |
                          |  a21, a22   |
                          +-            -+
```

which gives us the well-known formula for the determinant of an arbitrary 2 × 2 matrix:

```
>> linalg::det(A)

                          a11 a22 - a12 a21
```

Background:

- For an $n \times n$ matrix $A = \left(a_{ij}\right)_{1 \le i,j \le n}$ over a commutative ring its determinant is defined as:

$$\det(A) := \sum_{\sigma \in S_n} \operatorname{sign}(\sigma) \prod_{j=1}^{n} a_{\sigma(j),j}.$$

(S_n is the symmetric group of all permutations of $\{1, \ldots, n\}$.)

- For a component ring of A that is an integral domain (i.e., a domain of category `Cat::IntegralDomain`) and not defined over the domain `Dom::Float`, Gaussian elimination is used to compute the determinant of A.

  For any other commutative ring that is not an integral domain, a modification of the Berkowitz algorithm is used. Reference: A. Jounaidi: *The Berkowitz Algorithm, Maple and Computing the Characteristic Polynomial in an Arbitrary Commutative Ring*. Equipe de Mathèmatiques de Besancon, Universitè de Franche - Comtè, 25030 Besancon Cedex, May 1996.

Changes:

- Uses `numeric::det` for a floating-point approximation of the determinant.

`linalg::divergence` – divergence of a vector field

`linalg::divergence(v, x ...)` computes the divergence of the vector field $\vec{v}$ with respect to $\vec{x}$ in Cartesian coordinates. This is the sum $\mathrm{div}(\vec{v}) = \sum_{i=1}^{n} \frac{\partial}{\partial x_i} \vec{v}$.

Call(s):

- `linalg::divergence(v, x)`
- `linalg::divergence(v, x, ogCoord)`

Parameters:

`v`	—	a list of arithmetical expressions, or a vector (i.e., an $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`)
`x`	—	a list of (indexed) identifiers
`ogCoord`	—	a list, or a name (identifier) of a predefined coordinate system

Return Value: an arithmetical expression, or an element of the component ring of `v`.

Related Functions: `linalg::curl`, `linalg::grad`, `linalg::ogCoordTab`

Details:

- In the case of three dimensions, `linalg::divergence(v, x, ogCoord)` computes the divergence of the vector field `v` with respect to `x` in the orthogonally curvilinear coordinate system. The scaling factors of the specified coordinate system must be the value of the index `ogCoord` of the table `linalg::ogCoordTab` (see example 2).

- If `ogCoord` is given as a list then the divergence of `v` is computed in the orthogonal curvilinear coordinates, whose scaling factors are given in `ogCoord` (see example 3).
- If `v` is a vector then the component ring of `v` must be a field (i.e., a domain of category `Cat::Field`) for which differentiation with respect to `x` is defined.

Example 1. We compute the divergence of the vector field $\vec{v}(x, y, z) = \left(x^2, 2y, z\right)$ in Cartesian coordinates:

```
>> delete x, y, z:
   v := matrix([x^2, 2*y, z])

                            +-      -+
                            |    2   |
                            |   x    |
                            |        |
                            |  2 y   |
                            |        |
                            |   z    |
                            +-      -+
```

```
>> linalg::divergence(v, [x, y, z])

                             2 x + 3
```

Example 2. We compute the divergence of the vector field $\vec{v}(r, \phi, z) = (r, \cos(\phi), z)$ $(0 \leq \theta \leq 2\pi)$ in cylindrical coordinates:

```
>> delete r, phi, z:
   linalg::divergence([r, sin(phi), z], [r, phi, z], Cylindrical)

                         3 r + cos(phi)
                         --------------
                               r
```

The following relations between Cartesian and cylindrical coordinates hold:

$$x = r\cos(\phi),\ y = r\sin(\phi),\ z = z.$$

Other predefined orthogonal coordinate systems can be found in the table `linalg::ogCoordTab`.

Example 3. We want to compute the divergence of the vector field $\vec{v}(r, \theta, \phi) = \left(r^2, 0, 0\right)$ $(0 \leq \theta \leq \pi, 0 \leq \theta \leq 2\pi)$ in spherical coordinates.

The vectors

$$\vec{e}_r = \begin{pmatrix} \sin\theta\cos\phi \\ \sin\theta\sin\phi \\ \cos\theta \end{pmatrix}, \vec{e}_\theta = \begin{pmatrix} \cos\theta\cos\phi \\ \cos\theta\sin\phi \\ -\sin\theta \end{pmatrix}, \vec{e}_\phi = \begin{pmatrix} -\sin\phi \\ \cos\phi \\ 0 \end{pmatrix}$$

form an orthogonal system in spherical coordinates.

The scaling factors of the corresponding coordinate transformation (see `linalg::ogCoordTab`) are: $g_1 = |\vec{e}_r| = 1, g_2 = |\vec{e}_\theta| = r, g_3 = |\vec{e}_\phi| = r\sin\theta$, which we use in the following example to compute the divergence of the above vector field in spherical coordinates:

```
>> delete r, theta, phi:
   linalg::divergence(
     [r^2, 0, 0], [r, theta, phi], [1, r, r*sin(theta)]
   )

                                    4 r
```

Note that the spherical coordinates are already defined in `linalg::ogCoordTab`, i.e., the last result can also be achieved with the input `linalg::divergence([r^2, 0, 0], [r, theta, phi], Spherical)`.

Changes:

- The result is a vector even if the vector field is given as a list of expressions.

`linalg::eigenvalues` – eigenvalues of a matrix

`linalg::eigenvalues(A)` returns a list of the eigenvalues of the matrix A.

Call(s):

- `linalg::eigenvalues(A<, Multiple>)`

Parameters:

A — a square matrix of a domain of category `Cat::Matrix`

Options:

Multiple — In addition, the algebraic multiplicity of each eigenvalue of A is returned.

Return Value: a set of the eigenvalues of `A`, or a list of inner lists when the option *`Multiple`* is given (see below).

Related Functions: `numeric::eigenvalues`, `linalg::charpoly`, `linalg::eigenvectors`, `solve`

Details:

- A floating-point approximation of the eigenvalues is computed with `numeric::eigenvalues`, if the matrix `A` is defined over the component ring `Dom::Float` (see example 1). In this case it is recommended to call `numeric::eigenvalues` directly for a better efficiency.
- The eigenvalues are obtained by computing the zeros of the characteristic polynomial of `A`. The solver `solve` must be able to compute the roots of the characteristic polynomial over the component ring of `A`.

Option <*`Multiple`*>:

- Returns a list of sublists, where each sublist contains an eigenvalue of `A` and its algebraic multiplicity. Note that due to rounding errors, this may lead to wrong results in cases where multiple eigenvalues exist and `numeric::eigenvalues` is used.

Example 1. We compute the eigenvalues of the matrix

$$A = \begin{pmatrix} 1 & 4 & 2 \\ 1 & 4 & 2 \\ 2 & 5 & 3 \end{pmatrix}:$$

```
>> A := matrix([[1, 4, 2], [1, 4, 2], [2, 5, 3]]):
   linalg::eigenvalues(A)

                          1/2              1/2
                   {0, 15     + 4, 4 - 15     }
```

If we consider the matrix over the domain `Dom::Float`, then the call of `linalg::eigenvalues(A)` results in a numerical computation of the eigenvalues of `A` via `numeric::eigenvalues`:

```
>> B := Dom::Matrix(Dom::Float)(A): linalg::eigenvalues(B)

           {9.622294281e-19, 0.1270166538, 7.872983346}
```

Example 2. With the option *`Multiple`* we get the information about the algebraic multiplicity of each eigenvalue:

```
>> C := Dom::Matrix(Dom::Rational)(4, 4, [[-3], [0, 6]])

                        +-              -+
                        |  -3, 0, 0, 0   |
                        |                |
                        |   0, 6, 0, 0   |
                        |                |
                        |   0, 0, 0, 0   |
                        |                |
                        |   0, 0, 0, 0   |
                        +-              -+
```

```
>> linalg::eigenvalues(C, Multiple)

                    [[6, 1], [0, 2], [-3, 1]]
```

Changes:

- `linalg::eigenvalues` used to be `linalg::eigenValues`.
- Uses `numeric::eigenvalues` for a floating-point approximation of the eigenvalues.

`linalg::eigenvectors` – eigenvectors of a matrix

`linalg::eigenvectors(A)` computes the eigenvalues and eigenvectors of the matrix A.

Call(s):

- `linalg::eigenvectors(A)`

Parameters:

`A` — a square matrix of a domain of category `Cat::Matrix`

Return Value: a list of sublists, where each sublist consists of an eigenvalue λ of `A`, its algebraic multiplicity and a basis for the eigenspace of λ. If a basis of an eigenspace cannot be computed, then `FAIL` is returned.

Related Functions: `numeric::eigenvectors`, `linalg::eigenvalues`, `linalg::nullspace`

Details:

- A floating-point approximation of the eigenvalues and the eigenvectors is computed using `numeric::eigenvectors`, if the matrix `A` is defined over the component ring `Dom::Float` (see example 1). In this case it is recommended to call `numeric::eigenvalues` directly for a better efficiency.
- `linalg::eigenvectors` works as follows: For each eigenvalue λ of the $n \times n$ matrix `A` a basis for the kernel of $(\lambda I_n - A)$, the eigenspace of `A` with respect to the eigenvalue λ, is computed using the Gauss-Jordan algorithm (see `linalg::gaussJordan`). Here, I_n denotes the $n \times n$ identity matrix.
- The eigenvectors are of the domain `Dom::Matrix(R)`, where `R` is the component ring of `A`.
- The component ring of the matrix `A` must be a field, i.e., a domain of category `Cat::Field`, for which the solver `solve` is able to compute the zeros of a polynomial.
- It can happen that a basis for the eigenspace of `A` with respect to a certain eigenvalue cannot be computed (e.g., if the component ring does not have a canonical representation of the zero element). In this case `linalg::eigenvectors` answers with a warning message and returns `FAIL`.

Example 1. We compute the eigenvalues and the eigenvectors of the matrix

$$A = \begin{pmatrix} 1 & -3 & 3 \\ 6 & -10 & 6 \\ 6 & 6 & 4 \end{pmatrix} :$$

```
>> A := Dom::Matrix(Dom::Rational)(
     [[1, -3, 3], [6, -10, 6], [6, 6, 4]]
   ):
   linalg::eigenvectors(A)

 -- --       -- +-      -+ -- --  --        -- +-    -+ --
--
 |  |        |  |   1/4  |  |  |  |         |  |  -1  |  |  |
 |  |        |  |        |  |  |  |         |  |      |  |  |
 |  |  8, 1, |  |  5/12  |  |  |, |  -2, 1, |  |   0  |  |  |,
 |  |        |  |        |  |  |  |         |  |      |  |  |
 |  |        |  |    1   |  |  |  |         |  |   1  |  |  |
 -- --       -- +-      -+ -- --  --        -- +-    -+ --
--
```

```
    --          -- +-       -+ -- -- --
    |          |  |  -7/10  |  |  |  |
    |          |  |         |  |  |  |
    |  -11, 1, |  |   -9/5  |  |  |  |
    |          |  |         |  |  |  |
    |          |  |    1    |  |  |  |
    --          -- +-       -+ -- -- --
```

If we consider the matrix over the domain `Dom::Float` then the call of `linalg::eigenvectors(A)` results in a numerical computation of the eigenvalues and the eigenvectors of `A` via the function `numeric::eigenvectors`:

```
>> B := Dom::Matrix(Dom::Float)(A):
   linalg::eigenvectors(B)

 -- --          -- +-                -+ -- --
 |  |            |  |  -0.3218603429  |  |  |
 |  |            |  |                 |  |  |
 |  |  -11.0, 1, |  |  -0.8276408818  |  |  |,
 |  |            |  |                 |  |  |
 |  |            |  |   0.4598004899  |  |  |
 -- --          -- +-                -+ -- --

    --          -- +-                  -+ -- --
    |           |  |    -0.7071067812   |  |  |
    |           |  |                    |  |  |
    |  -2.0, 1, |  |  -1.518743801e-14  |  |  |,
    |           |  |                    |  |  |
    |           |  |    0.7071067812    |  |  |
    --          -- +-                  -+ -- --

    --          -- +-              -+ -- -- --
    |          |  |  0.2248595067  |  |  |  |
    |          |  |                |  |  |  |
    |  8.0, 1, |  |  0.3747658445  |  |  |  |
    |          |  |                |  |  |  |
    |          |  |  0.8994380268  |  |  |  |
    --          -- +-              -+ -- -- --
```

Changes:

- `linalg::eigenvectors` used to be `linalg::eigenVectors`.
- Uses `numeric::eigenvectors` for a floating-point approximation for the eigenvalues and eigenvectors.

`linalg::expr2Matrix` – **construct a matrix from equations**

`linalg::expr2Matrix(eqns, vars)` constructs the extended coefficient matrix $(A, \vec{b})$ of the system of m linear equations in `eqns` with respect to the n indeterminates in `vars`. The vector $\vec{b}$ is the right-hand side of this system.

Call(s):

- `linalg::expr2Matrix(eqns<, vars, R>)`
- `linalg::expr2Matrix(eqns<, vars, R>, `*`Include`*`)`

Parameters:

`eqns`	—	the system of linear equations, i.e. a set or list of expressions of type `"_equal"`
`vars`	—	a set or list of indeterminates
`R`	—	a commutative ring, i.e., a domain of category `Cat::CommutativeRing`

Options:

`Include`	—	Appends the negative of the right-hand side vector $\vec{b}$ to the coefficient matrix A of the given system of linear equations. The result is the $m \times (n+1)$ matrix $(A, -\vec{b})$.

Return Value: an $m \times (n+1)$ matrix of the domain `Dom::Matrix(R)`.

Related Functions: `linalg::matlinsolve`, `linsolve`, `indets`

Details:

- `linalg::expr2Matrix` returns the extended coefficient matrix $M = (A, \vec{b})$. The right-hand side vector $\vec{b}$ can be extracted from the matrix M by `linalg::col(M, n + 1)`.
- The coefficient matrix A can be extracted by `linalg::delCol(M, n + 1)`.
- Arithmetical expressions in `eqns` are considered as equations with right hand-sides zero.
- If no variables are given, then the indeterminates of the equations are determined with the function `indets` and the option `PolyExpr`, i.e., the left-hand sides of the equations are considered as polynomial expressions.

- If no component ring R is given then the standard domain `Dom::ExpressionField` is chosen as the component ring of the extended coefficient matrix.
- The coefficients of the linear equations are converted into elements of the component ring R. An error message is returned if this is not possible.

Example 1. The extended coefficient matrix of the system $x + y + z = 1, 2y - z + 5 = 0$ of linear equations in the variables x, y, z is the following 2×4 matrix:

```
>> delete x, y, z:
   Ab := linalg::expr2Matrix(
     [x + y + z = 1, 2*y - z + 5], [x, y, z], Dom::Real
   )

                     +-               -+
                     |  1, 1,  1,  1  |
                     |                 |
                     |  0, 2, -1, -5  |
                     +-               -+
```

We use `linalg::matlinsolve` to compute the general solution of this system:

```
>> linalg::matlinsolve(Ab)

              -- +-      -+  -- +-       -+ -- --
              |  |  7/2  |  |  |  -3/2  |  |  |
              |  |        |  |  |         |  |  |
              |  | -5/2  |, |  |   1/2  |  |  |
              |  |        |  |  |         |  |  |
              |  |   0   |  |  |    1    |  |  |
              -- +-      -+  -- +-       -+ -- --
```

The coefficient matrix or the right-hand side vector can be be extracted from the matrix `Ab` in the following way:

```
>> A := linalg::delCol(Ab, 4); b := linalg::col(Ab, 4)

                        +-           -+
                        |  1, 1,  1  |
                        |             |
                        |  0, 2, -1  |
                        +-           -+

                           +-     -+
                           |   1  |
                           |       |
                           |  -5  |
                           +-     -+
```

Example 2. The following two inputs lead to different linear systems:

```
>> delete x, y, z:
   linalg::expr2Matrix([x + y + z = 1, 2*y - z + 5 = x]),
   linalg::expr2Matrix([x + y + z = 1, 2*y - z + 5 = x], [x, y])

               +-                -+  +-                 -+
               |   1, 1,  1,  1  |  |   1, 1, - z + 1  |
               |                 |, |                  |
               |  -1, 2, -1, -5  |  |  -1, 2,  z - 5   |
               +-                -+  +-                 -+
```

Example 3. Note the difference between calling `linalg::expr2Matrix` with and without option *`Include`*:

```
>> delete x, y:
   linalg::expr2Matrix([x + y = 1, 2*x - y = 3], [x, y])

                            +-          -+
                            |  1,  1, 1  |
                            |            |
                            |  2, -1, 3  |
                            +-          -+

>> linalg::expr2Matrix([x + y = 1, 2*x - y = 3], [x, y], Include)

                            +-           -+
                            |  1,  1, -1  |
                            |             |
                            |  2, -1, -3  |
                            +-           -+
```

Changes:

- The result of `linalg::expr2Matrix` is the extended coefficient matrix $(A, \vec{b})$ of a linear system instead of a list of the coefficient matrix A and the right-hand side vector $\vec{b}$.
- Due to the changes of the return value the former option `Append` is obsolete and therefore no longer valid.

`linalg::factorCholesky` – the Cholesky decomposition of a matrix

`linalg::factorCholesky(A)` computes the Cholesky decomposition of a symmetric and positive definite matrix A and returns a lower triangular matrix R such that $RR^t = A$.

Call(s):

- `linalg::factorCholesky(A<, NoCheck>)`

Parameters:

`A` — a square matrix of a domain of category `Cat::Matrix`

Options:

`NoCheck` — It is not checked whether `A` is symmetric and positive definite.

Return Value: a matrix of the same domain type as `A`, or the value `FAIL`.

Side Effects: Properties of identifiers are taken into account.

Related Functions: `linalg::isHermitean`, `linalg::isPosDef`

Details:

- The system checks whether the matrix `A` is symmetric and positive definite, and returns an error message if this is not the case.
- The Option *`NoCheck`* suppresses such errors (see example 2).
- The component ring of `A` must be a field, i.e., a domain of category `Cat::Field`.
- `linalg::factorCholesky` returns `FAIL` if it fails to compute the matrix R over the component ring of `A` (the algorithm requires the computation of square roots of some elements in R).

Example 1. We compute the Cholesky decomposition of the following matrix:

```
>> S := Dom::Matrix(Dom::Rational)(
     [[4, -2, 4, 2], [-2, 10, -2, -7], [4, -2, 8, 4], [2, -
7, 4, 7]]
   )
```

```
                         +-                  -+
                         |   4, -2,  4,  2   |
                         |                   |
                         |  -2, 10, -2, -7   |
                         |                   |
                         |   4, -2,  8,  4   |
                         |                   |
                         |   2, -7,  4,  7   |
                         +-                  -+
```

```
>> R := linalg::factorCholesky(S)
```

```
                          +-               -+
                          |   2,  0, 0, 0   |
                          |                 |
                          |  -1,  3, 0, 0   |
                          |                 |
                          |   2,  0, 2, 0   |
                          |                 |
                          |   1, -2, 1, 1   |
                          +-               -+
```

and check the result:

```
>> R * linalg::transpose(R) = S
```

```
            +-                  -+   +-                  -+
            |   4, -2,  4,  2   |   |   4, -2,  4,  2   |
            |                   |   |                   |
            |  -2, 10, -2, -7   |   |  -2, 10, -2, -7   |
            |                   | = |                   |
            |   4, -2,  8,  4   |   |   4, -2,  8,  4   |
            |                   |   |                   |
            |   2, -7,  4,  7   |   |   2, -7,  4,  7   |
            +-                  -+   +-                  -+
```

Example 2. The option `NoCheck` can be helpful for matrices with symbolic components. For example, if we define the following matrix:

```
>> delete a, b:
   H := matrix([[a, b], [b, a]])
```

```
                              +-      -+
                              |  a, b  |
                              |        |
                              |  b, a  |
                              +-      -+
```

and have in mind that `a` and `b` are real, then `linalg::factorCholesky` is not able to check `H` to be positive definite:

```
>> linalg::factorCholesky(H)

 Error: cannot check whether matrix component is positive \
 [linalg::factorCholesky]
```

With the option *`NoCheck`* such errors are suppressed and `linalg::factorCholesky` continues the computation:

```
>> linalg::factorCholesky(H, NoCheck)

                    +-                     -+
                    |   1/2                 |
                    |  a   ,       0        |
                    |                       |
                    |        /      2 \1/2  |
                    |   b    |     b  |     |
                    |  ----, | a - -- |     |
                    |   1/2  \     a  /     |
                    |  a                    |
                    +-                     -+
```

Of course, this result is only valid if $a > 0$ and $|b| < a$.

Background:

- The Cholesky decomposition of a positive definite $n \times n$ matrix A is a decomposition of A in a product $\mathtt{A} = RR^t$ such that R is lower triangular and has positive (real) entries on the main diagonal.

  R is called the "Cholesky factor" of A.
- If $R = \left(r_{ij}\right)_{1 \le i,j \le n}$ is the Cholesky factor of A, then $\det(A) = \left(\prod_{i=1}^{n} r_{ii}\right)^2$.

Changes:

- `linalg::factorCholesky` used to be `linalg::cholesky`.
- The new option *`NoCheck`* was added.
- The option `isPositiveDefinite` is no longer available. Use the function `linalg::isPosDef` instead.

`linalg::factorLU` – LU-decomposition of a matrix

`linalg::factorLU(A)` computes an LU-decomposition of an $m \times n$ matrix A, i.e., a decomposition of the A into an $m \times m$ lower triangular matrix L and an $n \times m$ upper triangular matrix U such that $PA = LU$, where P is a permutation matrix.

Call(s):

- `linalg::factorLU(A)`

Parameters:

`A` — a matrix of a domain of category `Cat::Matrix`

Return Value: a list `[L, U, pivindex]` with the two matrices L and U of the domain `Dom::Matrix(R)` and a list `pivindex` of positive integers. `R` is the component ring of `A`.

Related Functions: `linalg::factorQR`, `linalg::factorCholesky`, `linalg::inverseLU`, `linalg::matlinsolveLU`, `lllint`, `numeric::factorLU`

Details:

- The diagonal entries of the lower triangular matrix L are equal to one (*Doolittle*-decomposition). The diagonal entries of U are the pivot elements used during the computation.

  The matrices L and U are unique.
- `pivindex` is a list `[r[1], r[2], ...]` representing the row exchanges of A in the pivoting steps, i.e., $B = PA = LU$, where $b_{ij} = a_{r[i],j}$.
- A floating-point approximation of the decomposition is computed using `numeric::factorLU`, if the matrix `A` is defined over the component ring `Dom::Float`. In this case it is recommended to call `numeric::factorLU` directly for a better efficiency.
- The algorithm also works for singular A. In this case either L or U is singular.
- L and U are nonsingular if and only if A is nonsingular.
- The component ring of the matrix `A` must be a field, i.e., a domain of category `Cat::Field`.

Example 1. We compute an LU-decomposition of the real matrix:

```
>> A := Dom::Matrix(Dom::Real)(
     [[2, -3, -1], [1, 1, -1], [0, 1, -1]]
   )

                         +-           -+
                         |  2, -3, -1  |
                         |             |
                         |  1,  1, -1  |
                         |             |
                         |  0,  1, -1  |
                         +-           -+
```

```
>> [L, U, pivlist] := linalg::factorLU(A)

     -- +-             -+  +-              -+            -
-
     |  |   1,   0,  0  |  |  2, -3,   -1   |             |
     |  |               |  |                |             |
     |  |  1/2,  1,  0  |, |  0, 5/2, -1/2  |, [1, 2, 3]  |
     |  |               |  |                |             |
     |  |   0,  2/5, 1  |  |  0,  0,  -4/5  |             |
     -- +-             -+  +-              -+            -
-
```

The lower triangular matrix *L* is the first element und the upper triangular matrix *U* is the second element of the list `LU`. The product of these two matrices is equal to the input matrix `A`:

```
>> L * U

                         +-           -+
                         |  2, -3, -1  |
                         |             |
                         |  1,  1, -1  |
                         |             |
                         |  0,  1, -1  |
                         +-           -+
```

Example 2. An LU-decomposition of the 3×2 matrix:

```
>> A := Dom::Matrix(Dom::Real)([[2, -3], [1, 2], [2, 3]])
```

```
                        +-        -+
                        |  2, -3   |
                        |          |
                        |  1,  2   |
                        |          |
                        |  2,  3   |
                        +-        -+
```

gives a 3 × 3 lower triangular matrix and a 2 × 3 upper triangular matrix:

```
>> [L, U, pivlist] := linalg::factorLU(A)

        -- +-                -+  +-         -+             --
        |  |   1,     0,  0  |  |  2,   -3  |              |
        |  |                 |  |           |              |
        |  |  1/2,    1,  0  |, |  0, 7/2   |, [1, 2, 3]   |
        |  |                 |  |           |              |
        |  |   1,  12/7, 1   |  |  0,   0   |              |
        -- +-                -+  +-         -+             --

>> L * U

                        +-        -+
                        |  2, -3   |
                        |          |
                        |  1,  2   |
                        |          |
                        |  2,  3   |
                        +-        -+
```

Example 3. To compute the LU-decomposition of the matrix:

```
>> A := matrix([[1, 2, -1], [0, 0, 3], [0, 2, -1]])

                      +-           -+
                      |  1, 2, -1   |
                      |             |
                      |  0, 0,  3   |
                      |             |
                      |  0, 2, -1   |
                      +-           -+
```

one row interchange is needed, and we therefore get a non-trivial permutation list:

```
>> [L, U, pivlist] := linalg::factorLU(A)
```

```
-- +-         -+  +-           -+              --
|  |  1, 0, 0  |  |  1, 2, -1  |               |
|  |           |  |            |               |
|  |  0, 1, 0  |, |  0, 2, -1  |, [1, 3, 2]    |
|  |           |  |            |               |
|  |  0, 0, 1  |  |  0, 0,  3  |               |
-- +-         -+  +-           -+              --
```

The corresponding permutation matrix is the following:

```
>> P := linalg::swapRow(matrix::identity(3), 3, 2)
```

```
+-         -+
|  1, 0, 0  |
|           |
|  0, 0, 1  |
|           |
|  0, 1, 0  |
+-         -+
```

Hence, we have a decomposition of A into the product of the three matrices P^{-1}, L and U as follows:

```
>> P^(-1) * L * U
```

```
+-          -+
|  1, 2, -1  |
|            |
|  0, 0,  3  |
|            |
|  0, 2, -1  |
+-          -+
```

Example 4. You may compute an LU-decomposition of a matrix with symbolic components, such as:

```
>> delete a, b, c, d:
   A := matrix([[a, b], [c, d]])
```

```
+-      -+
|  a, b  |
|        |
|  c, d  |
+-      -+
```

The diagonal entries of the matrix U are the pivot elements used during the computation. They must be non-zero, if the inverse of U is needed:

```
>> [L, U, pivlist] := linalg::factorLU(A)

           -- +-      -+  +-            -+          --
           |  |  1, 0  |  |  a,    b     |          |
           |  |        |  |              |          |
           |  |  c     |, |       b c    |, [1, 2]  |
           |  |  -, 1  |  |  0, d - ---  |          |
           |  |  a     |  |        a     |          |
           -- +-      -+  +-            -+          --
```

For example, if we use this decomposition to solve the linear system $A\vec{x} = \vec{b}$ for arbitrary vectors $\vec{b} = (b_1, b_2)^t$, then the following result is only correct for $a \neq 0$ and $d - \frac{bc}{a} \neq 0$:

```
>> delete b1, b2:
   linalg::matlinsolveLU(L, U, matrix([b1, b2]))

                   +-                      -+
                   |         /      c b1 \  |
                   |       b | b2 - ---- |  |
                   |         \       a   /  |
                   |  b1 - ---------------  |
                   |               b c      |
                   |           d - ---      |
                   |                a       |
                   |  --------------------  |
                   |           a            |
                   |                        |
                   |             c b1       |
                   |        b2 - ----       |
                   |              a         |
                   |        ---------       |
                   |             b c        |
                   |         d - ---        |
                   |              a         |
                   +-                      -+
```

Background:

- The following algorithm for solving the system $A\vec{x} = \vec{b}$ with a nonsingular matrix A uses LU-decomposition:

    1. Compute a LU-decomposition of A: $A = LU$.
    2. Solve $\vec{y} = L^{-1}\vec{b}$ by forward substitution.
    3. Solve $\vec{x} = R^{-1}\vec{y}$ by backward substitution.

- The LU-decomposition of a matrix A is useful for solving several systems of linear equations $A\vec{x} = \vec{b}$ with the same coefficient matrix A and several right-hand side vectors $\vec{b}$, because then step one of the algorithm above needs to be done only once.

Changes:

- `linalg::factorLU` is a new function.

`linalg::factorQR` – QR-decomposition of a matrix

`linalg::factorQR(A)` computes an QR-decomposition of an $m \times n$ matrix A, i.e., a decomposition of A into an $n \times n$ unitary matrix Q and an $n \times m$ upper triangular matrix R such that $QR = A$.

Call(s):

- `linalg::factorQR(A)`

Parameters:

`A` — a matrix of a domain of category `Cat::Matrix`

Return Value: a list `[Q, R]` of the two matrices Q and R (of the same domain type as `A`), or the value `FAIL`.

Related Functions: `linalg::factorLU`, `linalg::factorCholesky`, `lllint`, `numeric::factorQR`

Details:

- `linalg::factorQR` uses Gram-Schmidt orthonormalization to compute the decomposition.
- For a singular or non-square matrix `A` the QR-decomposition of `A` is not unique.
- The columns of Q form an orthonormal basis with respect to the scalar product of two vectors, defined by `linalg::scalarProduct`, and the 2-norm of two vectors (see the method `"norm"` of the domain constructor `Dom::Matrix`).
- If the component ring of `A` does not define the method `"conjugate"`, then the factor Q is orthogonal instead of unitary.
- If the columns of `A` cannot be orthonormalized then `FAIL` is returned.

- If `A` is a matrix over the domain `Dom::Float` and the computations are based on the standard scalar product, then the use of the corresponding function from the numeric library (`numeric::factorQR`) is recommended.
- Even if `A` is defined over the real or the complex numbers the call of `numeric::factorQR` with the option `Symbolic` is recommended for better efficiency.
- The component ring of the matrix `A` must be a field, i.e., a domain of category `Cat::Field`.

Example 1. We compute the QR-decomposition of a real matrix:

```
>> A := Dom::Matrix(Dom::Real)(
     [[2, -3, -1], [1, 1, -1], [0, 1, -1]]
   )

                         +-           -+
                         |  2, -3, -1  |
                         |             |
                         |  1,  1, -1  |
                         |             |
                         |  0,  1, -1  |
                         +-           -+

>> QR := linalg::factorQR(A)

 -- +-                              -+
 |  |     1/2     1/2     1/2   1/2  |
 |  |  2 5       6       8    15     |
 |  |  ------, - ----, - ----------  |
 |  |    5        6          60      |
 |  |                                |
 |  |    1/2     1/2     1/2   1/2   |
 |  |   5       6       8    15      |
 |  |   ----,   ----,   ----------   |,
 |  |    5       3          30       |
 |  |                                |
 |  |            1/2      1/2   1/2  |
 |  |           6        8    15     |
 |  |     0,    ----,  - ----------  |
 |  |            6          12       |
 -- +-                              -+

    +-                          -+ --
    |                      1/2   |  |
    |   1/2     1/2     3 5      |  |
    |  5    , - 5    ,   - ------    |  |
    |                        5       |  |
    |                                |  |
    |                        1/2     |  |
    |            1/2       6         |  |
    |    0,    6    ,     - ----     |  |
    |                        3       |  |
    |                                |  |
    |                    1/2   1/2   |  |
    |                   8    15      |  |
    |    0,      0,    ----------    |  |
    |                       15       |  |
    +-                              -+ --
```

The orthogonal matrix Q is the first element und the upper triangular matrix R is the second element of the list `QR`. The product of these two matrices is equal to the input matrix `A`:

```
>> QR[1] * QR[2]

                                         +-           -+
                                         |  2, -3, -1  |
                                         |             |
                                         |  1,  1, -1  |
                                         |             |
                                         |  0,  1, -1  |
                                         +-           -+
```

Example 2. The QR-decomposition of the 3 × 2 matrix:

```
>> B := Dom::Matrix(Dom::Real)(
     [[2, -3], [1, 2], [2, 3]]
   )

                                           +-       -+
                                           |  2, -3  |
                                           |         |
                                           |  1,  2  |
                                           |         |
                                           |  2,  3  |
                                           +-       -+
```

yields a 3 × 3 orthogonal matrix and a 3 × 2 upper triangular matrix:

```
>> QR := linalg::factorQR(B)
```

```
    -- +-                              -+                   -
-
    |  |               1/2       1/2    |                   |
    |  |         31 194       194       |                   |
    |  |  2/3, - ---------,   ------    |  +-            -+ |
    |  |            582        194      |  |  3,   2/3    | |
    |  |                                |  |              | |
    |  |              1/2         1/2   |  |        1/2   | |
    |  |         8 194       6 194      |  |     194      | |
    |  |  1/3,   --------,   --------   |, |  0, ------   | |
    |  |           291          97      |  |      3       | |
    |  |                                |  |              | |
    |  |              1/2          1/2  |  |  0,    0     | |
    |  |        23 194        7 194     |  +-            -+ |
    |  |  2/3,  ---------,  - --------  |                   |
    |  |           582          194     |                   |
    -- +-                              -+                   -
-

>> QR[1] * QR[2]

                           +-       -+
                           |  2, -3  |
                           |         |
                           |  1,  2  |
                           |         |
                           |  2,  3  |
                           +-       -+
```

For this example we may call `numeric::factorQR(B, Symbolic)` instead, which in general is faster than `linalg::factorQR`:

```
>> QR := numeric::factorQR(B, Symbolic)

    -- +-                              -+                   -
-
    |  |               1/2       1/2    |                   |
    |  |         31 194       194       |                   |
    |  |  2/3, - ---------,   ------    |  +-            -+ |
    |  |            582        194      |  |  3,   2/3    | |
    |  |                                |  |              | |
    |  |              1/2         1/2   |  |        1/2   | |
    |  |         8 194       6 194      |  |     194      | |
    |  |  1/3,   --------,   --------   |, |  0, ------   | |
    |  |           291          97      |  |      3       | |
    |  |                                |  |              | |
    |  |              1/2          1/2  |  |  0,    0     | |
    |  |        23 194        7 194     |  +-            -+ |
    |  |  2/3,  ---------,  - --------   |                   |
    |  |           582          194      |                   |
    -- +-                               -+                  -
-
```

Background:

- The QR-decomposition can be used to generate a least square solution to an overdetermined system of linear equations. If $A\vec{x} = \vec{b}$, then $R\vec{x} = Q^t\vec{b}$ can be solved via backward substitution.

Changes:

- `linalg::factorQR` was extended to handle singular as well as non-square matrices.

`linalg::frobeniusForm` – Frobenius form of a matrix

`linalg::frobeniusForm(A)` returns the Frobenius form of the matrix A, also called the Rational Canonical form of A.

Call(s):

- `linalg::frobeniusForm(A<, All>)`

Parameters:

`A` — a square matrix of a domain of category `Cat::Matrix`

Options:

`All` — returns the list `[R, P]` with the Frobenius form R of `A` and a transformation matrix P such that $A = PRP^{-1}$.

Return Value: a matrix of the same domain type as `A`, or the list `[R, P]` when the option `All` is given.

Related Functions: `linalg::jordanForm`, `linalg::hermiteForm`, `linalg::smithForm`, `linalg::minpoly`

Details:

- `linalg::frobeniusForm(A, All)` computes the Frobenius form R of `A` and a transformation matrix P such that PRP^{-1}.

- The Frobenius form as computed by `linalg::frobeniusForm` is unique (see below).
- The component ring of `A` must be a field, i.e., a domain of category `Cat::Field`.

Example 1. The Frobenius form of the following matrix over ℂ:

```
>> A := Dom::Matrix(Dom::Complex)(
     [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
   )

                          +-         -+
                          |  1, 2, 3  |
                          |           |
                          |  4, 5, 6  |
                          |           |
                          |  7, 8, 9  |
                          +-         -+
```

is the matrix:

```
>> R := linalg::frobeniusForm(A)

                         +-          -+
                         |  0, 0,  0  |
                         |            |
                         |  1, 0, 18  |
                         |            |
                         |  0, 1, 15  |
                         +-          -+
```

The transformation matrix P can be selected from the list `[R, P]`, which is the result of `linalg::frobeniusForm` with option *All*:

```
>> P := linalg::frobeniusForm(A, All)[2]

                         +-            -+
                         |  1, 1,  30   |
                         |              |
                         |  0, 4,  66   |
                         |              |
                         |  0, 7, 102   |
                         +-            -+
```

We check the result:

```
>> P * R * P^(-1)
```

```
+-         -+
|  1, 2, 3  |
|           |
|  4, 5, 6  |
|           |
|  7, 8, 9  |
+-         -+
```

Background:

- The Frobenius form of a square matrix A is the matrix

$$R = \begin{pmatrix} R_1 & & & 0 \\ & \cdot & & \\ & & \cdot & \\ & & & \cdot \\ 0 & & & R_r \end{pmatrix},$$

  where $R_1, \ldots, R_r$ are known as companion matrices and have the form:

$$R_i = \begin{pmatrix} 0 & & & -a_0 \\ 1 & \cdot & & \\ & \cdot & \cdot & \\ & 1 & 0 & -a_{n_i-1} \end{pmatrix}, \quad i = 1, \ldots, r.$$

  In the last column of the companion matrix R_i, you see the coefficients of its minimal polynomial in ascending order, i.e., the polynomial $m_i := X^{n_i} + a_{n_i-1}X^{n_i-1} + \ldots + a_1 X + a_0$ is the minimal polynomial of the matrix R_i.

  For these polynomials the following holds: m_i divides m_{i+1} for $i = 1, \ldots, r-1$, and m_r is the minimal polynomial of A.

  The Frobenius form defined in this way is unique.

- Reference: P. Ozello: *Calcul exact des formes de Jordan et de Frobenius d'une matrice*, pp. 30–43. Thèse de l'Universite Scientifique Technologique et Medicale de Grenoble, 1987

Changes:

- `linalg::frobeniusForm` is a new function.

`linalg::gaussElim` – Gaussian elimination

`linalg::gaussElim(A)` performs Gaussian elimination on the matrix A to reduce A to an upper row echelon form.

Call(s):

- `linalg::gaussElim(A<, `*`All`*`>)`

Parameters:

`A` — a matrix of a domain of category `Cat::Matrix`

Options:

`All` — additionally returns the rank and the determinant of `A` (if `A` is a square) as well as the characteristic column indices of the matrix in row echelon form.

Return Value: a matrix of the same domain type as `A`, or the list `[T, rank(A), det(A), {j1,...,jr}]` when the option *`All`* is given (see below).

Related Functions: `linalg::gaussJordan`, `lllint`

Details:

- A row echelon form of `A` returned by `linalg::gaussElim` is not unique. See `linalg::gaussJordan` for computing the *reduced* row echelon form.
- The component ring R of `A` must be an integral domain, i.e., a domain of category `Cat::IntegralDomain`.
- If R is a field, i.e., a domain of category `Cat::Field`, ordinary Gaussian elimination is used. Otherwise, `linalg::gaussElim` applies fraction-free Gaussian elimination to `A`.
- Refer to the help page of `Dom::Matrix` for details about the computation strategy of `linalg::gaussElim`.

Option <`All`>:

- Returns a list $\left[T, \text{rank}(A), \det(A), \{j_1, \ldots, j_r\}\right]$ where T is a row echelon form of `A` and $\{j_1, \ldots, j_r\}$ is the set of characteristic column indices of `T`.

  If `A` is not square, then the value `FAIL` is given instead of $\det(A)$.
- `linalg::gaussElim` serves as an interface function for the method `"gaussElim"` of the matrix domain of `A`, i.e., one may call `A::dom::gaussElim(A)` directly instead of `linalg::gaussElim(A, All)`

Example 1. We apply Gaussian elimination to the following matrix:

```
>> A := Dom::Matrix(Dom::Rational)(
     [[1, 2, 3, 4], [-1, 0, 1, 0], [3, 5, 6, 9]]
   )

                      +-             -+
                      |   1, 2, 3, 4  |
                      |               |
                      |  -1, 0, 1, 0  |
                      |               |
                      |   3, 5, 6, 9  |
                      +-             -+
```

which reduces `A` to the following row echelon form:

```
>> linalg::gaussElim(A)

                     +-                -+
                     |  1, 2,  3,  4   |
                     |                  |
                     |  0, 2,  4,  4   |
                     |                  |
                     |  0, 0, -1, -1   |
                     +-                -+
```

Example 2. We apply Gaussian elimination to the matrix:

```
>> B := Dom::Matrix(Dom::Integer)(
     [[1, 2, -1], [1, 0, 1], [2, -1, 4]]
   )

                       +-            -+
                       |  1,  2, -1   |
                       |              |
                       |  1,  0,  1   |
                       |              |
                       |  2, -1,  4   |
                       +-            -+
```

and get the following result:

```
>> linalg::gaussElim(B, All)

             -- +-            -+                     --
             |  |  1,  2, -1   |                      |
             |  |              |                      |
             |  |  0, -2,  2   |, 3, -2, {1, 2, 3}   |
             |  |              |                      |
             |  |  0,  0, -2   |                      |
             -- +-            -+                     --
```

We see that $\operatorname{rank}(B) = 3$ and $\det(B) = -2$.

Background:

- Let $T = \left(t_{ij}\right)_{1 \le i \le m, 1 \le j \le n}$ be an $m \times n$ matrix. Then T is a matrix in an upper *row echelon form,* if $r \in \{0, 1, \ldots, n\}$ and indices $j_1, j_2, \ldots, j_r \in \{1, \ldots, n\}$ exist with:

  1. $j_1 < j_2 < \cdots < j_r$.
  2. For each $i \in \{1, \ldots, r\}$: $t_{i,1} = t_{i,2} = \cdots = t_{i,j_i - 1} = 0$.
  3. For each $i \in \{r + 1, \ldots, m\}$: $t_{ij} = 0$ for each $j \in \{1, \ldots, n\}$.

  The indices $j_1, j_2, \ldots, j_r$ are the *characteristic column indices* of the matrix T.

`linalg::gaussJordan` – Gauss-Jordan elimination

`linalg::gaussJordan(A)` performs Gauss-Jordan elimination on the matrix A, i.e., it returns the reduced row echelon form of A.

Call(s):

- `linalg::gaussJordan(A<, All>)`

Parameters:

`A` — a matrix of a domain of category `Cat::Matrix`

Options:

`All` — additionally returns the rank and the determinant of `A` (if `A` is a square) as well as the characteristic column indices of the matrix in reduced row echelon form.

Return Value: a matrix of the same domain type as `A`, or the list `[T, rank(A), det(A), j1,...,jr]` when the option *`All`* is given (see below).

Related Functions: `linalg::gaussElim`

Details:

- The component ring R of `A` must be an integral domain, i.e., a domain of category `Cat::IntegralDomain`.

- If R is a field, i.e., a domain of category `Cat::Field`, then the leading entries of the matrix T in reduced row echelon form are equal to one.

  If R is a ring providing the method `"gcd"`, then the components of each row of T do not have a non-trivial common divisor.

- If the component ring of `A` is a field, then the reduced row echelon form is unique.

Option <`All`>:

- Returns a list $\left[T, \mathrm{rank}(A), \det(A), \{j_1, \ldots, j_r\}\right]$ where T is the reduced row echelon form of `A` and $\{j_1, \ldots, j_r\}$ is the set of characteristic column indices of `T`.

  If `A` is not square, then the value `FAIL` is given instead of $\det(A)$.

Example 1. We apply Gauss-Jordan elimination to the following matrix:

```
>> A := Dom::Matrix(Dom::Rational)(
     [[1, 2, 3, 4], [-5, 0, 3, 0], [3, 5, 6, 9]]
   )

                                  +-            -+
                                  |   1, 2, 3, 4  |
                                  |               |
                                  |  -5, 0, 3, 0  |
                                  |               |
                                  |   3, 5, 6, 9  |
                                  +-            -+
```

```
>> linalg::gaussJordan(A, All)

            -- +-               -+                       --
            |  |  1, 0, 0, 1/2  |                        |
            |  |                |                        |
            |  |  0, 1, 0, 1/2  |, 3, FAIL, {1, 2, 3}   |
            |  |                |                        |
            |  |  0, 0, 1, 5/6  |                        |
            -- +-               -+                       --
```

We see that $\mathrm{rank}(B) = 3$. Because the determinant of a matrix is only defined for square matrices, the third element of the returned list is the value `FAIL`.

Example 2. If we consider the matrix from example 1 as an integer matrix and apply the Gauss-Jordan elimination we get the following matrix:

```
>> B := Dom::Matrix(Dom::Integer)(
     [[1, 2, 3, 4], [-5, 0, 3, 0], [3, 5, 6, 9]]
   ):
   linalg::gaussJordan(B)
```

```
                          +-                 -+
                          |  2,  0,  0,  1  |
                          |                 |
                          |  0, -2,  0, -1  |
                          |                 |
                          |  0,  0, -6, -5  |
                          +-                 -+
```

Background:

- Let $T = \left(t_{ij}\right)_{1 \le i \le m, 1 \le j \le n}$ be an $m \times n$ matrix. Then T is a matrix in *reduced row echelon form,* if $r \in \{0, 1, \dots, n\}$ and indices $j_1, j_2, \dots, j_r \in \{1, \dots, n\}$ exist with:

  1. $j_1 < j_2 < \cdots < j_r$.
  2. For each $i \in \{1, \dots, r\}$: $t_{i,1} = t_{i,2} = \cdots = t_{i,j_i - 1} = 0$. In addition, if A is defined over a field: $t_{i,j_i} = 1$.
  3. For each $i \in \{r+1, \dots, m\}$: $t_{ij} = 0$ for each $j \in \{1, \dots, n\}$.
  4. For each $i \in \{1, \dots, r\}$: $t_{k,j_i} = 0$ for each $k \in \{1, \dots, i-1\}$.

  The indices $j_1, j_2, \dots, j_r$ are the *characteristic column indices* of the matrix T.

`linalg::grad` – vector gradient

`linalg::grad(f, x)` computes the vector gradient of the scalar function $f(\vec{x})$ with respect to $\vec{x}$ in Cartesian coordinates. This is the vector $\mathrm{grad}(f) = (\frac{\partial}{\partial x_1} f, \dots, \frac{\partial}{\partial x_n} f)$.

Call(s):

- `linalg::grad(f, x)`
- `linalg::grad(f, x, ogCoord)`

Parameters:

`f`	—	an arithmetical expression in the variables given in `x`
`x`	—	a list of (indexed) identifiers
`ogCoord`	—	a list, or a name (identifier) of a predefined coordinate system

Return Value: a column vector of the domain `Dom::Matrix()`.

Related Functions: `linalg::curl`, `linalg::divergence`, `linalg::ogCoordTab`, `linalg::vectorPotential`

Details:

- In the case of three dimensions, `linalg::grad(f, x, ogCoord)` computes the gradient of `f` with respect to `x` in the orthogonally curvilinear coordinate system specified by `ogCoord`. The scaling factors of the specified coordinate system must be the value of the index `ogCoord` of the table `linalg::ogCoordTab` (see example 2).
- If `ogCoord` is an identifier then the scaling factors must be defined under the name of the identifier as an entry of the table `linalg::ogCoordTab`.

Example 1. We compute the vector gradient of the scalar function $f(x, y) = x^2 + y$ in Cartesian coordinates:

```
>> delete x, y:
   linalg::grad(x^2 + y, [x, y])

                                  +-      -+
                                  |  2 x   |
                                  |        |
                                  |   1    |
                                  +-      -+
```

Example 2. We compute the gradient of the function $f(r, \phi, z) = r\cos(\phi)z$ $(0 \leq \phi \leq \pi)$ in cylindrical coordinates:

```
>> delete r, z, phi:
   linalg::grad(r*cos(phi)*z, [r, phi, z], Cylindrical)

                              +-              -+
                              |   z cos(phi)   |
                              |                |
                              |  -z sin(phi)   |
                              |                |
                              |   r cos(phi)   |
                              +-              -+
```

Example 3. We want to compute the gradient of the function $f(r, \theta, \phi) = r\cos(\theta)\sin(\phi)$ $(0 \leq \theta\pi, 0 \leq \theta \leq 2\pi)$ in spherical coordinates.

The vectors

$$\vec{e}_r = \begin{pmatrix} \sin\theta\cos\phi \\ \sin\theta\sin\phi \\ \cos\theta \end{pmatrix}, \vec{e}_\theta = \begin{pmatrix} \cos\theta\cos\phi \\ \cos\theta\sin\phi \\ -\sin\theta \end{pmatrix}, \vec{e}_\phi = \begin{pmatrix} -\sin\phi \\ \cos\phi \\ 0 \end{pmatrix}$$

form an orthogonal system in spherical coordinates.

The scaling factors of the corresponding coordinate transformation (see `linalg::ogCoordTab`) are: $g_1 = |\vec{e}_r| = 1, g_2 = |\vec{e}_\theta| = r, g_3 = |\vec{e}_\phi| = r\sin\theta$, which we use in the following example to compute the gradient of the function f in spherical coordinates:

```
>> delete r, theta, phi:
   linalg::grad(
     r*cos(theta)*sin(phi), [r, theta, phi], [1, r, r*sin(theta)]
   )
```

```
+-                       -+
|   sin(phi) cos(theta)   |
|                         |
|  -sin(phi) sin(theta)   |
|                         |
|   cos(phi) cos(theta)   |
|   -------------------   |
|       sin(theta)        |
+-                       -+
```

Note that the spherical coordinates are already defined in `linalg::ogCoordTab`, i.e., the last result can also be achieved with the input `linalg::grad(r*cos(theta)*sin(phi), [r, theta, phi], Spherical)`.

Changes:

- The parameter `x` must be given as a list, vectors are not longer allowed.
- The gradient is a column vector of the domain `Dom::Matrix()`.

`linalg::hermiteForm` – Hermite normal form of a matrix

`linalg::hermiteForm(A)` computes the Hermite normal form of an integer matrix A. This is an upper-triangular matrix H such that $H_{jj} \geq 0$ and $-\frac{1}{2}H_{jj} \leq H_{ij} < \frac{1}{2}H_{jj}$ for $j > i$.

Call(s):

- `linalg::hermiteForm(A)`

Parameters:

`A` — an integer matrix of category `Cat::Matrix`

Return Value: a matrix of the same domain type as `A`.

Related Functions: `linalg::frobeniusForm`, `linalg::jordanForm`, `linalg::smithForm`, `lllint`

Details:

- If the matrix `A` is not of the domain `Dom::Matrix(Dom::Integer)` then `A` is converted into a matrix of this domain for intermediate computations.

  If this conversion fails, then an error message is returned.

Example 1. We compute the Hermite normal form of the matrix:

```
>> A := Dom::Matrix(Dom::Rational)(
       [[9, -36, 30], [-36, 192, -180], [30, -180, 180]]
   )

                      +-                 -+
                      |   9,   -36,  30   |
                      |                   |
                      |  -36,  192, -180  |
                      |                   |
                      |   30, -180,  180  |
                      +-                 -+
```

```
>> linalg::hermiteForm(A)

                         +-            -+
                         |  3,  0, 30   |
                         |              |
                         |  0, 12,  0   |
                         |              |
                         |  0,  0, 60   |
                         +-            -+
```

Background:

- Let A be an $m \times n$ matrix with coefficients in $\mathbb{Z}$. Then there exists an $m \times n$ matrix $H = \left(h_{ij}\right)$ in Hermite normal form such that $H = AU$ with $|U| = \pm 1$.

  Note that H is unique, if A has full row rank. The matrix U is not unique.

- If A is a square matrix, then the product of the diagonal elements of its Hermite normal form is the determinant of A.

Changes:

- The algorithm works for arbitrary matrices that can be converted into the domain `Dom::Matrix(Dom::Integer)`.

`linalg::hessenberg` – Hessenberg matrix

`linalg::hessenberg(A)` returns an (upper) Hessenberg matrix H.

Call(s):

- `linalg::hessenberg(A<, All>)`

Parameters:

`A` — a square matrix of a domain of category `Cat::Matrix`

Options:

`All` — returns the list `[H, P]` with a Hessenberg matrix H similar to `A` and the corresponding nonsingular transformation matrix P such that $H = PAP^{-1}$.

Return Value: a matrix of the same domain type as `A`, or the list `[H, P]` when the option *`All`* is given.

Related Functions: `linalg::charpoly`

Details:

- `linalg::hessenberg` uses Gaussian elimination without pivoting. There is no special implementation for matrices with floating-point components.

- The component ring of `A` must be a field, i.e., a domain of category `Cat::Field`.

Example 1. Consider the matrix:

```
>> A := Dom::Matrix(Dom::Rational)(
     [[0, 1, 0, -1], [-4/3, 2/3, 5/3, -1/3],
      [-1, 2, 0, 0], [-5/3, 4/3, 1/3, 1/3]]
   )

                   +-                        -+
                   |    0,    1,    0,    -1   |
                   |                           |
                   |  -4/3, 2/3, 5/3, -1/3   |
                   |                           |
                   |   -1,    2,    0,     0   |
                   |                           |
                   |  -5/3, 4/3, 1/3,  1/3   |
                   +-                        -+
```

The following Hessenberg matrix is similar to A:

```
>> H := linalg::hessenberg(A)

                 +-                            -+
                 |    0,  -1/4,  -1/7,    -1    |
                 |                              |
                 |  -4/3,  3/2,  34/21, -1/3    |
                 |                              |
                 |    0,   7/8, -17/14,  1/4    |
                 |                              |
                 |    0,    0,  -72/49,  5/7    |
                 +-                            -+
```

If the corresponding transformation matrix is needed as well, call `linalg::hessenberg` with option *All*:

```
>> [H, P] := linalg::hessenberg(A, All)

  -- +-                          -+   +-                  -
+ --
  |  |    0,  -1/4,  -1/7,    -1   |   |  1,    0,     0,  0  |  |
  |  |                             |   |                      |  |
  |  |  -4/3,  3/2,  34/21, -1/3   |   |  0,    1,     0,  0  |  |
  |  |                             |,  |                      |  |
  |  |    0,   7/8, -17/14,  1/4   |   |  0, -3/4,     1,  0  |  |
  |  |                             |   |                      |  |
  |  |    0,    0,  -72/49,  5/7   |   |  0, -8/7,  -1/7, 1   |  |
  -- +-                          -+   +-                  -
+ --
```

Then P is a nonsingular matrix such that the product PAP^{-1} is equal to H:

```
>> P * A * P^(-1)

                  +-                              -+
                  |     0,  -1/4,   -1/7,     -1   |
                  |                                |
                  |  -4/3,   3/2,   34/21,  -1/3   |
                  |                                |
                  |     0,   7/8, -17/14,    1/4   |
                  |                                |
                  |     0,     0, -72/49,    5/7   |
                  +-                              -+
```

Background:

- An $n \times n$ matrix $A = \left(a_{i,j}\right)_{1 \le i,j \le n}$ is called an (upper) *Hessenberg matrix*, if the following holds: $a_{i,j} = 0$ for all $i, j \in \{1, \ldots, n\}$ with $i > j$.
- For each square matrix A over a field there exists a Hessenberg matrix similar to A. In general, the upper Hessenberg matrix is not unique.
- Reference: K.-H. Kiyek, F. Schwarz: *Lineare Algebra*. Teubner Studienbücher Mathematik, B.G. Teubner Stuttgart, Leipzig, 1999.

Changes:

- `linalg::hessenberg` is a new function.

`linalg::hessian` – Hessian matrix of a scalar function

`linalg::hessian(f, x)` computes the Hesse matrix (the Hessian) of the scalar function $f(\vec{x})$ in Cartesian coordinates, i.e., the square matrix of second partial derivatives of $f(\vec{x})$.

Call(s):

- `linalg::hessian(f,x)`

Parameters:

`f` — an arithmetical expression (the scalar function)
`x` — a list of (indexed) identifiers

Return Value: a matrix of the domain `Dom::Matrix()`.

Related Functions: `diff`, `linalg::grad`, `linalg::jacobian`

Example 1. The Hessian of the function $f(x, y, z) = xy + 2xz$ is the following matrix:

```
>> delete x, y, z:
   linalg::hessian(x*y + 2*z*x, [x, y, z])

                               +-         -+
                               |  0, 1, 2  |
                               |           |
                               |  1, 0, 0  |
                               |           |
                               |  2, 0, 0  |
                               +-         -+
```

Background:

- For a function $f : X \subset \mathbb{R}^n \to \mathbb{R}$ the $n \times n$ matrix

$$H_f(\vec{x}) := \begin{pmatrix} \frac{\partial^2 f(\vec{x})}{\partial x_i^2} & \frac{\partial^2 f(\vec{x})}{\partial x_2 \partial x_1} & \cdots & \frac{\partial^2 f(\vec{x})}{\partial x_p \partial x_1} \\ \frac{\partial^2 f(\vec{x})}{\partial x_1 \partial x_2} & \frac{\partial^2 f(\vec{x})}{\partial x_2^2} & \cdots & \frac{\partial^2 f(\vec{x})}{\partial x_p \partial x_2} \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2 f(\vec{x})}{\partial x_1 \partial x_p} & \frac{\partial^2 f(\vec{x})}{\partial x_2 \partial x_p} & \cdots & \frac{\partial^2 f(\vec{x})}{\partial x_p^2} \end{pmatrix}$$

  is called the *Hesse matrix* of f.

`linalg::hilbert` – Hilbert matrix

`linalg::hilbert(n)` returns the $n \times n$ Hilbert matrix $H = \left(h_{ij}\right)_{1 \le i,j \le n}$ defined by $h_{ij} = (i + j - 1)^{-1}$.

Call(s):

- `linalg::hilbert(n<, R>)`

Parameters:

`n` — the dimension of the matrix: a positive integer
`R` — the component ring: a domain of category `Cat::Rng`; default: `Dom::ExpressionField()`

Return Value: an $n \times n$ matrix of the domain `Dom::Matrix(R)`.

Related Functions: `linalg::invhilbert`

Details:

- Note that the entries of a Hilbert matrix are rational numbers. But the returned matrix is defined over the standard component domain `Dom::ExpressionField()` so that no conversion is necessary when working with other functions that expect or return matrices over that component domain.
- Use `linalg::hilbert(n, Dom::Rational)` to define the $n \times n$ Hilbert matrix over the field of rational numbers.

Example 1. We construct the 3×3 Hilbert matrix:

```
>> H := linalg::hilbert(3)

                                   +-                -+
                                   |   1,  1/2, 1/3   |
                                   |                  |
                                   |  1/2, 1/3, 1/4   |
                                   |                  |
                                   |  1/3, 1/4, 1/5   |
                                   +-                -+
```

This is a matrix of the domain `Dom::Matrix()`).

If you prefer a different component ring, the matrix may be converted into the desired domain afterwards (see `convert`, for example). Alternatively, one can specify the component ring when creating the Hilbert matrix, for example the domain `Dom::Float`:

```
>> H := linalg::hilbert(3, Dom::Float)

          +-                                             -+
          |       1.0,          0.5,      0.3333333333    |
          |                                               |
          |       0.5,     0.3333333333,      0.25        |
          |                                               |
          |  0.3333333333,     0.25,           0.2        |
          +-                                             -+

>> domtype( H )

                              Dom::Matrix(Dom::Float)
```

Background:

- Hilbert matrices are symmetric and positive definite.
- Hilbert matrices of large dimension are notoriously ill-conditioned challenging any numerical inversion scheme. However, their inverse can also be computed by a closed formula (see `linalg::invhilbert`).

Changes:

- `linalg::hilbert` is a new function.

`linalg::intBasis` – basis for the intersection of vector spaces

`linalg::intBasis(S1, S2, ...)` returns a basis for the intersection of the vector spaces spanned by the vectors in $S_1, S_2, \ldots$.

Call(s):

- `linalg::intBasis(S1, S2, ...)`

Parameters:

`S1, S2, ...` — either sets or lists of n-dimensional vectors (a vector is an $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`)

Return Value: a set or a list of vectors, according to the domain type of the parameter `S1`.

Related Functions: `linalg::basis`, `linalg::sumBasis`

Details:

- The domain type of the vectors of the returned set is the domain type of the first parameter `S1`.
- A basis for the zero-dimensional space is the empty set or empty list, respectively.
- The given vectors must be defined over the same component ring which must be a field, i.e., a domain of category `Cat::Field`.

Example 1. We define three vectors $\vec{v}_1, \vec{v}_2, \vec{v}_3$ in $\mathbb{Q}^2$:

```
>> MatQ := Dom::Matrix(Dom::Rational):
   v1 := MatQ([[3, -2]]); v2 := MatQ([[1, 0]]); v3 := MatQ([[5, -
3]])
```

```
+-      -+
| 3, -2 |
+-      -+

+-     -+
| 1, 0 |
+-     -+

+-      -+
| 5, -3 |
+-      -+
```

A basis for the vector space $V_1 \cap V_2 \cap V_3$ with $V_1 =< \{\vec{v}_1, \vec{v}_2, \vec{v}_3\} >$, $V_2 =< \{\vec{v}_1, \vec{v}_3\} >$ and $V_3 =< \{\vec{v}_1 + \vec{v}_2, \vec{v}_2, \vec{v}_1 + \vec{v}_3\} >$ is:

```
>> linalg::intBasis([v1, v2, v3], [v1, v3], [v1 + v2, v2, v1 + v3])

                 -- +-      -+  +-     -+ --
                 |  | 4, -2 |, | 1, 0 |  |
                 -- +-      -+  +-     -+ --
```

Example 2. The intersection of the two vector spaces spanned by the vectors in `S1` and `S2`, respectively:

```
>> S1 := {matrix([[1, 0, 1, 0]]), matrix([[0, 1, 0, 1]])};
   S2 := {matrix([[1, 2, 1, 1]]), matrix([[-1, -2, 1, 0]])}

             { +-          -+  +-          -+ }
             { | 0, 1, 0, 1 |, | 1, 0, 1, 0 | }
             { +-          -+  +-          -+ }

            { +-            -+  +-          -+ }
            { | -1, -2, 1, 0 |, | 1, 2, 1, 1 | }
            { +-            -+  +-          -+ }
```

is the zero-dimensional space:

```
>> linalg::intBasis(S1, S2)

                       {}
```

`linalg::inverseLU` – computing the inverse of a matrix using LU-decomposition

`linalg::inverseLU(A)` computes the inverse A^{-1} of the square matrix A using LU-decomposition.

`linalg::inverseLU(L, U, pivindex)` computes the inverse of the matrix $A = P^{-1}LU$ where `L`, `U` and `pivindex` are the result of an LU-deomposition of the (nonsingular) Matrix A, as computed by `linalg::factorLU`.

Call(s):

- `linalg::inverseLU(A)`
- `linalg::inverseLU(L, U, pivindex)`

Parameters:

`A, L, U` — a square matrix of a domain of category `Cat::Matrix`
`pivindex` — a list of positive integers

Return Value: a matrix of the same domain type as `A` or `L`, respectively.

Related Functions: `_invert`, `linalg::factorLU`, `linalg::matlinsolveLU`

Details:

- The matrix `A` must be nonsingular.
- `pivindex` is a list `[r[1], r[2], ...]` representing a permutation matrix P such that $B = PA = LU$, where $b_{ij} = a_{r[i],j}$.

  It is not checked whether `pivindex` has such a form.
- The component ring of the input matrices must be a field, i.e., a domain of category `Cat::Field`.

Example 1. We compute the inverse of the matrix:

```
>> A := Dom::Matrix(Dom::Real)(
     [[2, -3, -1], [1, 1, -1], [0, 1, -1]]
   )
```

```
                               +-             -+
                               |  2, -3, -1  |
                               |             |
                               |  1,  1, -1  |
                               |             |
                               |  0,  1, -1  |
                               +-             -+
```

using LU-decomposition:

```
>> Ai := linalg::inverseLU(A)
```

```
                      +-                 -+
                      |    0,   1,   -1   |
                      |                   |
                      |  -1/4, 1/2, -1/4  |
                      |                   |
                      |  -1/4, 1/2, -5/4  |
                      +-                 -+
```

We check the result:

```
>> A * Ai, Ai * A
```

```
                  +-         -+  +-         -+
                  |  1, 0, 0  |  |  1, 0, 0  |
                  |           |  |           |
                  |  0, 1, 0  |, |  0, 1, 0  |
                  |           |  |           |
                  |  0, 0, 1  |  |  0, 0, 1  |
                  +-         -+  +-         -+
```

We can also compute the inverse of A in the usual way:

```
>> 1/A
```

```
                      +-                 -+
                      |    0,   1,   -1   |
                      |                   |
                      |  -1/4, 1/2, -1/4  |
                      |                   |
                      |  -1/4, 1/2, -5/4  |
                      +-                 -+
```

`linalg::inverseLU` should be used for efficiency reasons in the case where an LU decomposition of a matrix already is computed, as the next example illustrates.

Example 2. If we already have an LU decomposition of a (nonsingular) matrix, we can compute the inverse of the matrix $A = P^{-1}LU$ as follows:

```
>> LU := linalg::factorLU(linalg::hilbert(3))
```

```
     -- +-           -+  +-                -+            -
-
     |  |   1,  0, 0  |  |  1,  1/2,  1/3   |            |
     |  |             |  |                  |            |
     |  |  1/2, 1, 0  |, |  0, 1/12,  1/12  |, [1, 2, 3] |
     |  |             |  |                  |            |
     |  |  1/3, 1, 1  |  |  0,   0,  1/180  |            |
     -- +-           -+  +-                -+            -
-
```

```
>> linalg::inverseLU(op(LU))

                         +-                  -+
                         |   9,   -36,   30   |
                         |                    |
                         |  -36,  192, -180   |
                         |                    |
                         |   30, -180,  180   |
                         +-                  -+
```

`linalg::inverseLU` then only needs to perform forward and backward substitution to compute the inverse matrix (see also `linalg::matlinsolveLU`).

Changes:

- `linalg::inverseLU` is a new function.

`linalg::invhilbert` – inverse of a Hilbert matrix

`linalg::invhilbert(n)` returns the inverse of the $n \times n$ Hilbert matrix H. The $n \times n$ Hilbert matrix $H = \left(h_{ij}\right)_{1 \le i,j \le n}$ is defined by $h_{ij} = (i + j - 1)^{-1}$.

Call(s):

- `linalg::invhilbert(n<, R>)`

Parameters:

`n` — the dimension of the matrix: a positive integer
`R` — the component ring: a domain of category `Cat::Rng`; default: `Dom::ExpressionField()`

Return Value: an $n \times n$ matrix of the domain `Dom::Matrix(R)`.

Related Functions: `linalg::hilbert`

Details:

- `linalg::invhilbert` uses an explicit formula for the inverse.
- Note that the entries of the inverse of a Hilbert matrix are integers. But the returned matrix is defined over the standard component domain `Dom::ExpressionField()` so that no conversion is necessary when working with other functions that expect or return matrices over that component domain.

- `linalg::invhilbert(n,Dom::Integer)` returns the inverse of the $n \times n$ Hilbert matrix defined over the integers.

Example 1. We compute the inverse of the 3×3 Hilbert matrix:

```
>> A := linalg::invhilbert(3)

                          +-                  -+
                          |    9,   -36,   30  |
                          |                    |
                          |  -36,   192, -180  |
                          |                    |
                          |   30,  -180,  180  |
                          +-                  -+
```

This is a matrix of the domain `Dom::Matrix()`.

If you prefer a different component ring, the matrix may be converted into the desired domain afterwards (see `convert`, for example). Alternatively, one can specify the component ring when calling `linalg::invhilbert`, for example the domain `Dom::Float`:

```
>> A := linalg::invhilbert(3, Dom::Float)

                      +-                          -+
                      |    9.0,   -36.0,   30.0    |
                      |                            |
                      |  -36.0,   192.0, -180.0    |
                      |                            |
                      |   30.0,  -180.0,  180.0    |
                      +-                          -+

>> domtype( A )

                         Dom::Matrix(Dom::Float)
```

Background:

- Hilbert matrices of large dimension are notoriously ill-conditioned, challenging any numerical inversion scheme.

- `linalg::invhilbert` uses the formula

  $$(H^{-1})_{ij} = (-1)^{i+j} \, \frac{c_i \, c_j}{i+j-1} \; , \quad c_i = \frac{(n+i-1)!}{(n-i)! \, ((i-1)!)^2}$$

  for the inverse of the $n \times n$ Hilbert matrix H (N.J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM 1996). All entries of H^{-1} are integers.

Changes:

- `linalg::invhilbert` is a new function.

`linalg::isHermitean` – checks whether a matrix is Hermitean

`linalg::isHermitean(A)` determines whether the matrix A is Hermitean, i.e., whether $A = \overline{A}^t$ ($^-$ denotes conjugation).

Call(s):

- `linalg::isHermitean(A)`

Parameters:

`A` — a square matrix of a domain of category `Cat::Matrix`

Return Value: either `TRUE` or `FALSE`.

Related Functions: `linalg::isPosDef`

Details:

- If the component ring of the matrix `A` does not provide the method `"conjugate"`, then `A` is tested for symmetry, i.e., `linalg::isHermitean` returns `TRUE` if and only if `A` satisfies the equation $A = A^t$.

Example 1. Here is an example of a Hermitean matrix:

```
>> A := Dom::Matrix(Dom::Complex)([[1, I], [-I, 1]])

                          +-         -+
                          |   1,  I   |
                          |           |
                          |  - I, 1   |
                          +-         -+
```

```
>> linalg::isHermitean(A)

                              TRUE
```

The following matrix is not Hermitean:

```
>> B := Dom::Matrix(Dom::Complex)([[1, -I], [-I, 1]])
```

```
+-          -+
|   1,  - I  |
|            |
|  - I,  1   |
+-          -+
```

```
>> linalg::isHermitean(B)
```

```
FALSE
```

The reason is the following:

```
>> linalg::transpose(conjugate(B)) <> B
```

```
+-       -+    +-           -+
|  1, I  |    |   1,  - I  |
|        | <> |            |
|  I, 1  |    |  - I,  1   |
+-       -+    +-           -+
```

Example 2. Here is an example of a symmetric matrix over the integers:

```
>> C := Dom::Matrix(Dom::Integer)([[1, 2], [2, -1]])
```

```
+-        -+
|  1,  2  |
|         |
|  2, -1  |
+-        -+
```

```
>> linalg::isHermitean(C)
```

```
TRUE
```

Changes:

- `linalg::isHermitean` used to be `linalg::isHermitian`.

`linalg::isPosDef` – test a matrix for positive definiteness

`linalg::isPosDef(A)` checks whether the matrix A is positive definite, so that $\vec{x}^{\dagger} A\vec{x} > 0$ for arbitrary vectors $\vec{x} \neq \vec{0}$.

Call(s):

- `linalg::isPosDef(A)`

Parameters:

`A` — a matrix of a domain of category `Cat::Matrix`

Return Value: either `TRUE` or `FALSE`.

Side Effects: Properties of identifiers are taken into account.

Related Functions: `linalg::factorCholesky`, `linalg::isHermitean`

Details:

- The component ring of `A` must be a field, i.e., a domain of category `Cat::Field`.
- An error message is returned, if a result of an intermediate computation cannot be checked for being positive (which could happen, for example, if components of `A` are symbolic).

Example 1. Here is an example of a positive definite matrix:

```
>> MatR := Dom::Matrix( Dom::Real ):
   A := MatR([[14, 6, 9], [6, 17, -4], [9, -4, 13]])

                        +-             -+
                        |  14,  6,   9  |
                        |               |
                        |   6, 17, -4   |
                        |               |
                        |   9, -4, 13   |
                        +-             -+
```

```
>> linalg::isPosDef(A)

                              TRUE
```

The following matrix is not positive definite:

```
>> B := MatR([[1, 2, 3], [2, 3, 4], [5, 6, 7]])

                          +-         -+
                          |  1, 2, 3  |
                          |           |
                          |  2, 3, 4  |
                          |           |
                          |  5, 6, 7  |
                          +-         -+
>> linalg::isPosDef(B)

                                   FALSE
```

Example 2. `linalg::isPosDef` in general does not work for matrices with symbolic entries. It may respond with an error message (because the system in general cannot decide whether a symbolic component is positive), such as for the following matrix:

```
>> delete a, b:
   C := matrix([[a, b], [b, a]])

                                +-        -+
                                |   a, b   |
                                |          |
                                |   b, a   |
                                +-        -+
```

```
>> linalg::isPosDef(C)

 Error: cannot check whether matrix component is positive \
 [linalg::factorCholesky]
```

However, properties of identifiers are taken into account, so that, for example, `linalg::isPosDef` is able to perform the test correctly for the following matrix:

```
>> assume(a > 1): C := matrix([[a, 1], [1, a]]):
```

```
>> linalg::isPosDef(C)

                                   TRUE
```

Note that such computations depend on the power of the underlying property mechanism implemented in the library `property`.

`linalg::isUnitary` – **test whether a matrix is unitary**

`linalg::isUnitary` tests whether the matrix A is a unitary matrix. An $n \times n$ matrix A is unitary, if $A\overline{A}^t = I_n$, where I_n is the $n \times n$ identity matrix.

Call(s):

- `linalg::isUnitary(A)`

Parameters:

`A` — a square matrix of a domain of category `Cat::Matrix`

Return Value: either `TRUE`, `FALSE`, or `UNKNOWN`.

Related Functions: `linalg::orthog`, `linalg::scalarProduct`

Details:

- The square matrix `A` is a unitary matrix, if and only if the columns of `A` form an orthonormal basis with respect to the scalar product `linalg::scalarProduct` of two vectors.
- The correctness of the result `FALSE` of `linalg::isUnitary` can only be guaranteed if the elements of the component ring R of the matrix `A` are canonically represented, i.e., if each element of R has only one unique representation.
- The axiom `Ax::canonicalRep` states that a domain has this property. Hence, `linalg::isUnitary` returns `FALSE` or `UNKNOWN`, respectively, depending on whether the component ring of `A` has the axiom `Ax::canonicalRep`.
- If the component ring of `A` does not define the method `"conjugate"` then it is checked whether `A` is an orthogonal matrix such that $AA^t = E_n$, where E_n is the $n \times n$ identity matrix.

Example 1. The following matrix is unitary:

```
>> A := 1/sqrt(5) * matrix([[1, 2], [2, -1]])

                     +-                 -+
                     |    1/2      1/2   |
                     |   5      2 5      |
                     |   ----,  ------   |
                     |    5        5     |
                     |                   |
                     |     1/2     1/2   |
                     |  2 5        5     |
                     |  ------, - ----   |
                     |    5         5    |
                     +-                 -+
```

```
>> linalg::isUnitary(A)

                               TRUE
```

Changes:

- `linalg::isUnitary` used to be `linalg::isOrthogonal`.

`linalg::jacobian` – Jacobian matrix of a vector function

`linalg::jacobian(v, x)` computes the Jacobian matrix of the vector function $\vec{v}$ with respect to $\vec{x}$.

Call(s):

- `linalg::jacobian(v, x)`

Parameters:

`v` — a list of arithmetical expressions, or a vector (i.e., an $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`)
`x` — a list of (indexed) identifiers

Return Value: a matrix of the domain `Dom::Matrix(R)`, where `R` is the component ring of `v` or the domain `Dom::ExpressionField()`.

Related Functions: `linalg::hessian`, `linalg::grad`

Details:

- If `v` is a vector then the component ring of `v` must be a field (i.e., a domain of category `Cat::Field`) for which differentiation with respect to `x` is defined.
- If `v` is given as a list of arithmetical expressions, then `linalg::jacobian` returns a matrix with the standard component ring `Dom::ExpressionField()`.

Example 1. The Jacobian matrix of the vector function $\vec{v} = \begin{pmatrix} x^3 \\ xz \\ y+z \end{pmatrix}$ is:

```
>> delete x, y, z:
   linalg::jacobian([x^3, x*z, y+z], [x, y, z])

                        +-             -+
                        |      2        |
                        |  3 x , 0, 0   |
                        |               |
                        |    z,  0, x   |
                        |               |
                        |    0,  1, 1   |
                        +-             -+
```

Background:

- For a vector function $\vec{v}: G \subset \mathbb{R}^n \to \mathbb{R}^m, \vec{v} = (v_1, \ldots, v_m)^t$ the matrix

$$J_{\vec{v}}(\vec{x}) := \begin{pmatrix} \frac{\partial v_1(\vec{x})}{\partial x_1} & \ldots & \frac{\partial v_1(\vec{x})}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial v_m(\vec{x})}{\partial x_1} & \ldots & \frac{\partial v_m(\vec{x})}{\partial x_n} \end{pmatrix}$$

  is the *Jacobian matrix* of $\vec{v}$.

`linalg::jordanForm` – Jordan normal form of a matrix

`linalg::jordanForm(A)` returns the Jordan normal form J of the matrix A.

Call(s):

- `linalg::jordanForm(A<, All>)`

Parameters:

`A` — a square matrix of a domain of category `Cat::Matrix`

Options:

`All` — returns the list `[J, P]` with the Jordan normal form J of `A` and the corresponding transformation matrix P such that $A = PJP^{-1}$.

Return Value: either a matrix of the same domain type as `A`, the list `[J, P]` when the option *`All`* is given, or the value `FAIL`.

Related Functions: `linalg::eigenvalues`, `linalg::frobeniusForm`, `linalg::smithForm`, `linalg::hermiteForm`

Details:

- `linalg::jordanForm` computes a nonsingular transformation matrix P and a matrix J such that $A = PJP^{-1}$ with $J = \operatorname{diag}(J_1, \ldots, J_r)$ and Jordan matrices $J_1, \ldots, J_r$.
- The Jordan normal form of a square matrix A over a field F exists if the characteristic polynomial of A splits over F into linear factors. If this is not the case for the matrix `A`, then `linalg::jordanForm` returns `FAIL`.

  The Jordan normal form is unique up to permutations of the Jordan matrices $J_1, \ldots, J_r$.

- The implemented method computes the eigenvalues of `A`. It returns `FAIL` if this is not possible (see `linalg::eigenvalues`).
- The component ring of `A` must be a field, i.e., a domain of category `Cat::Field`.

Example 1. The Jordan normal form of the matrix:

```
>> A := Dom::Matrix(Dom::Complex)([[1, 2], [4, 5]])

                           +-      -+
                           |  1, 2  |
                           |        |
                           |  4, 5  |
                           +-      -+
```

is the following matrix:

```
>> J := linalg::jordanForm(A)

                 +-                          -+
                 |       1/2                  |
                 |  - 2 3    + 3,      0      |
                 |                            |
                 |                   1/2      |
                 |        0,      2 3    + 3  |
                 +-                          -+
```

The corresponding transformation matrix P can be obtained from the result `[J, P]` of `linalg::jordanForm` with the option *All*:

```
>> P := linalg::jordanForm(A, All)[2]

                 +-                          -+
                 |   1/2           1/2        |
                 |  3             3           |
                 |  ---- + 1/2, - ---- + 1/2  |
                 |   6             6          |
                 |                            |
                 |       1/2         1/2      |
                 |      3           3         |
                 |    - ----,       ----      |
                 |       3           3        |
                 +-                          -+
```

We check the result:

```
>> map(P * J * P^(-1), radsimp)
```

```
+-       -+
|  1, 2   |
|         |
|  4, 5   |
+-       -+
```

To get this result we must apply the function `radsimp` to each component of the matrix that is returned by the matrix product PJP^{-1}.

`linalg::matdim` – **dimension of a matrix**

`linalg::matdim(A)` returns the dimension of the matrix A, i.e., the number of rows and columns of A.

Call(s):

- `linalg::matdim(A)`

Parameters:

`A` — an $m \times n$ matrix of a domain of category `Cat::Matrix`

Return Value: the list `[m, n]`, where `m` is the number of rows and `n` is the number of columns of `A`.

Related Functions: `linalg::vecdim`, `linalg::ncols`, `linalg::nrows`

Details:

- `linalg::matdim` is an interface function for the method `"matdim"` of the matrix domain of `A`, i.e., instead of `linalg::matdim(A)` one may call `A::dom::matdim(A)` directly.

Example 1. The dimension of the matrix:

```
>> A := matrix([[1, 2, 3, 4], [3, 1, 4], [5, 6]])

                         +-             -+
                         |  1, 2, 3, 4   |
                         |               |
                         |  3, 1, 4, 0   |
                         |               |
                         |  5, 6, 0, 0   |
                         +-             -+
```

can be determined by:

```
>> linalg::matdim(A)

                                        [3, 4]
```

Changes:

- `linalg::matdim` used to be `linalg::dimen`.

`linalg::matlinsolve` – solving systems of linear equations

`linalg::matlinsolve(A, b)` computes the general solution of the equation $A\vec{x} = \vec{b}$.

Call(s):

- `linalg::matlinsolve(A, b)`
- `linalg::matlinsolve(A, b, list)`
- `linalg::matlinsolve(A, b`<, *`Special`*><, *`Unique`*>`)`
- `linalg::matlinsolve(A, B)`
- `linalg::matlinsolve(A)`

Parameters:

`A` — an $m \times n$ matrix of a domain of category `Cat::Matrix`

`B` — an $m \times k$ matrix of a domain of category `Cat::Matrix`

`b` — an m-dimensional column vector, i.e., a $m \times 1$ matrix of a domain of category `Cat::Matrix`

`list` — a list of n elements of the component ring of `A`

Options:

`Special` — Computes one particular solution of the system $A\vec{x} = \vec{b}$.

`Unique` — Checks whether the system has a unique solution and returns the solution, or the value `NIL` otherwise.

Return Value: a vector, a list `[s, kern]` (possibly empty), where `s` is a solution vector and `kern` is a list of basis vectors for the kernel of `A`, a matrix, or the value `NIL`.

The matrix and the vectors, respectively, are of the domain type `Dom::Matrix(R)`, where `R` is the component ring of `A`.

Related Functions: `linsolve`, `linalg::expr2Matrix`, `linalg::nullspace`, `linalg::matlinsolveLU`, `linalg::wiedemann`, `numeric::matlinsolve`

Details:

- `linalg::matlinsolve(A, b)` returns the solution vector $\vec{x}$ of the system $A\vec{x} = \vec{b}$ if it is a unique solution.
- `linalg::matlinsolve(A, b)` returns a list $[\vec{w}, [\vec{v}_1, \ldots, \vec{v}_r]]$ if the system $A\vec{x} = \vec{b}$ has more than one solution, where $\vec{w}$ is one particular solution, i.e., $A\vec{w} = \vec{b}$ and $\vec{v}_1, \ldots, \vec{v}_r$ form a basis of the kernel of `A`, i.e., the solution space of the homogenous system $A\vec{x} = \vec{0}$.

  Each solution $\vec{x}$ has the form $\vec{x}_s + s_1\vec{v}_1 + \ldots + s_r\vec{v}_r$ $(r \leq n)$ with certain scalars $s_1, \ldots, s_r$.
- A list of n scalars $[s_1, ..., s_n]$ may be passed as the additional parameter `list`. This extracts the solution $\vec{x}_s + s_{i_1}\vec{v}_1 + \ldots + s_{i_r}\vec{v}_r$ with $\{i_1, \ldots, i_r\} = \{1, \ldots, n\} \setminus \{j_1, \ldots, j_l\}$ from the solution space of the system $A\vec{x} = \vec{b}$, where $j_1, \ldots, j_l$ are the characteristic column indices of `A` (see `linalg::gaussJordan`).

  The entries of `list` are converted to elements of the component ring of `A` (an error message is returned if this is not possible).
- If the system $A\vec{x} = \vec{b}$ has no solution, then the empty list `[]` is returned.
- `linalg::matlinsolve(A)` solves the matrix equation $C\vec{x} = \vec{b}$, where $\vec{b}$ is the last column of `A` and C is `A` with the last column deleted.
- `linalg::matlinsolve(A, B)` returns the solution X of the matrix equation $AX = B$, if it has exactly one solution. Otherwise the empty list `[]` is returned.
- The vector `b` and the matrix `B` respectively, are converted into the domain `Dom::Matrix(R)`, where `R` is the component ring of `A`. Solution vectors also belong to this domain.
- The component ring of `A` must be an integral domain, i.e., a domain of category `Cat::IntegralDomain`.
- `linalg::matlinsolve` can compute the general solution for systems with more than one solution only over fields, i.e., component rings of category `Cat::Field`. If in this case the component ring of `A` does not have a canonical representation of the zero element, then it may happen that `linalg::matlinsolve` does not find a basis for the null space. In such a case, a wrong result is returned.
- `linalg::matlinsolve` does not exploit the structure of `A`, e.g., sparsity. A matrix is *sparse* if it has many zero components (see example 5).
- To get a floating-point approximation use the function `numeric::matlinsolve`. Also if the input matrices are defined over the component ring `Dom::Float`, we recommend for efficieny reasons to use `numeric::matlinsolve` instead of `linalg::matlinsolve`!

Option <*Special*>:

- Only one particular solution `w` of the system $A\vec{x} = \vec{b}$ is returned. This supresses the computation of a basis for the kernel of `A`.

Option <*Unique*>:

- Checks whether the system has a unique solution and returns it. The return value `NIL` means that the system has more than one solution.

Example 1. We solve the linear system:

$$\begin{pmatrix} 1 & 2 \\ -1 & 2 \end{pmatrix} \cdot \vec{x} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

over the reals. First we enter the coefficient matrix and the right-hand side:

```
>> MatR := Dom::Matrix(Dom::Real):
   A := MatR([[1, 2], [-1, 2]]); b := MatR([1, -1])

                         +-         -+
                         |    1, 2   |
                         |           |
                         |   -1, 2   |
                         +-         -+

                          +-     -+
                          |   1   |
                          |       |
                          |  -1   |
                          +-     -+
```

Next we call `linalg::matlinsolve` to solve the system:

```
>> x := linalg::matlinsolve(A, b)

                           +-    -+
                           |  1   |
                           |      |
                           |  0   |
                           +-    -+
```

We see that the system has exactly one solution. The vector `x` satisfies the matrix equation given above:

```
>> A * x
```

```
+-     -+
|   1   |
|       |
|  -1   |
+-     -+
```

Example 2. The system:

$$\begin{pmatrix} 1 & 2 \\ -1 & -2 \end{pmatrix} \cdot \vec{x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

does not have a solution over $\mathbb{R}$ (in fact, over no component domain):

```
>> A := MatR([[1, 2], [-1, -2]]): b := MatR([1, 0]):
   linalg::matlinsolve(A, b)

                                   []
```

Example 3. We solve the linear system:

$$\begin{pmatrix} 1 & 1 & -4 & -7 & -6 \\ 0 & 1 & -3 & -5 & -7 \end{pmatrix} \cdot \vec{x} = \begin{pmatrix} 30 \\ 17 \end{pmatrix}$$

over the rational numbers. First we enter the coefficient matrix and the right-hand side:

```
>> MatQ := Dom::Matrix(Dom::Rational):
   A := MatQ([[1, 1, -4, -7, -6], [0, 1, -3, -5, -7]]);
   b := MatQ([30, 17])

                       +-                     -+
                       |  1, 1, -4, -7, -6   |
                       |                       |
                       |  0, 1, -3, -5, -7   |
                       +-                     -+

                              +-     -+
                              |  30   |
                              |       |
                              |  17   |
                              +-     -+
```

Next we call `linalg::matlinsolve` to solve the system:

```
>> sol:= linalg::matlinsolve(A, b)
```

```
-- +-     -+   -- +-    -+  +-    -+  +-     -+ -- --
|  |  13   |   |  |  1   |  |  2   |  |  -1   |  |  |
|  |       |   |  |      |  |      |  |       |  |  |
|  |  17   |   |  |  3   |  |  5   |  |   7   |  |  |
|  |       |   |  |      |  |      |  |       |  |  |
|  |   0   |,  |  |  1   |, |  0   |, |   0   |  |  |
|  |       |   |  |      |  |      |  |       |  |  |
|  |   0   |   |  |  0   |  |  1   |  |   0   |  |  |
|  |       |   |  |      |  |      |  |       |  |  |
|  |   0   |   |  |  0   |  |  0   |  |   1   |  |  |
-- +-     -+   -- +-    -+  +-    -+  +-     -+ -- --
```

The result is to be interpreted as follows: The first vector of the list `sol` is a particular solution of the linear system:

```
>> A * sol[1]

                                   +-    -+
                                   |  30  |
                                   |      |
                                   |  17  |
                                   +-    -+
```

The second entry of the list contains a basis for the null space of A, i.e., the solution space of the corresponding homogenous system $A\vec{x} = \vec{0}$ (the kernel of A). The basis returned is given as a list of vectors.

The following input checks this fact by computing the product $A\vec{x}$ for each vector $\vec{x}$ of the list `sol[2]`:

```
>> map(sol[2], x -> A * x)

                         -- +-    -+  +-    -+  +-    -+ --
                         |  |  0   |  |  0   |  |  0   |  |
                         |  |      |, |      |, |      |  |
                         |  |  0   |  |  0   |  |  0   |  |
                         -- +-    -+  +-    -+  +-    -+ --
```

Any solution of the linear system can be represented as a sum of a particular solution (here: `sol[1]`) and a linear combination of the basis vectors of the kernel of A. Hence our input system has an infinite number of solutions.

For example, another solution of the system is given by:

```
>> x := sol[1] + 1*sol[2][1] + 1/2*sol[2][2] - 2*sol[2][3]

                                  +-      -+
                                  |   17   |
                                  |        |
                                  |  17/2  |
                                  |        |
                                    |    1    |
                                    |         |
                                    |   1/2   |
                                    |         |
                                    |   -2    |
                                    +-       -+
```

```
>> A * x

                                     +-     -+
                                     |  30  |
                                     |      |
                                     |  17  |
                                     +-     -+
```

If we identify the columns of the coefficient matrix A of our linear system with the variables x_1, x_2, x_3, x_4, x_5, then we see from the general solution that the variables x_3, x_4, x_5 act as free parameters. They can be assigned arbitrary rational values to obtain a unique solution.

By giving a list of values for these variables as a third parameter to `linalg::matlinsolve`, we can select a certain vector from the set of all solutions of the linear system. For example, to select the same vector `x` as chosen in the previous input, we enter:

```
>> linalg::matlinsolve(A, b, [0, 0, 1, 1/2, -2])

                                    +-       -+
                                    |   17    |
                                    |         |
                                    |  17/2   |
                                    |         |
                                    |    1    |
                                    |         |
                                    |   1/2   |
                                    |         |
                                    |   -2    |
                                    +-       -+
```

If one is only interested in a particular solution and does not need the general solution of the linear system, one may enter:

```
>> linalg::matlinsolve(A, b, Special)

                                     +-     -+
                                     |  13  |
                                     |      |
                                     |  17  |
                                     |      |
                                     |   0  |
                          |       |
                          |   0   |
                          |       |
                          |   0   |
                          +-     -+
```

This call suppresses the computation of the kernel of A.

Example 4. If the linear system is given in form of equations the function `linalg::expr2Matrix` can be used to form the corresponding matrix equation:

```
>> delete x, y, z:
   Ab := linalg::expr2Matrix(
     [x + y + z = 6, 2*x + y + 2*z = 10, x + 3*y + z = 10]
   )

                    +-               -+
                    |  1, 1, 1,   6   |
                    |                 |
                    |  2, 1, 2, 10    |
                    |                 |
                    |  1, 3, 1, 10    |
                    +-               -+
```

The result here is the extended coefficient matrix of the input system, i.e., the right-hand side vector $\vec{b}$ is the 4th column vector of the matrix `Ab`. Since we did not specify a component ring for this matrix, the standard component ring for matrices, the domain `Dom::ExpressionField()`, was chosen.

To solve the linear system, we call:

```
>> linalg::matlinsolve(Ab)

                 -- +-    -+   -- +-     -+ -- --
                 |  |  4   |   |  |   -1  |  |  |
                 |  |      |   |  |       |  |  |
                 |  |  2   |,  |  |    0  |  |  |
                 |  |      |   |  |       |  |  |
                 |  |  0   |   |  |    1  |  |  |
                 -- +-    -+   -- +-     -+ -- --
```

We see that the system has an infinite number of solutions. The third variable z acts as a free parameter and therefore can have any (complex) value.

To get the general solution in parameter form, one may use parameters for the variables x, y, z of the input system:

```
>> delete u, v, w:
   sol := linalg::matlinsolve(Ab, [u, v, w])
```

```
+-          -+
|  - w + 4   |
|            |
|     2      |
|            |
|     w      |
+-          -+
```

This is possible here because we perform the matrix computations over `Dom::ExpressionField()` which allows to compute with symbolical (arithmetical) expressions.

To select a certain vector from the set of solutions, for example, the solution for $w = 1$, we enter:

```
>> x := subs(sol, w = 1)
```

```
+-   -+
|  3  |
|     |
|  2  |
|     |
|  1  |
+-   -+
```

Example 5. Suppose that we have a system of linear equations with a sparse structure, i.e., with a coefficient matrix having many zero components. For example:

```
>> eqs := {x1 + x5 = 0, x2 - x4 = 1, x3 + 2*x5 = 2, x4 - x5 = -
1}:
   Ab := linalg::expr2Matrix(eqs)
```

```
+-                         -+
|  0,  1, 0, -1, 0, -1      |
|                           |
|  1,  0, 0,  2, 0,  2      |
|                           |
|  0, -1, 1,  0, 0,  1      |
|                           |
|  0,  0, 0,  1, 1,  0      |
+-                         -+
```

As `linalg::matlinsolve` does not exploit the sparsity of the given coefficient matrix, we recommend not to solve the system with `linalg::matlinsolve`, but to use the function `linsolve` which directly works on the equations and therefore preserves the sparse structure of the input system:

```
>> linsolve(eqs)
```

```
        [x3 = 2 x1 + 2, x4 = - x1 - 1, x2 = -x1, x5 = -x1]
```

If a system is given in matrix form, we recommend to use the function `numeric::matlinsolve` with option `Symbolic` instead of `linalg::matlinsolve` even for exact computations. This function works more efficiently than `linalg::matlinsolve` and is able to exploit sparsity:

```
>> A := linalg::delCol(Ab, 5): b := linalg::col(Ab, 6):
   numeric::matlinsolve(A, b)

                       -- +-       -+  +-       -+ --
                       |  |   2.0   |  |  -2.0   |  |
                       |  |         |  |         |  |
                       |  |  -1.0   |  |   1.0   |  |
                       |  |         |  |         |  |
                       |  |    0    |, |    0    |  |
                       |  |         |  |         |  |
                       |  |    0    |  |    0    |  |
                       |  |         |  |         |  |
                       |  |    0    |  |    1    |  |
                       -- +-       -+  +-       -+ --
```

Note that the function `numeric::matlinsolve` always works over a subfield of the complex numbers and does not allow to specify the domain of computation. Without the option `Symbolic` `numeric::matlinsolve` converts input data to floating point numbers.

Example 6. Let us check whether the matrix equation

$$\begin{pmatrix} 1 & 2 \\ -2 & 3 \end{pmatrix} \cdot \vec{x} = \begin{pmatrix} 4 & 2 \\ 6 & 3 \end{pmatrix}$$

has a unique solution over the integers.

We start by entering the coefficient matrix and the right-hand side matrix:

```
>> MatZ := Dom::Matrix(Dom::Integer):
   A := MatZ([[1, 2], [-2, 3]]); B := MatZ([[4, 2], [6, 3]])

                                    +-        -+
                                    |   1, 2   |
                                    |          |
                                    |  -2, 3   |
                                    +-        -+

                                    +-       -+
                                    |  4, 2   |
                                    |         |
                                    |  6, 3   |
                                    +-       -+
```

Next we solve the matrix equation:

```
>> X := linalg::matlinsolve(A, B)

                        +-        -+
                        |  0, 0    |
                        |          |
                        |  2, 1    |
                        +-        -+
```

The equation indeed has a unique solution (otherwise the answer of `linalg::matlinsolve` would be the empty list `[]`). Let us check the result:

```
>> A * X

                        +-        -+
                        |  4, 2    |
                        |          |
                        |  6, 3    |
                        +-        -+
```

Background:

- Let A be an $m \times n$ matrix with components from a field F and $\vec{b}$ an m-dimensional vector over F. Let $(A, \vec{b})$ be the extended coefficient matrix of the linear system $A\vec{x} = \vec{b}$.

  Then the following holds:

  - The linear system $A\vec{x} = \vec{b}$ has a solution, if and only if $\text{rank}(A, \vec{b}) = \text{rank}(A)$.
  - It has exactly one solution, if and only if $\text{rank}(A, \vec{b}) = \text{rank}(A) = n$.
  - If $\vec{x}_s$ is a solution of the system $A\vec{x} = \vec{b}$ and $\{\vec{v}_1, \ldots, \vec{v}_r\}$ a basis of the kernel of A, then

    $$L(A, \vec{b}) = \{\vec{x}_s + \lambda_1\vec{v}_1 + \ldots \lambda_r\vec{v}_r \mid \lambda_1, \ldots, \lambda_r \in F\}$$

    is the set of all solutions of the linear system $A\vec{x} = \vec{b}$, the *general solution* of the (inhomogeneus) linear system.

- The *kernel of the matrix A* is defined as:

  $$\ker(A) := \left\{\vec{w} \mid A\vec{w} = \vec{0}\right\}.$$

  The kernel of A is a vector space over F of dimension $n - \text{rang}(A)$.

Changes:

- `linalg::matlinsolve` used to be `linalg::linearSolve`.

`linalg::matlinsolveLU` – solving the linear system given by an LU decomposition

`linalg::matlinsolveLU(L, U, b)` solves the linear system $LU\vec{x} = \vec{b}$, where the matrices L and U form an LU-decomposition, as computed by `linalg::factorLU`.

Call(s):

- `linalg::matlinsolveLU(L, U, b)`
- `linalg::matlinsolveLU(L, U, B)`

Parameters:

`L` — an $n \times n$ lower triangular matrix of a domain of category `Cat::Matrix`
`U` — an $n \times n$ upper triangular form matrix of the same domain as `L`
`B` — an $n \times k$ matrix of a domain of category `Cat::Matrix`
`b` — an n-dimensional column vector, i.e., an $n \times 1$ matrix of a domain of category `Cat::Matrix`

Return Value: an n-dimensional solution vector or $n \times k$ dimensional solution matrix, respectively, of the domain type `Dom::Matrix(R)`, where `R` is the component ring of `A`.

Related Functions: `linalg::factorLU`, `linalg::inverseLU`, `linalg::matlinsolve`

Details:

- If the third parameter is an $n \times k$ matrix `B` then the result is an $n \times k$ matrix X satisfying the matrix equation $LUX = B$.
- The system to be solved always has a unique solution.
- The diagonal entries of the lower diagonal matrix `L` must be equal to one (*Doolittle*-decomposition, see `linalg::factorLU`).
- `linalg::matlinsolveLU` expects `L` and `U` to be nonsingular.
- `linalg::matlinsolveLU` does not check on any of the required properties of `L` and `U`.
- The component ring of the matrices `L` and `U` must be a field, i.e., a domain of category `Cat::Field`.

The parameters must be defined over the same component ring.

Example 1. We solve the system

$$\begin{pmatrix} 2 & -3 & -1 \\ 1 & 1 & -1 \\ 0 & 1 & -1 \end{pmatrix} X = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}:$$

```
>> MatR := Dom::Matrix(Dom::Real):
   A  := MatR([[2, -3, -1], [1, 1, -1], [0, 1, -1]]);
   I3 := MatR::identity(3)

                         +-           -+
                         |  2, -3, -1  |
                         |             |
                         |  1,  1, -1  |
                         |             |
                         |  0,  1, -1  |
                         +-           -+

                          +-         -+
                          |  1, 0, 0  |
                          |           |
                          |  0, 1, 0  |
                          |           |
                          |  0, 0, 1  |
                          +-         -+
```

We start by computing an LU-decomposition of A:

```
>> LU := linalg::factorLU(A)

     -- +-             -+  +-              -+             -
-
     |  |   1,   0,  0  |  |  2,  -3,  -1   |              |
     |  |               |  |                |              |
     |  |  1/2,  1,  0  |, |  0, 5/2, -1/2  |, [1, 2, 3]   |
     |  |               |  |                |              |
     |  |   0,  2/5, 1  |  |  0,  0,   -4/5  |              |
     -- +-             -+  +-              -+             -
-
```

Now we solve the system $AX = I_3$, which gives us the inverse of A:

```
>> Ai := linalg::matlinsolveLU(LU[1], LU[2], I3)
```

```
+-                    -+
|    0,    1,    -1    |
|                      |
|  -1/4, 1/2, -1/4     |
|                      |
|  -1/4, 1/2, -5/4     |
+-                    -+
```

```
>> A * Ai, Ai * A

+-           -+   +-           -+
|  1, 0, 0    |   |  1, 0, 0    |
|             |   |             |
|  0, 1, 0    |,  |  0, 1, 0    |
|             |   |             |
|  0, 0, 1    |   |  0, 0, 1    |
+-           -+   +-           -+
```

Changes:

- `linalg::matlinsolveLU` is a new function.

`linalg::minpoly` – minimal polynomial of a matrix

`linalg::minpoly(A, x)` computes the minimal polynomial of the square matrix A in x, i.e., the monic polynomial of lowest degree annihilating the matrix A.

Call(s):

- `linalg::minpoly(A, x)`

Parameters:

`A` — a square matrix of a domain of category `Cat::Matrix`
`x` — an indeterminate

Return Value: a polynomial of the domain `Dom::DistributedPolynomial([x],R)`, where `R` is the component ring of `A`.

Related Functions: `linalg::charpoly`, `linalg::frobeniusForm`

Details:

- The minimal polynomial of `A` divides the characteristic polynomial of `A`, by Cayley-Hamilton theorem.
- The component ring of `A` must be a field, i.e., a domain of category `Cat::Field`.

Example 1. We define the following matrix over the rational numbers:

```
>> A := Dom::Matrix(Dom::Rational)(
     [[0, 2, 0], [0, 0, 2], [2, 0, 0]]
   )

                                      +-         -+
                                      |  0, 2, 0  |
                                      |           |
                                      |  0, 0, 2  |
                                      |           |
                                      |  2, 0, 0  |
                                      +-         -+
```

The minimal polynomial of the matrix A in the variable x is then given by:

```
>> delete x: linalg::minpoly(A, x)

                                         3
                                        x  - 8
```

In this case, the minimal polynomial is in fact equal to the characteristic polynomial of A:

```
>> linalg::charpoly(A, x)

                                         3
                                        x  - 8
```

Example 2. The minimal polynomial of the matrix:

```
>> B := matrix([[0, 1, 0], [0, 0, 0], [0, 0, 0]])

                                      +-         -+
                                      |  0, 1, 0  |
                                      |           |
                                      |  0, 0, 0  |
                                      |           |
                                      |  0, 0, 0  |
                                      +-         -+
```

is a polynomial of degree 2:

```
>> m := linalg::minpoly(B, x)

                                            2
                                           x
```

The characteristic polynomial of B has degree 3 and is divided by the minimal polynomial of B:

```
>> p := linalg::charpoly(B, x)

                                            3
                                           x

>> p / m

                                            x
```

Changes:

- `linalg::minpoly` is a new function.

`linalg::multCol` – multiply columns with a scalar

`linalg::multCol(A, c, s)` returns a copy of the matrix A resulting from A by multiplying the c-th column of A with the scalar s.

Call(s):

- `linalg::multCol(A, c, s)`
- `linalg::multCol(A, c1..c2, s)`
- `linalg::multCol(A, list, s)`

Parameters:

`A` — an $m \times n$ matrix of a domain of category `Cat::Matrix`
`c` — the column index: a positive integer $\leq n$
`c1..c2` — a range of column indices (positive integers $\leq n$)
`list` — a list of column indices (positive integers $\leq n$)

Return Value: a matrix of the same domain type as `A`.

Related Functions: `linalg::addCol`, `linalg::addRow`, `linalg::multRow`

Details:

- `linalg::multCol(A, c1..c2, s)` returns a copy of the matrix `A` obtained from `A` by multiplying those columns whose indices are in the range `c1..c2` with the scalar `s`.
- `linalg::multCol(A, list, s)` returns a copy of the matrix `A` obtained from matrix `A` by multiplying those columns whose indices are contained in `list` with the scalar `s`.
- The scalar `s` is converted into an element of the component ring of the matrix `A`. An error message is returned if the conversion fails.

Example 1. We define the following matrix:

```
>> A := matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

                          +-         -+
                          |  1, 2, 3  |
                          |           |
                          |  4, 5, 6  |
                          |           |
                          |  7, 8, 9  |
                          +-         -+
```

and illustrate the three different input formats for `linalg::multCol`:

```
>> linalg::multCol(A, 2, -1)

                         +-           -+
                         |  1, -2, 3   |
                         |             |
                         |  4, -5, 6   |
                         |             |
                         |  7, -8, 9   |
                         +-           -+

>> linalg::multCol(A, 1..2, 2)

                         +-             -+
                         |   2,  4, 3    |
                         |               |
                         |   8, 10, 6    |
                         |               |
                         |  14, 16, 9    |
                         +-             -+

>> linalg::multCol(A, [3, 1], 0)
```

```
+-         -+
|  0, 2, 0  |
|           |
|  0, 5, 0  |
|           |
|  0, 8, 0  |
+-         -+
```

`linalg::multRow` – multiply rows with a scalar

`linalg::multRow(A, r, s)` returns a copy of the matrix A resulting from A by multiplying the r-th row of A with the scalar s.

Call(s):

- `linalg::multRow(A, r, s)`
- `linalg::multRow(A, r1..r2, s)`
- `linalg::multRow(A, list, s)`

Parameters:

`A`	— an $m \times n$ matrix of a domain of category `Cat::Matrix`
`r`	— the row index: a positive integer $\leq m$
`r1..r2`	— a range of row indices (positive integers $\leq m$)
`list`	— a list of row indices (positive integers $\leq m$)

Return Value: a matrix of the same domain type as `A`.

Related Functions: `linalg::addCol`, `linalg::addRow`, `linalg::multCol`

Details:

- `linalg::multRow(A, r1..r2, s)` returns a copy of the matrix `A` obtained from `A` by multiplying those rows whose indices are in the range `r1..r2` with the scalar `s`.
- `linalg::multRow(A, list, s)` returns a copy of the matrix `A` obtained from matrix `A` by multiplying those rows whose indices are contained in `list` with the scalar `s`.
- The scalar `s` is converted into an element of the component ring of the matrix `A`. An error message is returned if the conversion fails.

Example 1. We define the following matrix:

```
>> A := matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

                       +-         -+
                       |  1, 2, 3  |
                       |           |
                       |  4, 5, 6  |
                       |           |
                       |  7, 8, 9  |
                       +-         -+
```

and illustrate the three different input formats for `linalg::multRow`:

```
>> linalg::multRow(A, 2, -1)

                     +-              -+
                     |   1,  2,   3   |
                     |                |
                     |  -4, -5,  -6   |
                     |                |
                     |   7,  8,   9   |
                     +-              -+
```

```
>> linalg::multRow(A, 1..2, 2)

                      +-            -+
                      |  2,  4,   6  |
                      |              |
                      |  8, 10,  12  |
                      |              |
                      |  7,  8,   9  |
                      +-            -+
```

```
>> linalg::multRow(A, [3, 1], 0)

                       +-         -+
                       |  0, 0, 0  |
                       |           |
                       |  4, 5, 6  |
                       |           |
                       |  0, 0, 0  |
                       +-         -+
```

`linalg::ncols` – number of columns of a matrix

`linalg::ncols(A)` returns the number of columns of the matrix A.

Call(s):

- `linalg::ncols(A)`

Parameters:

`A` — a matrix of a domain of category `Cat::Matrix`

Return Value: a positive integer.

Related Functions: `linalg::matdim`, `linalg::nrows`, `linalg::vecdim`

Example 1. The matrix:

```
>> A:= matrix([[1, 2, 3, 4], [3, 1, 4], [5, 6]])

                        +-             -+
                        |  1, 2, 3, 4  |
                        |              |
                        |  3, 1, 4, 0  |
                        |              |
                        |  5, 6, 0, 0  |
                        +-             -+
```

has four columns:

```
>> linalg::ncols(A)

                                 4
```

`linalg::nonZeros` – number of non-zero elements of a matrix

`linalg::nonZeros(A)` returns the number of non-zero components of the matrix A.

Call(s):

- `linalg::nonZeros(A)`

Parameters:

`A` — a matrix of a domain of category `Cat::Matrix`

Return Value: a nonnegative integer

Example 1. The matrix

```
>> MZ7 := Dom::Matrix(Dom::IntegerMod(7)):
   A := MZ7([[18, -1], [4, 81]])
```

```
                     +-                    -+
                     |  4 mod 7, 6 mod 7  |
                     |                      |
                     |  4 mod 7, 4 mod 7  |
                     +-                    -+
```

has four non-zero entries:

```
>> linalg::nonZeros(A)
```

```
                                4
```

The matrix:

```
>> B := MZ7([[21, 2], [-1, 14]])
```

```
                     +-                    -+
                     |  0 mod 7, 2 mod 7  |
                     |                      |
                     |  6 mod 7, 0 mod 7  |
                     +-                    -+
```

has only two non-zero entries:

```
>> linalg::nonZeros(B)
```

```
                                2
```

`linalg::normalize` – **normalize a vector**

`linalg::normalize(v)` normalizes the vector $\vec{v}$ with respect to the 2-norm ($|\vec{v}| = \sqrt{\vec{v} * \vec{v}}$).

Call(s):

- `linalg::normalize(v)`

Parameters:

`v` — a vector, i.e., an $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`

Return Value: a vector of the same domain type as `v`.

Related Functions: `norm`, `linalg::scalarProduct`

Details:

- The result of `linalg::normalize(v)` is a vector that has norm 1 and the same direction as `v`.
- The scalar product $\vec{v} * \vec{v}$ for a vector $\vec{v}$ is implemented by the function `linalg::scalarProduct`.
- The norm of a vector is computed with the function `norm`, which is overloaded for vectors. See the method `"norm"` of the domain constructor `Dom::Matrix` for details.
- If the norm is an object that cannot be converted into an element of the component ring of `v`, then an error occurs (see example 2).

Example 1. We define the following vector:

```
>> u := matrix([[1, 2]])

                              +-     -+
                              | 1, 2 |
                              +-     -+
```

Then the vector of norm 1 with the same direction as `u` is given by:

```
>> linalg::normalize(u)

                         +-                -+
                         |   1/2     1/2   |
                         |  5      2 5     |
                         |  ----, ------   |
                         |   5       5     |
                         +-                -+
```

Example 2. The following computation fails because the vector $(1, 2)$ cannot be normalized over the rationals:

```
>> v := Dom::Matrix(Dom::Rational)([[1, 2]]):
   linalg::normalize(v)

 Error: can't normalize given vector over its component ring [l\
 inalg::normalize]
```

If we define `v` over the real numbers, then we get the normalized vector of `v` as follows:

```
>> w := Dom::Matrix(Dom::Real)(v): linalg::normalize(w)
```

```
+-               -+
|   1/2     1/2   |
|  5      2 5     |
|  ----, ------   |
|   5      5      |
+-               -+
```

`linalg::nrows` – number of rows of a matrix

`linalg::nrows(A)` returns the number of rows of the matrix *A*.

Call(s):

- `linalg::nrows(A)`

Parameters:

`A` — a matrix of a domain of category `Cat::Matrix`

Return Value: a positive integer.

Related Functions: `linalg::matdim`, `linalg::ncols`, `linalg::vecdim`

Example 1. The matrix:

```
>> A := matrix([[1, 2, 3, 4], [3, 1, 4], [5, 6]])
```

```
+-              -+
|  1, 2, 3, 4    |
|                |
|  3, 1, 4, 0    |
|                |
|  5, 6, 0, 0    |
+-              -+
```

has three rows:

```
>> linalg::nrows(A)
```

```
                             3
```

`linalg::nullspace` – basis for the null space of a matrix

`linalg::nullspace(A)` returns a basis for the null space of the matrix A, i.e., a list B of linearly independent vectors such that $A\vec{x} = \vec{0}$ if and only if $\vec{x}$ is a linear combination of the vectors in B.

Call(s):

- `linalg::nullspace(A)`

Parameters:

`A` — a matrix of a domain of category `Cat::Matrix`

Return Value: a list of (column) vectors of the domain `Dom::Matrix(R)`, where `R` is the component ring of `A`.

Related Functions: `linalg::basis`, `linalg::matlinsolve`, `linsolve`, `numeric::matlinsolve`

Details:

- The component ring of the matrix `A` must be a field, i.e., a domain of category `Cat::Field`.
- If the component ring of `A` does not have a canonical representation of the zero element, it can happen that `linalg::nullspace` does not find a basis for the null space. In such a case, a wrong result is returned.

Example 1. The kernel of the matrix:

```
>> A := Dom::Matrix(Dom::Real)(
     [[3^(1/2)*2 - 2, 2], [4, 3^(1/2)*2 + 2]]
   )

                          +-                      -+
                          |      1/2               |
                          |   2 3    - 2,      2   |
                          |                        |
                          |                 1/2    |
                          |        4,     2 3    + 2  |
                          +-                      -+
```

is one-dimensional, and a basis is $\left\{ \left(\begin{array}{c} -\frac{1}{\sqrt{3}-1} \\ 1 \end{array} \right) \right\}$:

```
>> linalg::nullspace(A)
```

```
-- +-            -+ --
|  |       1      |  |
|  |  - --------  |  |
|  |     1/2      |  |
|  |    3    - 1  |  |
|  |              |  |
|  |       1      |  |
-- +-            -+ --
```

Changes:

- `linalg::nullspace` used to be `linalg::nullSpace`.

`linalg::ogCoordTab` – table of orthogonal coordinate transformations

`linalg::ogCoordTab` is a table of predefined orthogonal coordinate transformations in $\mathbb{R}^3$.

Call(s):

- `linalg::ogCoordTab[ogCoord<, *Scales*>](u1, u2, u3<, c, ...>)`

Parameters:

`ogCoord`	— the name of a predefined coordinate system (an identifier)
`u1,u2,u3`	— names of the coordinates of the specified coordinate system (identifiers)
`c`	— an arithmetical expression

Options:

`Scales`	— returns the scaling factors of the coordinate system `ogCoord`.

Return Value: a function in the coordinates `u1,u2,u3` of the specified coordinate system. The function returns a list of the three vectors $\vec{e}_{u_1}, \vec{e}_{u_2}, \vec{e}_{u_3}$, where each vector is a list of three arithmetical expressions.

Related Functions: `linalg::curl`, `linalg::divergence`, `linalg::grad`, `linalg::hessian`, `linalg::jacobian`

Details:

- The entry associated with `ogCoord` defines a coordinate transformation $\vec{x} = T(\vec{u})$, which maps the vector $\vec{u} = (u_1, u_2, u_3)$ in the corresponding orthogonal coordinate system to a vector $\vec{x} = (x, y, z)$ in Cartesian coordinates.

- The result of the transformation $\vec{x} = T(\vec{u})$ is a list of three arithmetical expressions in the unknows `u1,u2,u3`.

- Some coordinate systems need additional constants, which appear as additional parameters of the transformation T (see example 2).

- `linalg::ogCoordTab` is used by functions such as `linalg::curl`, `linalg::divergence` and `linalg::grad` to perform computations with respect to other coordinates than Cartesian coordinates.

- `linalg::ogCoordTab` defines the following coordinate transformations with the implicit assumptions $0 \le \theta \le \pi$ and $0 \le \phi \le 2\pi$:

  `Cartesian` — $\vec{u} = \vec{x} = (x, y, z)$

  `Spherical` — $\vec{u} = (r, \theta, \phi)$

  $$\begin{aligned} x &= r\sin(\theta)\cos(\phi) \\ y &= r\sin(\theta)\sin(\phi) \\ z &= r\cos(\theta) \end{aligned}$$

  .

  `Cylindrical` — $\vec{u} = (r, \phi, z)$

  $$\begin{aligned} x &= r\cos(\phi) \\ y &= r\sin(\phi) \\ z &= z \end{aligned}$$

  `ParabolicCylindrical` — $\vec{u} = (u, v, z)$

  $$\begin{aligned} x &= \tfrac{1}{2}(u^2 + v^2) \\ y &= uv \\ z &= z \end{aligned}$$

  `Torus` — $\vec{u} = (r, \theta, \phi)$

  $$\begin{aligned} x &= (c - r\cos(\theta))\cos(\phi) \\ y &= (c - r\cos(\theta))\sin(\phi) \\ z &= r\sin(\theta), \quad 0 \le r < c \ (c \text{ a real constant}) \end{aligned}$$

  `RotationParabolic` — $\vec{u} = (u, v, \phi)$

  $$\begin{aligned} x &= uv\cos(\phi) \\ y &= uv\sin(\phi) \\ z &= \tfrac{1}{2}(u^2 - v^2) \end{aligned}$$

`EllipticCylindrical` — $\vec{u} = (u, v, z)$

$$\begin{aligned} x &= c\cosh(u)\cos(v) \\ y &= c\sinh(u)\sin(v) \\ z &= z \quad (c \text{ a real constant}) \end{aligned}$$

Option <`Scales`>:

- Returns the *scaling factors* of the specified coordinate transformation. The scaling factors g_1, g_2, g_3 of a coordinate transformation are defined by $g_i := |\frac{\partial T}{\partial u_i}|$ for $i = 1, 2, 3$.

Example 1. The following call returns the vector (x, y, z) in spherical coordinates, expressed in terms of r, θ and ϕ:

```
>> delete r, theta, phi:
   linalg::ogCoordTab[Spherical](r, theta, phi)
```

```
 [[cos(phi) sin(theta), sin(phi) sin(theta), cos(theta)],

    [cos(phi) cos(theta), sin(phi) cos(theta), -sin(theta)],

    [-sin(phi), cos(phi), 0]]
```

The scaling factors of the corresponding coordinate transformation are:

```
>> linalg::ogCoordTab[Spherical,Scales](r, theta, phi)
```

```
                      [1, r, r sin(theta)]
```

Example 2. We express the Cartesian coordinates (x, y, z) in elliptic cylindrical coordinates written in terms of u, v and z, choosing $c = 1$:

```
>> delete u, v, z:
   linalg::ogCoordTab[EllipticCylindrical](u, v, z, 1)
```

```
 -- --     cos(v) sinh(u)           sin(v) cosh(u)         -
-
 |  |  -----------------------, -----------------------, 0  |,
 |  |          2         2 1/2          2         2 1/2     |
 -- -- (cosh(u)  - cos(v) )     (cosh(u)  - cos(v) )       -
-

    --       sin(v) cosh(u)           cos(v) sinh(u)         -
-
```

```
    |  - ----------------------, ----------------------, 0  |
    |            2         2 1/2           2         2 1/2     |
    --  (cosh(u)  - cos(v) )     (cosh(u)  - cos(v) )       -
-


                --
    , [0, 0, 1]  |
                 |
                --
```

To compute the gradient of the vector function $2xy + z$ in elliptic cylindrical coordinates with $c = 1$ we enter:

```
>> delete x, y, z:
   linalg::grad(2*x*y + z, [x, y, z],
     linalg::ogCoordTab[EllipticCylindrical,Scales](u, v, z, 1)
   )

                 +-                             -+
                 |             2 y              |
                 |  -------------------------   |
                 |           2           2 1/2  |
                 |  (- cos(v)  + cosh(u) )      |
                 |                              |
                 |             2 x              |
                 |  -------------------------   |
                 |           2           2 1/2  |
                 |  (- cos(v)  + cosh(u) )      |
                 |                              |
                 |              1               |
                 +-                             -+
```

`linalg::orthog` – **orthogonalization of vectors**

`linalg::orthog(S)` orthogonalizes the vectors in S using the Gram-Schmidt orthogonalization algorithm.

Call(s):

- `linalg::orthog(S)`

Parameters:

`S` — a set or list of vectors of the same dimension (a vector is an $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`)

Return Value: a set or a list of vectors, respectively.

Related Functions: `linalg::factorQR`, `linalg::isUnitary`, `linalg::normalize`, `linalg::scalarProduct`, `lllint`, `norm`

Details:

- The vectors in `S` are orthogonalized with respect to the scalar product `linalg::scalarProduct`.
- If O is the returned set, then the vectors of O span the same subspace as the vectors in `S`, and they are pairwise orthogonal, i.e.: $\vec{v} \cdot \vec{w} = 0$ for all $\vec{v}, \vec{w} \in O$ with $\vec{v} \neq \vec{w}$.
- The vectors returned are not normalized. To normalize them use `map(O, linalg::normalize)`.
- For an ordered set of orthogonal vectors, `S` should be a list.
- The vectors in `S` must be defined over the same component ring.
- The component ring of the vectors in `S` must be a field, i.e., a domain of category `Cat::Field`.

Example 1. The following list of vectors is a basis of the vector space $\mathbb{R}^3$:

```
>> MatR := Dom::Matrix(Dom::Real):
   S := [MatR([2, 1, 0]), MatR([-3, 1, 1]), MatR([-1, -1, -
1])]
```

```
                        -- +-    -+  +-     -+  +-     -+ --
                        |  |  2  |  |  -3  |  |  -1  |  |
                        |  |     |  |      |  |      |  |
                        |  |  1  |, |   1  |, |  -1  |  |
                        |  |     |  |      |  |      |  |
                        |  |  0  |  |   1  |  |  -1  |  |
                        -- +-    -+  +-     -+  +-     -+ --
```

The Gram-Schmidt algorithm then returns an orthogonal basis for $\mathbb{R}^3$. We get an orthonormal basis with the following input:

```
>> ON := map(linalg::orthog(S), linalg::normalize)
```

```
--               +-        -+  +-              -+ --

|                |     1/2  |  |     1/2   1/2  |  |
|  +-        -+  |    6     |  |    8    15     |  |
|  |     1/2  |  |  - ----  |  |  - ----------  |  |
|  |  2 5     |  |     6    |  |        60      |  |
|  |  ------  |  |          |  |                |  |
|  |    5     |  |    1/2   |  |    1/2   1/2   |  |
|  |          |  |   6      |  |   8    15      |  |
|  |    1/2   |, |   ----   |, |   ----------   |  |
|  |   5      |  |    3     |  |       30       |  |
|  |   ----   |  |          |  |                |  |
|  |    5     |  |    1/2   |  |     1/2   1/2  |  |
|  |          |  |   6      |  |    8    15     |  |
|  |     0    |  |   ----   |  |  - ----------  |  |
|  +-        -+  |    6     |  |        12      |  |
--               +-        -+  +-              -+ --
```

Example 2. The orthogonalization of the vectors:

```
>> T := {matrix([[-2, 5, 3]]), matrix([[0, 2, 1]])}

                  { +-       -+  +-        -+ }
                  { | 0, 2, 1 |, | -2, 5, 3 | }
                  { +-       -+  +-        -+ }
```

gives:

```
>> linalg::orthog(T)

            { +-        -+  +-                    -+ }
            { | -2, 5, 3 |, | 13/19, 11/38, -1/38 | }
            { +-        -+  +-                    -+ }
```

Example 3. The result of `linalg::orthog` is a list or set of linearly independent vectors, even if the input contains linearly dependent vectors:

```
>> MatQ := Dom::Matrix(Dom::Rational):
   S := [MatQ([2, 1]), MatQ([3, 4]), MatQ([-1, 1])]

                 -- +-   -+  +-   -+  +-    -+ --
                 |  |  2  |  |  3  |  |  -1  |  |
                 |  |     |, |     |, |      |  |
                 |  |  1  |  |  4  |  |   1  |  |
                 -- +-   -+  +-   -+  +-    -+ --
```

```
>> linalg::orthog(S)

                       -- +-   -+  +-    -+ --
                       |  |  2  |  |  -1  |  |
                       |  |     |, |      |  |
                       |  |  1  |  |   2  |  |
                       -- +-   -+  +-    -+ --
```

Changes:

- `linalg::orthog` used to be `linalg::ogSystem`.
- The function `linalg::onSystem` was removed. Use `linalg::orthog` and `linalg::normalize` instead.

`linalg::permanent` – **permanent of a matrix**

`linalg::permanent(A)` computes the permanent of the square matrix A.

Call(s):

- `linalg::permanent(A)`

Parameters:

`A` — a square matrix of a domain of category `Cat::Matrix`

Return Value: an element of the component ring of `A`.

Related Functions: `linalg::det`

Details:

- The component ring of the matrix `A` must be a commutative ring, i.e., a domain of category `Cat::CommutativeRing`.

Example 1. We compute the permanent of the following matrix:

```
>> delete a11, a12, a21, a22:
   A := matrix([[a11, a12], [a21, a22]])

                         +-          -+
                         |  a11, a12  |
                         |            |
                         |  a21, a22  |
                         +-          -+
```

which gives us the general formula for the permanent of an arbitrary 2 × 2 matrix:

```
>> linalg::permanent(A)

                              a11 a22 + a12 a21
```

Example 2. The permanent of a matrix can be computed over arbitrary commutative rings. Let us create a random matrix defined over the ring $\mathbb{Z}_6$, the integers modulo 6:

```
>> B := linalg::randomMatrix(5, 5, Dom::IntegerMod(6))

          +-                                                -+
          |  3 mod 6, 2 mod 6, 3 mod 6, 5 mod 6, 4 mod 6  |
          |                                                |
          |  2 mod 6, 5 mod 6, 2 mod 6, 1 mod 6, 1 mod 6  |
          |                                                |
          |  1 mod 6, 3 mod 6, 3 mod 6, 2 mod 6, 2 mod 6  |
          |                                                |
          |  1 mod 6, 0 mod 6, 3 mod 6, 3 mod 6, 5 mod 6  |
          |                                                |
          |  0 mod 6, 0 mod 6, 0 mod 6, 1 mod 6, 3 mod 6  |
          +-                                                -+
```

The permanent of this matrix is:

```
>> linalg::permanent(B)

                                    4 mod 6
```

Its determinant is:

```
>> linalg::det(B)

                                    0 mod 6
```

Background:

- The permanent of an $n \times n$ matrix $A = \left(a_{ij}\right)_{1 \le i,j \le n}$ is defined similary as the determinant of A, only the signs of the permutations do not enter the definition:

  $$\mathrm{perm}(A) := \sum_{\sigma \in S_n} \prod_{j=1}^{n} a_{\sigma(j),j}.$$

  (S_n is the symmetric group of all permutations of $\{1, \ldots, n\}$.)

- In contrast to the computation of the determinant, the computation of the permanent takes exponential time in n!

Changes:

- `linalg::permanent` is a new function.

`linalg::pseudoInverse` – Moore-Penrose inverse of a matrix

`linalg::pseudoInverse(A)` computes the Moore-Penrose inverse of A.

Call(s):

- `linalg::pseudoInverse(A)`

Parameters:

`A` — a matrix of category `Cat::Matrix`

Return Value: a matrix of the same domain type as `A`, or the value `FAIL`.

Related Functions: `_invert`

Details:

- If the Moore-Penrose inverse of `A` does not exist, then `FAIL` is returned.
- The component ring of the matrix `A` must be a field, i.e., a domain of category `Cat::Field`.

Example 1. The Moore-Penrose inverse of the 2 × 3 matrix:

```
>> A := Dom::Matrix(Dom::Complex)([[1, I, 3], [1, 3, 2]])

                          +-          -+
                          |  1, I, 3   |
                          |            |
                          |  1, 3, 2   |
                          +-          -+
```

is the 3 × 2 matrix:

```
>> Astar := linalg::pseudoInverse(A)

              +-                                -+
              |   7/96 + 1/32 I,  1/24 - 1/32 I  |
              |                                  |
              |  - 7/32 - 5/96 I, 5/16 + 7/96 I  |
              |                                  |
              |   7/24 + 1/16 I,  1/96 - 3/32 I  |
              +-                                -+
```

Note that in this example, only:

```
>> A * Astar
```

```
                            +-      -+
                            |  1, 0  |
                            |        |
                            |  0, 1  |
                            +-      -+
```

yields the identity matrix, but not (see "Backgrounds" below):

```
>> Astar * A
```

```
     +-                                                    -
+
     |       11/96,       3/32 - 1/48 I,    29/96 + 1/32 I   |
     |                                                       |
     |   3/32 + 1/48 I,       95/96,       - 1/32 - 1/96 I   |
     |                                                       |
     |  29/96 - 1/32 I, - 1/32 + 1/96 I,       43/48         |
     +-                                                    -
+
```

Background:

- For an invertible matrix A, the Moore-Penrose inverse $A^\star$ of A coincides with the inverse of A. In general, only $AA^\star A = A$ and $A^\star AA^\star = A^\star$ holds.

  If A is of dimension $m \times n$, then $A^\star$ is of dimension $n \times m$.
- The computation of the Moore-Penrose inverse requires the existence of a scalar product on the vector space K^n, where K is the coefficient field of the matrix A. This is only the case for some fields K in theory, but `linalg::scalarProduct` works also for vectors over other fields (e.g. finite fields). The computation of a Moore-Penrose inverse may fail in such cases.

Changes:

- `linalg::pseudoInverse` is a new function.

`linalg::randomMatrix` – generate a random matrix

`linalg::randomMatrix(m, n)` returns an $m \times n$ matrix with random components.

Call(s):

- `linalg::randomMatrix(m, n<, R>)`
- `linalg::randomMatrix(m, n<, R><, bound>, `*`Diagonal`*`)`
- `linalg::randomMatrix(m, n<, R><, bound>, `*`Unimodular`*`)`

Parameters:

`m, n` — positive integers
`R` — the component ring, i.e., a domain of category `Cat::Rng`; default: `Dom::ExpressionField()`
`bound` — an arithmetical expression

Options:

`Diagonal` — creates a random $m \times n$ diagonal matrix over `R`.
`Unimodular` — creates a random $m \times n$ unimodular matrix over `R`.

Return Value: a matrix of the domain `Dom::Matrix(R)`.

Related Functions: `random`, `Dom::Matrix`

Details:

- The call `linalg::randomMatrix(m, n)` returns a random $m \times n$ matrix over the default component ring for matrices, i.e., over the domain `Dom::ExpressionField()`.
- The matrix components are generated by the method `"random"` of the domain `R` (see example 2).
- The parameter `bound` is given as a parameter to the method `"random"` of the domain `R` in order to bound the size of the components of the random matrix. The correct type of `bound` is determined by the method `"random"`. The parameter has no effect if the slot `"random"` does not have a size argument.

Option <*`Unimodular`*>:

- Creates a random $m \times n$ unimodular matrix over `R`, so that its determinant is a unit in `R`.
- The norm of each component of the matrix returned does not exceed `bound`, which must be a positive integer, if specified. The default value of `bound` is 10.

Option <*`Diagonal`*>:

- Creates a random $m \times n$ diagonal matrix over `R`.

Example 1. We create a random square matrix over the integers. Because the matrix is random the created matrix can vary:

```
>> linalg::randomMatrix(2, 2, Dom::Integer)

                        +-            -+
                        |   824,  -65  |
                        |              |
                        |  -814, -741  |
                        +-            -+
```

If you want to bound the size of its components, say between -2 and 2, enter:

```
>> linalg::randomMatrix(2, 2, Dom::Integer, -2..2)

                           +-       -+
                           |  -1, 1  |
                           |         |
                           |  -2, 1  |
                           +-       -+
```

Example 2. The following input creates a random vector over the default component ring `Dom::ExpressionField()`. Because the vector is random the created vector can vary:

```
>> v := linalg::randomMatrix(1, 2)

          array(1..1, 1..2,
                                3         5
                          470 R1  - 494 R1  - 246
            (1, 1) = ---------------------------------,
                                    2          3
                     381 R1 + 747 R1  - 1150 R1  - 535
                                     3         5
                     1169 R1 - 977 R1  + 932 R1  - 781
            (1, 2) = ----------------------------------
                                     2          3
                     - 214 R1 + 10 R1  - 1240 R1  + 712
          )

>> domtype(v)
```

```
Dom::Matrix()
```

The components of this matrix are random univariate polynomials created by the function `polylib::randpoly`. See the method `"random"` of the domain constructor `Dom::ExpressionField` for details.

Example 3. To create a random diagonal matrix over the rationals we enter, for example:

```
>> linalg::randomMatrix(3, 3, Dom::Rational, Diagonal)

                 +-                             -+
                 |  -64/305,    0,        0      |
                 |                               |
                 |     0,    41/617,      0      |
                 |                               |
                 |     0,       0,    -167/509   |
                 +-                             -+
```

Example 4. The following command creates a random unimodular matrix over the integers so that its determinant is either 1 or -1:

```
>> A := linalg::randomMatrix(3, 3, Dom::Integer, Unimodular)

                         +-           -+
                         |   9, 2,  8  |
                         |             |
                         |   4, 1,  0  |
                         |             |
                         |  -1, 0, -7  |
                         +-           -+

>> linalg::det(A)

                              1
```

We can bound the size of the components. The following input returns a unimodular matrix $A = \left(a_{ij}\right)$ with $|a_{ij}| \leq 2$ for $i, j = 1, 2, 3$:

```
>> A := linalg::randomMatrix(3, 3, 2, Unimodular)

                        +-             -+
                        |  -1,  0, -2  |
                        |               |
                        |   1,  1,  0  |
                        |               |
                        |  -2, -1, -1  |
                        +-             -+
```

Since we did not specifiy the component ring, the matrix is defined over the standard component ring for matrices (the domain `Dom::ExpressionField()`):

```
>> domtype(A)

                          Dom::Matrix()
```

Background:

- For generating random unimodular matrices, see Jürgen Hansen: *Generating Problems in Linear Algebra*, MapleTech, Volume 1, No.2, 1994.

Changes:

- The new options *`Diagonal`* and *`Unimodular`* were added.

`linalg::rank` – rank of a matrix

`linalg::rank(A)` computes the rank of the matrix A.

`linalg::rank(S)` computes the rank of the matrix whose columns are the vectors in S.

Call(s):

- `linalg::rank(A)`
- `linalg::rank(S)`

Parameters:

`A` — a matrix of a domain of category `Cat::Matrix`
`S` — a list or set of column vectors of the same dimension (a column vector is an $n \times 1$ matrix of a domain of category `Cat::Matrix`)

Return Value: a nonnegative integer

Related Functions: `linalg::det`, `linalg::gaussElim`

Details:

- The component ring of `A` or of the vectors given in `S`, respectively, must be an integral domain, i.e., a domain of category `Cat::IntegralDomain`.

Example 1. We define the following matrix over $\mathbb{Z}$:

```
>> MatZ := Dom::Matrix( Dom::Integer ):
   A := MatZ([[1, 2, 3, 4], [-1, 0, 1, 0], [3, 5, 6, 9]])
```

```
                        +-            -+
                        |   1, 2, 3, 4  |
                        |               |
                        |  -1, 0, 1, 0  |
                        |               |
                        |   3, 5, 6, 9  |
                        +-            -+
```

and compute its rank:

```
>> linalg::rank(A)
```

```
                                3
```

Example 2. The rank of the matrix $A = (\vec{s_i})_{1 \leq i \leq 3}$ with $\vec{s_1} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$, $\vec{s_2} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ and $\vec{s_3} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ is:

```
>> S:= { MatZ([0,1,1]), MatZ([0,1,0]), MatZ([0,0,1]) }:
   linalg::rank(S)
```

```
                                2
```

Background:

- The row rank of a matrix A is defined as the maximal number of linearly independent row vectors of A. The column rank of A is the maximal number of linearly independent column vectors of A.
- Because for each matrix A its row rank is equal to its column rank this number is just called the rank of A.
- The rank of A is computed by Gaussian elimination (see `linalg::gaussElim`), it is equal to the number of characteristic column indices.

`linalg::row` – extract rows of a matrix

`linalg::row(A, r)` extracts the r-th row vector of the matrix A.

Call(s):

- `linalg::row(A, r)`
- `linalg::row(A, r1..r2)`
- `linalg::row(A, list)`

Parameters:

`A`	— an $m \times n$ matrix of a domain of category `Cat::Matrix`
`r`	— the row index: a positive integer $\leq m$
`r1..r2`	— a range of row indices (positive integers $\leq m$)
`list`	— a list of row indices (positive integers $\leq m$)

Return Value: a single row vector or a list of row vectors; a row vector is a $1 \times n$ matrix of category `Cat::Matrix(R)`, where R is the component ring of `A`.

Related Functions: `linalg::col`, `linalg::delCol`, `linalg::delRow`, `linalg::setCol`, `linalg::setRow`

Details:

- `linalg::row(A, r1..r2)` returns a list of row vectors whose indices are in the range `r1..r2`. If `r2 < r1` then the empty list `[]` is returned.
- `linalg::row(A, list)` returns a list of row vectors whose indices are contained in `list` (in the same order).

Example 1. We define a matrix over $\mathbb{Q}$:

```
>> A := Dom::Matrix(Dom::Rational)(
     [[1, 1/5], [-3/2, 5], [2, -3]]
   )

                    +-             -+
                    |    1,   1/5   |
                    |               |
                    |  -3/2,   5    |
                    |               |
                    |    2,    -3   |
                    +-             -+
```

and illustrate the three different input formats for the function `linalg::row`:

```
>> linalg::row(A, 2)

                       +-        -+
                       | -3/2, 5 |
                       +-        -+
>> linalg::row(A, [2, 1, 3])

            -- +-        -+  +-       -+  +-      -+ --
            |  | -3/2, 5 |, | 1, 1/5 |, | 2, -3 |  |
            -- +-        -+  +-       -+  +-      -+ --

>> linalg::row(A, 2..3)

                  -- +-        -+  +-      -+ --
                  |  | -3/2, 5 |, | 2, -3 |  |
                  -- +-        -+  +-      -+ --
```

`linalg::scalarProduct` – scalar product of vectors

`linalg::scalarProduct(u, v)` computes the scalar product of the vectors $\vec{u} = (u_1, \ldots, u_n)$ and $\vec{v} = (v_1, \ldots, v_n)$ with respect to the standard basis, namely the sum $u_1\overline{v}_1 + \ldots + u_n\overline{v}_n$.

Call(s):

- `linalg::scalarProduct(u, v)`

Parameters:

`u, v` — vectors of the same dimension (a vector is an $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`)

Return Value: an element of the component ring of `u` and `v`.

Side Effects: Properties of identifiers are taken into account.

Related Functions: `linalg::angle`, `linalg::crossProduct`, `linalg::isUnitary`, `linalg::factorQR`, `linalg::orthog`, `norm`

Details:

- The scalar product is also called "inner product" or "dot product".
- If the component ring of the vectors `u` and `v` does not define the entry `"conjugate"`, then `linalg::scalarProduct` uses the scalar product defined by $u_1v_1 + \ldots + u_nv_n$.
- The vectors `u` and `v` must be defined over the same component ring.

- `linalg::scalarProduct` can be redefined to a different scalar product. This also affects the behaviour of functions such as `linalg::angle`, `linalg::factorQR`, `linalg::isUnitary`, `norm` (for vectors and matrices), `linalg::orthog` and `linalg::pseudoInverse` depend on the definition of `linalg::scalarProduct`. See example 3.

Example 1. We compute the scalar product of the vectors $(i, 1)$ and $(1, -i)$:

```
>> MatC := Dom::Matrix(Dom::Complex):
   u := MatC([I, 1]): v := MatC([1, -I]):
   linalg::scalarProduct(u, v)

                                  2 I
```

Example 2. We compute the scalar product of the vectors $\vec{u} = (u_1, u_2)$ and $\vec{v} = (v_1, v_2)$ with the symbolic entries u_1, u_2, v_1, v_2 over the standard component ring for matrices:

```
>> delete u1, u2, v1, v2:
   u := matrix([u1, u2]): v := matrix([v1, v2]):
   linalg::scalarProduct(u, v)

                u1 conjugate(v1) + u2 conjugate(v2)
```

You can use `assume` to tell the system that the symbolic components are to represent real numbers:

```
>> assume([u1, u2, v1, v2], Type::Real):
```

Then the scalar product of $\vec{u}$ and $\vec{v}$ simplifies to:

```
>> linalg::scalarProduct(u, v)

                            u1 v1 + u2 v2
```

Example 3. One particular scalar product in the real vector space of continuous functions on the interval $[0, 1]$ is defined by

$$(f, g) = \int_0^1 f(t)g(t)dt.$$

To compute an orthogonal basis corresponding to the polynomial basis $1, t, t^2, t^3, \ldots$ with respect to this scalar product, we replace the standard scalar product by the following procedure:

```
>> standardScalarProduct := linalg::scalarProduct:
   unprotect(linalg):
   linalg::scalarProduct := proc(u, v)
       local F, f, t;
   begin
       // (0)
       f := expr(u[1] * v[1]);

       // (1)
       t := indets(f);
       if t = {} then t := genident("t") else t := op(t, 1) end_if;

       // (2)
       F := int(f, t = 0..1);

       // (3)
       u::dom::coeffRing::coerce(F)
   end:
```

We start with step `(0)` to convert $f(t)g(t)$ to an expression of a basic domain type, such that the system function `int` in step `(2)` can handle its input (this is not necessary if the elements of the component ring of the vectors are already represented by elements of basic domains).

Step `(1)` extracts the indeterminate of the polynomials, step `(2)` computes the scalar product as defined above and step `(3)` converts the result back to an element of the component ring of vectors `u` and `v`.

Note that we need to unprotect the write protected identifier `linalg`, otherwise the assignment would lead to an error message.

We next create the matrix which consists of the first five of the above polynomials:

```
>> P := matrix([[1, t, t^2, t^3, t^4]])

                          +-        2    3    4-+
                          | 1, t, t , t , t   |
                          +-                 -+
```

If we now perform the Gram-Schmidt orthogonalization procedure on the columns of `P` with the function `linalg::orthog`, we get:

```
>> S := linalg::orthog(linalg::col(P, 1..4))

 --
 |
 |  +- -+  +-        -+  +-          2      -+
 |  | 1 |, | t - 1/2 |, | - t + t   + 1/6 |,
 |  +- -+  +-        -+  +-                 -+
 --
```

```
+-                        -+ --
|           2              |  |
|  3 t   3 t     3         |  |
|  --- - ---- + t  - 1/20  |  |
|   5     2                |  |
+-                        -+ --
```

Each vector in `S` is orthogonal to the other vectors in `S` with respect to the modified scalar product. We check this for the first vector:

```
>> linalg::scalarProduct(S[1], S[j]) $ j = 2..nops(S)

                              0, 0, 0
```

Finally, we undo the redefinition of the scalar product, so as not to run into trouble with subsequent computations:

```
>> linalg::scalarProduct := standardScalarProduct:
   protect(linalg, Error):
```

`linalg::setCol` – **change a column of a matrix**

`linalg::setCol(A, p, c)` returns a copy of matrix A with the p-th column replaced by the column vector $\vec{c}$.

Call(s):

- `linalg::setCol(A, p, c)`

Parameters:

`A` — an $m \times n$ matrix of a domain of category `Cat::Matrix`
`c` — a column vector, or a list that can be converted into a column vector of the domain `Dom::Matrix(R)`, where `R` is the component ring of `A` (a column vector is an $m \times 1$ matrix)

Return Value: a matrix of the same domain type as `A`.

Related Functions: `linalg::col`, `linalg::delCol`, `linalg::delRow`, `linalg::row`, `linalg::setRow`

Details:

- If `c` is a list with at most m elements, then `c` is converted into a column vector. An error message is returned if the conversion is not possible (e.g., if an element of the list cannot be converted into an object of the component ring of `A`; see example 2).

Example 1. We define a matrix over the rationals:

```
>> MatQ := Dom::Matrix(Dom::Rational):
   A := MatQ([[1, 2], [3, 2]])

                           +-       -+
                           |  1, 2   |
                           |         |
                           |  3, 2   |
                           +-       -+
```

and replace the 2nd column by the 2×1 zero vector:

```
>> linalg::setCol(A, 2, MatQ([0, 0]))

                           +-       -+
                           |  1, 0   |
                           |         |
                           |  3, 0   |
                           +-       -+
```

Example 2. We create the 2×2 zero matrix over $\mathbb{Z}_6$:

```
>> B := Dom::Matrix(Dom::IntegerMod(6))(2, 2)

                     +-                  -+
                     |  0 mod 6, 0 mod 6  |
                     |                    |
                     |  0 mod 6, 0 mod 6  |
                     +-                  -+
```

and replace the 2nd column by the vector $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$. We give the column vector in form of a list. Its elements are converted implicitly into objects of the component ring of `B`:

```
>> linalg::setCol(B, 2, [1, -1])
```

```
+-                 -+
|  0 mod 6, 1 mod 6  |
|                    |
|  0 mod 6, 5 mod 6  |
+-                 -+
```

The following input leads to an error message because the number `1/3` can not be converted into an object of type `Dom::IntegerMod(6)`:

```
>> linalg::setCol(B, 1, [1/3, 0])

 Error: invalid column vector [linalg::setCol]
```

`linalg::setRow` – change a row of a matrix

`linalg::setRow(A, p, r)` returns a copy of the matrix A with the p-th row replaced by the row vector $\vec{r}$.

Call(s):

- `linalg::setRow(A, p, r)`

Parameters:

`A` — an $m \times n$ matrix of a domain of category `Cat::Matrix`
`r` — a row vector or a list that can be converted into a row vector the domain `Dom::Matrix(R)`, where `R` is the component ring of `A` (a row vector is a $1 \times n$ matrix)

Return Value: a matrix of the same domain type as `A`.

Related Functions: `linalg::col`, `linalg::delCol`, `linalg::delRow`, `linalg::row`, `linalg::setCol`

Details:

- If `r` is a list with at most n elements, then `r` is converted into a row vector. An error message is returned if the conversion is not possible (e.g., if an element of the list cannot be converted into an object of the component ring of `A`; see example 2).

Example 1. We define a matrix over the rationals:

```
>> MatQ := Dom::Matrix(Dom::Rational):
   A := MatQ([[1, 2], [3, 2]])

                         +-       -+
                         |  1, 2   |
                         |         |
                         |  3, 2   |
                         +-       -+
```

and replace the 2nd row by the 1×2 zero vector:

```
>> linalg::setRow(A, 2, MatQ(1, 2, [0, 0]))

                         +-       -+
                         |  1, 2   |
                         |         |
                         |  0, 0   |
                         +-       -+
```

Example 2. We create the 2×4 zero matrix over $\mathbb{Z}_6$:

```
>> B := Dom::Matrix(Dom::IntegerMod(6))(2, 4)

            +-                                      -+
            |  0 mod 6, 0 mod 6, 0 mod 6, 0 mod 6   |
            |                                        |
            |  0 mod 6, 0 mod 6, 0 mod 6, 0 mod 6   |
            +-                                      -+
```

and replace the 2nd row by the vector $(1, -1, 1, -1)$. We give the row vector in form of a list. Its elements are converted implicitly into objects of the component ring of `B`:

```
>> linalg::setRow(B, 2, [1, -1, 1, -1])

            +-                                      -+
            |  0 mod 6, 0 mod 6, 0 mod 6, 0 mod 6   |
            |                                        |
            |  1 mod 6, 5 mod 6, 1 mod 6, 5 mod 6   |
            +-                                      -+
```

The following input leads to an error message because the number $\frac{1}{3}$ can not be converted into an object of type `Dom::IntegerMod(6)`:

```
>> linalg::setRow(B, 1, [1/3, 0, 1, 0])
```

```
Error: invalid row vector [linalg::setRow]
```

`linalg::smithForm` – Smith canonical form of a matrix

`linalg::smithForm(A)` computes the Smith canonical form of the n-dimensional square matrix A, i.e., an $n \times n$ diagonal matrix S such that $S_{i-1,i-1}$ divides $S_{i,i}$ for $i = 2, \ldots, n$.

Call(s):

- `linalg::smithForm(A)`

Parameters:

`A` — a square matrix of a domain of category `Cat::Matrix`

Return Value: a matrix of the same domain type as `A`.

Related Functions: `linalg::frobeniusForm`, `linalg::hermiteForm`, `linalg::jordanForm`

Details:

- The Smith canonical form of a matrix `A` is unique.
- The component ring of `A` must be a Euclidean ring, i.e., a domain of category `Cat::EuclideanDomain`.

Example 1. We define a matrix over the integers:

```
>> MatZ := Dom::Matrix(Dom::Integer):
   A := MatZ([[9, -36, 30], [-36, 192, -180], [30, -180, 180]])

                      +-                 -+
                      |   9,   -36,  30   |
                      |                   |
                      |  -36,  192, -180  |
                      |                   |
                      |   30, -180,  180  |
                      +-                 -+
```

The Smith canonical form of `A` is then given by:

```
>> linalg::smithForm(A)
```

```
+-           -+
|  3,  0,  0  |
|             |
|  0, 12,  0  |
|             |
|  0,  0, 60  |
+-           -+
```

Example 2. We compute the Smith canonical form of a matrix over a ring of polynomials:

```
>> MatPoly := Dom::Matrix(Dom::DistributedPolynomial([x], Dom::Rational)):
   B := MatPoly(
     [[-(x - 3)^2*(x - 2),(x - 3)*(x - 2)*(x - 4)],
      [(x - 3)*(x - 2)*(x - 4),-(x - 3)^2*(x - 4)]
   ])
```

```
     +-                                                    -
+
     |     3      2                  3      2                |
     |  - x  + 8 x  - 21 x + 18,    x  - 9 x  + 26 x - 24    |
     |                                                       |
     |    3      2                  3       2                |
     |   x  - 9 x  + 26 x - 24,  - x  + 10 x  - 33 x + 36   |
     +-                                                    -
+
```

The Smith canonical form of the matrix `B` is the following matrix:

```
>> linalg::smithForm(B)
```

```
+-                              -+
|  x - 3,           0            |
|                                |
|            3      2            |
|    0,    x  - 9 x  + 26 x - 24 |
+-                              -+
```

Changes:

- `linalg::smithForm` is a new function.

`linalg::stackMatrix` – join matrices vertically

`linalg::stackMatrix(A, B1<, B2, ...>)` returns the matrix formed by joining the matrices $A, B_1, B_2, \ldots$ vertically.

Call(s):

- `linalg::stackMatrix(A, B1 <, B2, ...>)`

Parameters:

`A, B1, B2, ...` — matrices of a domain of category `Cat::Matrix`

Return Value: a matrix of the domain type `Dom::Matrix(R)`, where `R` is the component ring of `A`.

Related Functions: `linalg::concatMatrix`

Details:

- The matrices `B1, B2, ...` are converted into the matrix domain `Dom::Matrix(R)`, where `R` is the component ring of `A`.

  An error message is raised if one of these conversions fails, or if the matrices do not have the same number of columns as the matrix `A`.

Example 1. We define the matrix:

```
>> A:= matrix( [[sin(x),x], [-x,cos(x)]] )

                        +-                  -+
                        |  sin(x),     x     |
                        |                    |
                        |    -x,    cos(x)   |
                        +-                  -+
```

and append the 2 × 2 identity matrix to the lower end of the matrix `A`:

```
>> linalg::stackMatrix(A, matrix::identity(2))

                        +-                  -+
                        |  sin(x),     x     |
                        |                    |
                        |    -x,    cos(x)   |
                        |                    |
                        |     1,       0     |
                        |                    |
                        |     0,       1     |
                        +-                  -+
```

Example 2. We define a matrix from the ring of 2×2 square matrices:

```
>> SqMatQ := Dom::SquareMatrix(2,Dom::Rational):
   A := SqMatQ([[1, 2], [3, 4]])

                                   +-      -+
                                   |  1, 2  |
                                   |        |
                                   |  3, 4  |
                                   +-      -+
```

Note that the following operation:

```
>> AA := linalg::stackMatrix(A, A)

                                   +-      -+
                                   |  1, 2  |
                                   |        |
                                   |  3, 4  |
                                   |        |
                                   |  1, 2  |
                                   |        |
                                   |  3, 4  |
                                   +-      -+
```

returns a matrix of a different domain type as the input matrix:

```
>> domtype(AA)

                         Dom::Matrix(Dom::Rational)
```

`linalg::submatrix` – extract a submatrix or a subvector from a matrix or a vector, respectively

`linalg::submatrix(A, r1..r2, c1..c2)` returns a copy of the submatrix of the matrix A obtained by selecting the rows $r_1, r_1+1, \ldots, r_2$ and the columns $c_1, c_1+1, \ldots, c_2$.

`linalg::submatrix(v,i1..i2)` returns a copy of the subvector of the vector $\vec{v}$ obtained by selecting the components with indices $i_1, i_1+1, \ldots, i_2$.

Call(s):

- `linalg::submatrix(A, r1..r2, c1..c2)`
- `linalg::submatrix(A, rlist, clist)`
- `linalg::submatrix(v, i1..i2)`
- `linalg::submatrix(v, list)`

Parameters:

`A`	— an $m \times n$ matrix of a domain of category `Cat::Matrix`
`v`	— a vector with k components, i.e., a $k \times 1$ or $1 \times k$ matrix of a domain of category `Cat::Matrix`
`r1..r2, c1..c2`	— ranges of row/column indices: positive integers less or equal to m and n, respectively
`rlist, clist`	— lists of row/column indices: positive integers less or equal to m and n, respectively
`i1..i2`	— a range of vector indices: positive integers less or equal to k
`list`	— a list of vector indices: positive integers less or equal to k

Return Value: a matrix of the same domain type as `A` or a vector of the same domain type as `v`, respectively.

Related Functions: `linalg::col`, `linalg::row`, `linalg::substitute`

Details:

- The index notation `A[r1..r2,c1..c2]` and `v[i1..i2]`, respectively, can be used instead of `linalg::submatrix(A, r1..r2, c1..c2)` and `linalg::submatrix(v, i1..i2)`.
- `linalg::submatrix(A,rlist,clist)` returns the submatrix of the matrix `A` whose (i, j)-th component is $a_{rlist[i],clist[j]}$.
- `linalg::submatrix(v,list)` returns the subvector of the vector `v` whose i-th component is $v_{list[i]}$.
- If `v` is a row vector or a column vector, then `linalg::submatrix(v, 1..1, i1..i2)` and `linalg::submatrix(v, i1..i1, 1..1)`, respectively, are valid inputs, and they both are equivalent to the call `linalg::submatrix(v,i1`

Example 1. We define the following matrix:

```
>> A := matrix([[1, x, 0], [0, x^2, 1]])
```

```
                          +-           -+
                          |  1,  x, 0  |
                          |             |
                          |      2      |
                          |  0, x , 1  |
                          +-           -+
```

The submatrix $(a_{1,j})_{1 \le j \le 2}$ of `A` is given by:

```
>> linalg::submatrix(A, 1..1, 1..2)

                                             +-     -+
                                             | 1, x |
                                             +-     -+
```

Equivalent to the use of the index operator we obtain:

```
>> A[1..1, 1..2]

                                             +-     -+
                                             | 1, x |
                                             +-     -+
```

We extract the first and the third column of `A` and get the 2×2 identity matrix:

```
>> linalg::submatrix(A, [1, 2], [1, 3])

                                            +-       -+
                                            |  1, 0  |
                                            |        |
                                            |  0, 1  |
                                            +-       -+
```

Example 2. Vector components can be accessed by a single index or a range of indices. For example, to extract the first two components of the following vector:

```
>> v := matrix([1, 2, 3])

                                              +-    -+
                                              |  1  |
                                              |     |
                                              |  2  |
                                              |     |
                                              |  3  |
                                              +-    -+
```

just enter the command:

```
>> v[1..2]

                                              +-    -+
                                              |  1  |
                                              |     |
                                              |  2  |
                                              +-    -+
```

Of course, the same subvector can be extracted with the command `linalg::submatrix( v,1..2 )`.

The following input returns the vector comprising the first and the third component of `v`:

```
>> linalg::submatrix(v, [1, 3])

                                 +-   -+
                                 |  1  |
                                 |     |
                                 |  3  |
                                 +-   -+
```

Changes:

- `linalg::submatrix` used to be `linalg::extractMatrix`.

`linalg::substitute` – replace a part of a matrix by another matrix

`linalg::substitute(B, A, m, n)` returns a copy of the matrix B, where entries starting at position $[m, n]$ are replaced by the entries of the matrix A.

Call(s):

- `linalg::substitute(B, A, m, n)`

Parameters:

`A, B` — matrices of a domain of category `Cat::Matrix`
`m, n` — positive integers

Return Value: a matrix of the same domain type as `B`.

Related Functions: `linalg::submatrix`, `linalg::concatMatrix`, `linalg::setCol`, `linalg::setRow`, `linalg::stackMatrix`

Details:

- `linalg::substitute(B, A, m, n)` returns a copy of the matrix `B`, where entries starting at position $[m, n]$ are replaced by the entries of the matrix `A`, i.e., B_{mn} is A_{11}.
- If the matrices are defined over different component domains, then the entries of `A` are converted into elements of the component domain of the matrix `B`. If one of these conversions fails, then an error message is returned.

Example 1. We define the following matrix:

```
>> B := 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   |
                     |                   |
                     |   9, 10, 11, 12   |
                     |                   |
                     |  13, 14, 15, 16   |
                     +-                 -+
```

and copy the 2×2 zero matrix into the matrix `B`, beginning at position $[3, 3]$:

```
>> A := matrix(2, 2):
   linalg::substitute(B, A, 3, 3)
```

```
                      +-               -+
                      |   1,  2, 3, 4   |
                      |                 |
                      |   5,  6, 7, 8   |
                      |                 |
                      |   9, 10, 0, 0   |
                      |                 |
                      |  13, 14, 0, 0   |
                      +-               -+
```

Matrix entries out of range are ignored:

```
>> linalg::substitute(B, A, 4, 4)
```

```
                     +-                 -+
                     |   1,  2,  3,  4   |
                     |                   |
                     |   5,  6,  7,  8   |
                     |                   |
                     |   9, 10, 11, 12   |
                     |                   |
                     |  13, 14, 15,  0   |
                     +-                 -+
```

Changes:

- `linalg::substitute` is a new function.

`linalg::sumBasis` – basis for the sum of vector spaces

`linalg::sumBasis(S1, S2, ...)` returns a basis of the vector space $V_1 + V_2 + \ldots$, where V_i denotes the vector space spanned by the vectors in S_i.

Call(s):

- `linalg::sumBasis(S1, S2, ...)`

Parameters:

`S1, S2, ...` — a set or list of vectors of the same dimension (a vector is a $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`)

Return Value: a set or a list of vectors, according to the domain type of the parameter `S1`.

Related Functions: `linalg::basis`, `linalg::intBasis`, `linalg::rank`

Details:

- To obtain an ordered basis, `S1, S2, ...` should be given as lists of vectors.
- A basis of the zero-dimensional space is the empty set or list, respectively.
- The given vectors must be defined over the same component ring, which must be a field, i.e., a domain of category `Cat::Field`.

Example 1. We define three vectors $\vec{v}_1, \vec{v}_2, \vec{v}_3$ over $\mathbb{Q}$:

```
>> MatQ := Dom::Matrix(Dom::Rational):
   v1 := MatQ([[3, -2]]); v2 := MatQ([[1, 0]]); v3 := MatQ([[5, -
3]])
```

```
+-       -+
| 3, -2 |
+-       -+

+-     -+
| 1, 0 |
+-     -+

+-       -+
| 5, -3 |
+-       -+
```

A basis of the vector space $V_1 + V_2 + V_3$ with $V_1 =< \{\vec{v}_1, \vec{v}_2, \vec{v}_3\} >$, $V_2 =< \{\vec{v}_1, \vec{v}_3\} >$ and $V_3 =< \{\vec{v}_1 + \vec{v}_2, \vec{v}_2, \vec{v}_1 + \vec{v}_3\} >$ is:

```
>> linalg::sumBasis([v1, v2, v3], [v1, v3], [v1 + v2, v2, v1 + v3])

           -- +-       -+   +-      -+ --
           |  | 3, -2 |,  | 1, 0 |   |
           -- +-       -+   +-      -+ --
```

Example 2. The following set of two vectors:

```
>> MatQ := Dom::Matrix(Dom::Rational):
   S1 := {MatQ([1, 2, 3]), MatQ([-1, 0, 2])}

            { +-      -+  +-     -+ }
            { |  -1   |  |  1   | }
            { |        |  |       | }
            { |   0   |, |  2   | }
            { |        |  |       | }
            { |   2   |  |  3   | }
            { +-      -+  +-     -+ }
```

is a basis of a two-dimensional subspace of $\mathbb{Q}^3$:

```
>> linalg::rank(S1)

                        2
```

The same holds for the following set:

```
>> S2 := {MatQ([0, 2, 3]), MatQ([2, 4, 6])};
   linalg::rank(S2)
```

```
                      { +-   -+  +-   -+ }
                      { |  0  |  |  2  | }
                      { |     |  |     | }
                      { |  2  |, |  4  | }
                      { |     |  |     | }
                      { |  3  |  |  6  | }
                      { +-   -+  +-   -+ }

                                2
```

The sum of the corresponding two subspaces is the vector space $\mathbb{Q}^3$:

```
>> Q3 := linalg::sumBasis(S1, S2)

                 { +-    -+  +-   -+  +-   -+ }
                 { |  -1  |  |  0  |  |  1  | }
                 { |      |  |     |  |     | }
                 { |   0  |, |  2  |, |  2  | }
                 { |      |  |     |  |     | }
                 { |   2  |  |  3  |  |  3  | }
                 { +-    -+  +-   -+  +-   -+ }
```

`linalg::swapCol` – **swap two columns in a matrix**

`linalg::swapCol(A, c1, c2)` returns a copy of the matrix A with the columns with indices c_1 and c_2 interchanged.

Call(s):

- `linalg::swapCol(A, c1, c2)`
- `linalg::swapCol(A, c1, c2, r1..r2)`

Parameters:

`A` — an $m \times n$ matrix of a domain of category `Cat::Matrix`
`c1, c2` — the column indices: positive integers $\leq n$
`r1..r2` — a range of row indices (positive integers $\leq m$)

Return Value: a matrix of the same domain type as `A`.

Related Functions: `linalg::col`, `linalg::delCol`, `linalg::delRow`, `linalg::row`, `linalg::setCol`, `linalg::setRow`, `linalg::swapRow`

Details:

- The affect of `linalg::swapCol(A, c1, c2, r1..r2)` is that only the components from row `r1` to row `r2` of column `c1` are interchanged with the corresponding components of column `c2`.

Example 1. We consider the following matrix:

```
>> A := matrix(3, 3, (i, j) -> 3*(i - 1) + j)

                          +-         -+
                          |  1, 2, 3  |
                          |           |
                          |  4, 5, 6  |
                          |           |
                          |  7, 8, 9  |
                          +-         -+
```

The following command interchanges the first and the second column of `A`. The result is the following matrix:

```
>> linalg::swapCol(A, 1, 2)

                          +-         -+
                          |  2, 1, 3  |
                          |           |
                          |  5, 4, 6  |
                          |           |
                          |  8, 7, 9  |
                          +-         -+
```

If only the components in the first two rows should be affected, we enter:

```
>> linalg::swapCol(A, 1, 2, 1..2)

                          +-         -+
                          |  2, 1, 3  |
                          |           |
                          |  5, 4, 6  |
                          |           |
                          |  7, 8, 9  |
                          +-         -+
```

The third row remains unchanged.

`linalg::swapRow` – swap two rows in a matrix

`linalg::swapRow(A, r1, r2)` returns a copy of the matrix A with the rows with indices r_1 and r_2 interchanged.

Call(s):

- `linalg::swapRow(A, r1, r2)`
- `linalg::swapRow(A, r1, r2, c1..c2)`

Parameters:

`A` — an $m \times n$ matrix of a domain of category `Cat::Matrix`
`r1, r2` — the row indices: positive integers $\leq m$
`c1..c2` — a range of column indices (positive integers $\leq n$)

Return Value: a matrix of the same domain type as `A`.

Related Functions: `linalg::col`, `linalg::delCol`, `linalg::delRow`, `linalg::row`, `linalg::setCol`, `linalg::setRow`, `linalg::swapCol`

Details:

- The affect of `linalg::swapRow(A, r1, r2, c1..c2)` is that only the components from column `c1` to column `c2` of row `r1` are interchanged with the corresponding components of row `r2`.

Example 1. We consider the following matrix:

```
>> A := matrix(3, 3, (i, j) -> 3*(i - 1) + j)

                         +-         -+
                         |  1, 2, 3  |
                         |           |
                         |  4, 5, 6  |
                         |           |
                         |  7, 8, 9  |
                         +-         -+
```

The following command interchanges the first and the second row of `A`. The result is the following matrix:

```
>> linalg::swapRow(A, 1, 2)
```

```
+-         -+
|  4, 5, 6  |
|           |
|  1, 2, 3  |
|           |
|  7, 8, 9  |
+-         -+
```

If only the components in the first two columns should be affected, we enter:

```
>> linalg::swapRow(A, 1, 2, 1..2)
```

```
+-         -+
|  4, 5, 3  |
|           |
|  1, 2, 6  |
|           |
|  7, 8, 9  |
+-         -+
```

The third column remains unchanged.

`linalg::sylvester` – **Sylvester matrix of two polynomials**

`linalg::sylvester(p, q)` returns the Sylvester matrix of the two polynomials *p* and *q*.

Call(s):

- `linalg::sylvester(p, q)`
- `linalg::sylvester(f, g, x)`

Parameters:

`p, q` — polynomials
`f, g` — polynomials or polynomial expressions of positive degree
`x` — a variable

Return Value: a matrix of the domain `Dom::Matrix(R)`, where `R` is the coefficient domain of the polynomials (see below).

Related Functions: `polylib::discrim`, `polylib::resultant`

Details:

- If no variable is specified, then the polynomials `p` and `q` must be either of the domain `DOM_POLY` or from a domain of category `Cat::Polynomial`. Polynomial expressions are not allowed.

- If the polynomials `p` and `q` are of the domain `DOM_POLY`, then they must be univariate polynomials. The component ring of the Sylvester matrix is the common coefficient ring R of `p` and `q`, except in the following two cases for built-in coefficient rings: If R is `Expr` then the domain `Dom::ExpressionField()` is the component ring of the Sylvester matrix. If R is `IntMod(m)`, then the Sylvester matrix is defined over the ring `Dom::IntegerMod(m)` (see example 2).

- Otherwise, if the polynomials `p` and `q` are from a domain of category `Cat::Polynomial`, then the Sylvester matrix is computed with respect to the main variable of `p` and `q` (see the method `"mainvar"` of the category `Cat::Polynomial`). In the case of univariate polynomials the Sylvester matrix is defined over the common coefficient ring of `p` and `q`. In the case of multivariate polynomials, the Sylvester matrix is defined over the component ring `Dom::DistributedPolynomial(ind, R)`, where `ind` is the list of all variables of `p` and `q` except `x`, and `R` is the common coefficient ring of the polynomials.

- If `f` and `g` are polynomial expressions or multivariate polynomials of type `DOM_POLY`, then you must specifiy the variable `x`.

- In the case of polynomial expressions, the component ring of the Sylvester matrix is the domain `Dom::ExpressionField()` (see example 3).

- In the case of multivariate polynomials the Sylvester matrix is defined over the component ring `Dom::DistributedPolynomial(ind, R)`, where `ind` is the list of all variables of `f` and `g` except `x`, and `R` is the common coefficient ring of the polynomials (see example 4).

- At least one of the input polynomials must have positive degree with respect to the main variable or `x`, respectively, but it is not necessary that both of them have positive degree.

Example 1. The Sylvester matrix of the two polynomials $p = x^2 + 2x - 1$ and $q = x^4 + 1$ over $\mathbb{Z}$ is the following 6×6 matrix:

```
>> delete x: Z := Dom::Integer:
   S := linalg::sylvester(poly(x^2 + 2*x - 1, Z), poly(x^4 + 1, Z))
```

```
                  +-                       -+
                  |  1, 2, -1,  0,  0,  0  |
                  |                        |
                  |  0, 1,  2, -1,  0,  0  |
                  |                        |
                  |  0, 0,  1,  2, -1,  0  |
                  |                        |
                  |  0, 0,  0,  1,  2, -1  |
                  |                        |
                  |  1, 0,  0,  0,  1,  0  |
                  |                        |
                  |  0, 1,  0,  0,  0,  1  |
                  +-                       -+
```

Example 2. If the polynomials have the built-in coefficient ring `IntMod(m)`, then the Sylvester matrix is defined over the domain `Dom::IntegerMod(m)`:

```
>> delete x:
   S:= linalg::sylvester(
     poly(x + 1, IntMod(7)), poly(x^2 - 2*x + 2, IntMod(7))
   )

                 +-                            -+
                 |  1 mod 7, 1 mod 7, 0 mod 7  |
                 |                             |
                 |  0 mod 7, 1 mod 7, 1 mod 7  |
                 |                             |
                 |  1 mod 7, 5 mod 7, 2 mod 7  |
                 +-                            -+

>> domtype(S)

                 Dom::Matrix(Dom::IntegerMod(7))
```

Example 3. The Sylvester matrix of the following two polynomial expressions with respect to the variable `x` is:

```
>> delete x, y:
   S := linalg::sylvester(x + y^2, 2*x^3 - 1, x)

                       +-                -+
                       |      2           |
                       |  1, y ,  0,  0   |
                       |                  |
                       |          2       |
                       |  0,  1, y ,  0  |
                       |                 |
                       |              2  |
                       |  0,  0,  1, y   |
                       |                 |
                       |  2,  0,  0, -1  |
                       +-               -+
```

```
>> domtype(S)

                         Dom::Matrix()
```

The Sylvester matrix of these two polynomials with respect to `y` is the following 2×2 matrix:

```
>> linalg::sylvester(x + y^2, 2*x^3 - 1, y)

                   +-                    -+
                   |     3                |
                   |  2 x  - 1,     0     |
                   |                      |
                   |                3     |
                   |      0,    2 x  - 1  |
                   +-                    -+
```

Example 4. Here is an example for computing the Sylvester matrix of multivariate polynomials:

```
>> delete x, y: Q := Dom::Rational:
   T := linalg::sylvester(poly(x^2 - x + y, Q), poly(x + 2, Q), x)

                        +-          -+
                        |  1, -1, y  |
                        |            |
                        |  1,  2, 0  |
                        |            |
                        |  0,  1, 2  |
                        +-          -+
```

```
>> domtype( T )

 Dom::Matrix(Dom::DistributedPolynomial([y], Dom::Rational,

    LexOrder))
```

The Sylvester matrix of these two multivariate polynomials with respect to `y` is:

```
>> linalg::sylvester(poly(x^2 - x + y, Q), poly(x + 2, Q), y)

                        +-       -+
                        | x + 2 |
                        +-       -+
```

`linalg::tr` – **trace of a matrix**

`linalg::tr(A)` returns the trace of the square matrix A, i.e., the sum of the diagonal elements of A.

Call(s):

- `linalg::tr(A)`

Parameters:

`A` — a square matrix of a domain of category `Cat::Matrix`

Return Value: an element of the component ring of `A`.

Related Functions: `linalg::det`

Example 1. We compute the trace of the following matrix:

```
>> A := Dom::Matrix(Dom::Integer)
     (3, 3, (i, j) -> 3*(i - 1) + j)

                              +-           -+
                              |  1, 2, 3  |
                              |           |
                              |  4, 5, 6  |
                              |           |
                              |  7, 8, 9  |
                              +-           -+

>> linalg::tr(A)

                                   15
```

`linalg::transpose` – **transpose of a matrix**

`linalg::transpose(A)` returns the transpose A^t of the matrix A.

Call(s):

- `linalg::transpose(A)`

Parameters:

`A` — a matrix of a domain of category `Cat::Matrix`

Return Value: a matrix of the same domain type as `A`.

Details:

- `linalg::transpose` is an interface function for the method `"transpose"` of the matrix domain of `A`, i.e., instead of `linalg::transpose(A)` one may call `A::dom::transpose(A)` directly.

Example 1. We define a 3×4 matrix:

```
>> A := matrix([[1, 2, 3, 4], [-1, 0, 1, 0], [3, 5, 6, 9]])

                        +-              -+
                        |   1, 2, 3, 4   |
                        |                |
                        |  -1, 0, 1, 0   |
                        |                |
                        |   3, 5, 6, 9   |
                        +-              -+
```

Then the transpose of `A` is the 4×3 matrix:

```
>> linalg::transpose(A)

                         +-           -+
                         |  1, -1, 3   |
                         |             |
                         |  2,  0, 5   |
                         |             |
                         |  3,  1, 6   |
                         |             |
                         |  4,  0, 9   |
                         +-           -+
```

Background:

- Let $A = \left(a_{ij}\right)_{1 \le i \le m, 1 \le j \le n}$ be an $m \times n$ matrix. Then the transpose of A is the $n \times m$ matrix:

$$A^t = \left(a_{ij}\right)_{1 \le j \le n, 1 \le i \le m} = \begin{pmatrix} a_{11} & a_{21} & \dots & a_{m,1} \\ a_{12} & a_{22} & \dots & a_{m,2} \\ \vdots & \vdots & & \vdots \\ a_{1,n} & a_{2,n} & \dots & a_{mn} \end{pmatrix}.$$

`linalg::vandermondeSolve` – solve a linear Vandermonde system

`linalg::vandermondeSolve(v, y)` returns the solution $\vec{x}$ of the linear Vandermonde system $\sum_{j=1}^{n} v_i^{j-1}\, x_j = y_i$, for $i = 1, \dots, n$.

Call(s):

- `linalg::vandermondeSolve(v, y)`
- `linalg::vandermondeSolve(v, y, Transposed)`

Parameters:

`v` —	a vector with distinct elements (a vector is an $n \times 1$ or $1 \times n$ matrix of category `Cat::Matrix`)
`y` —	a vector of the same dimension and domain type as `v`

Options:

`Transposed` —	returns the solution $\vec{x}$ of the transposed system $\sum_{j=1}^{n} v_j^{i-1}\, x_j = y_i$, for $i = 1, \dots, n$.

Return Value: a vector of the same domain type as `y`.

Related Functions: `solve`, `linsolve`, `linalg::matlinsolve`, `numeric::lagrange`, `numeric::linsolve`, `numeric::matlinsolve`

Details:

- `linalg::vandermondeSolve` uses $O(n^2)$ elementary operations to solve the Vandermonde system. It is faster than the general system solver `solve` and the solver `linsolve`, `numeric::linsolve`, `linalg::matlinsolve` and `numeric::matlinsolve` for linear systems.
- The solution $\vec{x} = (x_1, \dots, x_n)$ returned by

```
linalg::vandermondeSolve([vi $ i=1..n], [yi $ i=1..n])
```

yields the coefficients of the polynomial $p(v) = x_1 + x_2\, v + \cdots + x_n\, v^(n - 1)$ interpolating the data table $(v_1, y_1), \ldots, (v_n, y_n)$, i.e.,

$$p(v_1) = y_1\ ,\ \ldots\ ,\ p(v_n) = y_n\ .$$

See example 1.

Example 1. The Vandermonde points v and the right hand side y of the linear system are entered as vectors:

```
>> delete y0, y1, y2:
   v := matrix([[0, 1, 2]]); y:= matrix([[y0, y1, y2]])
```

```
                                   +-        -+
                                   | 0, 1, 2 |
                                   +-        -+


                                 +-            -+
                                 | y0, y1, y2 |
                                 +-            -+
```

The solution vector is:

```
>> x := linalg::vandermondeSolve(v, y)
```

```
           +-                                        -+
           |        3 y0          y2  y0        y2  |
           |  y0, - ---- + 2 y1 - --, -- - y1 + --  |
           |         2             2   2         2  |
           +-                                        -+
```

The solution yields the coefficients of the interpolating polynomial:

```
>> P := v -> _plus(x[i+1]*v^i $ i=0..2):
```

through the points $(0, y_0), (1, y_1), (2, y_2)$:

```
>> P(v[1]), P(v[2]), P(v[3])
```

```
                                   y0, y1, y2
```

With the optional argument `Transposed`, the linear system with the transposed Vandermonde matrix corresponding to `v` is solved:

```
>> linalg::vandermondeSolve(v, y, Transposed)
```

```
           +-                                        -+
           |       3 y1   y2                y1   y2  |
           |  y0 - ---- + --, 2 y1 - y2, - -- + --  |
           |        2     2                 2    2  |
           +-                                        -+
```

Example 2. The Vandermonde points v and the right hand side y of the linear system are entered as 2×1 matrices:

```
>> Mat := Dom::Matrix(Dom::ExpressionField(normal)):
```

```
>> delete v1, v2, y1, y2:
   v := Mat([v1, v2]): y:= Mat([y1, y2]):
```

We define the vectors over the domain `Dom::ExpressionField(normal)` in order to simplify intermediate computations.

Next, we compute the solution of the corresponding Vandermonde system:

```
>> x := linalg::vandermondeSolve(v, y)

                          +-                  -+
                          |  - v1 y2 + v2 y1  |
                          |  ---------------  |
                          |     - v1 + v2     |
                          |                   |
                          |     - y1 + y2     |
                          |     ---------     |
                          |     - v1 + v2     |
                          +-                  -+
```

We construct the Vandermonde matrix `V` and verify the result:

```
>> V := Mat([[1, v[1]], [1, v[2]]])

                               +-        -+
                               |  1, v1  |
                               |          |
                               |  1, v2  |
                               +-        -+
```

```
>> V * x

                                +-     -+
                                |  y1  |
                                |       |
                                |  y2  |
                                +-     -+
```

Example 3. We solve a Vandermonde system over the field $\mathbb{Z}_7$ (the integers modulo 7) represented by the domain `Dom::IntegerMod(7)`:

```
>> MatZ7 := Dom::Matrix(Dom::IntegerMod(7)):
   v := MatZ7([1, 2, 3]): y := MatZ7([0, 1, 2]):
```

```
>> linalg::vandermondeSolve(v, y)

                          +-          -+
                          |  6 mod 7   |
                          |            |
                          |  1 mod 7   |
                          |            |
                          |  0 mod 7   |
                          +-          -+
```

Background:

- The Vandermonde matrix

  $$V = \begin{pmatrix} 1 & v_1 & v_1^2 & \dots & v_1^{n-1} \\ 1 & v_2 & v_2^2 & \dots & v_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & v_n & v_n^2 & \dots & v_n^{n-1} \end{pmatrix}$$

  generated by $v = [v_1, \dots, v_n]$ is invertible if and only if the v_i are distinct.

- The vector $\vec{x}$ returned by `linalg::vandermondeSolve(x, y)` solves $V\vec{x} = \vec{y}$ and is unique.

- The vector x returned by `linalg::vandermondeSolve(x, y, Transposed)` solves $V^t\vec{x} = \vec{y}$ and is unique.

Changes:

- `linalg::vandermondeSolve` is a new function.

`linalg::vecdim` – number of components of a vector

`linalg::vecdim(v)` returns the number of elements of the vector $\vec{v}$.

Call(s):

- `linalg::vecdim(v)`

Parameters:

`v` — a vector, i.e., an $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`

Return Value: a positive integer.

Related Functions: `linalg::matdim`, `linalg::ncols`, `linalg::nrows`

Example 1. We define a column vector with two elements and a row vector with four elements:

```
>> v1 := matrix([1, 0]); v2 := matrix([[1, 2, 3, 4]])

                             +-   -+
                             |  1  |
                             |     |
                             |  0  |
                             +-   -+

                          +-          -+
                          | 1, 2, 3, 4 |
                          +-          -+
```

`linalg::vecdim` gives us the number of elements, i.e., the dimension of these vectors:

```
>> linalg::vecdim(v1), linalg::vecdim(v2)

                              2, 4
```

In contrast, the function `linalg::matdim` returns the number of rows and columns of these vectors:

```
>> linalg::matdim(v1), linalg::matdim(v2)

                          [2, 1], [1, 4]
```

Changes:

- `linalg::vecdim` used to be `linalg::vectorDimen`.

`linalg::VectorOf` – type specifier for vectors

`linalg::VectorOf(R, n)` is a type specifier for vectors with *n* components over the component ring *R*.

Call(s):

- `linalg::VectorOf(R)`
- `linalg::VectorOf(R, n)`

Parameters:

`R` — the component ring: a library domain
`n` — a positive integer

Return Value: a type expression of the domain type `Type`.

Related Functions: `testtype`

Details:

- `linalg::VectorOf(R)` is a type specifier representing all objects of a domain of category `Cat::Matrix` with component ring `R` and number of rows or number of columns equal to one.
- `linalg::VectorOf(R,n)` is a type specifier representing all objects of a domain of category `Cat::Matrix` with component ring `R` and number of rows equal to `n` and number of columns equal to one, or vice versa.
- `linalg::VectorOf(Type::AnyType,n)` is a type specifier representing all objects of a domain of category `Cat::Matrix` with an arbitrary component ring `R` and number of rows equal to `n` and number of columns equal to one, or vice versa.

Example 1. `linalg::VectorOf` can be used together with `testtype` to check whether a MuPAD object is a vector:

```
>> MatZ := Dom::Matrix(Dom::Integer):
   v := MatZ([1, 0, -1])
```

```
+-     -+
|   1   |
|       |
|   0   |
|       |
|  -1   |
+-     -+
```

The following yields `FALSE` because `v` is 3-dimensional vector:

```
>> testtype(v, linalg::VectorOf(Dom::Integer, 4))

                       FALSE
```

The following yields `FALSE` because `v` is defined over the integers:

```
>> testtype(v, linalg::VectorOf(Dom::Rational))

                       FALSE
```

Of course, `v` can be converted into a vector over the rationals, as shown by the following call:

```
>> testtype(v, Dom::Matrix(Dom::Rational))

                              TRUE
```

This shows that `testtype` in conjunction with `linalg::VectorOf(R)` does not check whether an object can be converted into a vector over the specified component ring `R`. It checks only if the object is a vector whose component ring is `R`.

The following test returns `TRUE` because `v` is a 3-dimensional vector:

```
>> testtype(v, linalg::VectorOf(Type::AnyType, 3))

                              TRUE
```

Example 2. `linalg::VectorOf` can also be used for checking parameters of procedures. The following procedure computes the orthogonal complement of a 2-dimensional vector:

```
>> orth := proc(v:linalg::VectorOf(Type::AnyType, 2))
   begin
       [v[1], v[2]] := [-v[2],v[1]];
       return(v)
   end:

   u := matrix([[1, 2]]); u_ := orth(u)

                             +-    -+
                             | 1, 2 |
                             +-    -+

                             +-     -+
                             | -2, 1 |
                             +-     -+
```

Calling the procedure `orth` with an invalid parameter leads to an error message:

```
>> orth([1, 2])

 Error: Wrong type of 1. argument (type 'slot(Type, VectorOf)(T\
 ype::AnyType, 2)' expected,
        got argument '[1, 2]');
 during evaluation of 'orth'
```

`linalg::vectorPotential` – vector potential of a three-dimensional vector field

`linalg::vectorPotential(j, x)` returns the vector potential of the vector field $\vec{j}(\vec{x})$ with respect to $\vec{x}$. This is a vector field $\vec{v}$ with $\mathrm{curl}_{\vec{x}}(\vec{v}) = \vec{j}$.

Call(s):

- `linalg::vectorPotential(j, [x1, x2, x3]<, Test>)`

Parameters:

`j`	—	a list of three arithmetical expressions, or a 3-dimensional vector (i.e., a 3×1 or 1×3 matrix of a domain of category `Cat::Matrix`)
`x1,x2,x3`	—	(indexed) identifiers

Options:

`Test`	—	`linalg::vectorPotential` only checks whether the vector field `j` has a vector potential and returns `TRUE` or `FALSE`, respectively.

Return Value: a vector with three components, i.e., an 3×1 or $1 \times n$ matrix of a domain of category `Cat::Matrix`, or a boolean value.

Related Functions: `linalg::curl`, `linalg::divergence`, `linalg::grad`

Details:

- The vector potential of a vector function `j` exists if and only if the divergence of `j` is zero. It is uniquely determined.
- If the vector potential of `j` does not exist, then `linalg::vectorPotential` returns `FALSE`.
- If `j` is a vector then the component ring of `j` must be a field (i.e., a domain of category `Cat::Field`) for which definite integration can be performed.
- If `j` is given as a list of three arithmetical expressions, then `linalg::vectorPotential` returns a vector of the domain `Dom::Matrix()`.

Example 1. We check if the vector function $\vec{j}(x, y, z) = \left(x^2 y, -\frac{1}{2}y^2 x, -xyz\right)$ has a vector potential:

```
>> delete x, y, z:
   linalg::vectorPotential(
     [x^2*y, -1/2*y^2*x, -x*y*z], [x, y, z], Test
   )

                                    TRUE
```

The answer is yes, so let us compute the vector potential of $\vec{j}$:

```
>> linalg::vectorPotential(
     [x^2*y, -1/2*y^2*x, -x*y*z], [x, y, z]
   )

                         +-           -+
                         |       2     |
                         |    x y   z  |
                         |  - ------   |
                         |      2      |
                         |             |
                         |     2       |
                         |  - x   y z  |
                         |             |
                         |      0      |
                         +-           -+
```

We check the result:

```
>> linalg::curl(%, [x, y, z])

                          +-         -+
                          |    2      |
                          |   x   y   |
                          |           |
                          |        2  |
                          |    x y    |
                          |  - ----   |
                          |      2    |
                          |           |
                          |  -x y z   |
                          +-         -+
```

Example 2. The vector function $\vec{j} = \left(x^2, 2y, z\right)$ does not have a vector potential:

```
>> linalg::vectorPotential([x^2, 2*y, z], [x, y, z])

                        FALSE
```

Changes:

- The result is a vector even if the vector field is given as a list of expressions.

`linalg::wiedemann` – solving linear systems by Wiedemann's algorithm

`linalg::wiedemann(A, b, mult ...)` tries to find a vector $\vec{x}$ that satisfies the equation $A\vec{x} = \vec{b}$ by using Wiedemann's algorithm.

Call(s):

- `linalg::wiedemann(A, b<, mult>)`
- `linalg::wiedemann(A, b<, mult>, prob)`

Parameters:

`A`	— an $n \times n$ matrix of a domain of category `Cat::Matrix`
`b`	— an n-dimensional column vector, i.e., an $n \times 1$ matrix of a domain of category `Cat::Matrix`
`mult`	— a matrix-vector multiplication method: function or functional expression; default: `_mult`
`prob`	— `TRUE` or `FALSE` (default: `TRUE`)

Return Value: either the list `[x, TRUE]` if a solution for the system $A\vec{x} = \vec{b}$ has been found, or the list `[x, FALSE]` if a non-zero solution for the corresponding homogeneous system $A\vec{x} = \vec{0}$ has been found, or the value `FAIL` (see below).

Related Functions: `linalg::matlinsolve`, `linalg::vandermondeSolve`

Details:

- The parameter `mult` must be a function such that the result of `mult(A,y)` equals $A\vec{y}$ for every n-dimensional column vector $\vec{y}$. The parameter `y` is of the same domain type as `A`. The argument `mult` does not need to handle other types of parameters, nor does it need to handle other matrices than `A`.

- `linalg::wiedemann` uses a probabilistic algorithm. For a deterministic variant enter `FALSE` for the optional parameter `prob`.
- If the system $A\vec{x} = \vec{b}$ does not have a solution, then `linalg::wiedemann` returns `FAIL`.
- If the system $A\vec{x} = \vec{b}$ has more than one solution, then a random one is returned.
- Due to the probabilistic nature of Wiedemann's algorithm, the computation may fail with small probability. In this case `FAIL` is returned. If the deterministic variant is chosen, then the algorithm may be slower for a small number of matrices.
- The vector `b` must be defined over the component ring of `A`.
- The coefficient ring of `A` must be a field, i.e., a domain of category `Cat::Field`.
- It is recommended to use `linalg::wiedemann` only if `mult` uses significantly less than $O(n^2)$ field operations.

Example 1. We define a matrix and a column vector over the finite field with 29 elements:

```
>> MatZ29 := Dom::Matrix(Dom::IntegerMod(29)):
   A := MatZ29([[1, 2, 3], [4, 7, 8], [9, 12, 17]]);
   b := MatZ29([1, 2, 3])
```

```
             +-                                  -+
             |  1 mod 29,  2 mod 29,  3 mod 29  |
             |                                    |
             |  4 mod 29,  7 mod 29,  8 mod 29  |
             |                                    |
             |  9 mod 29, 12 mod 29, 17 mod 29  |
             +-                                  -+

                        +-            -+
                        |  1 mod 29  |
                        |              |
                        |  2 mod 29  |
                        |              |
                        |  3 mod 29  |
                        +-            -+
```

Since `A` does not have a special form that would allow a fast matrix-vector multiplication, we simply use `_mult`. Wiedemann's algorithm works in this case, although it is less efficient than Gaussian elimination:

```
>> linalg::wiedemann(A, b, _mult)
```

```
-- +-          -+        --
|  |  24 mod 29  |         |
|  |             |         |
|  |  21 mod 29  |, TRUE   |
|  |             |         |
|  |  17 mod 29  |         |
-- +-          -+        --
```

Example 2. Now let us define another matrix that has a special form:

```
>> MatZ29 := Dom::Matrix(Dom::IntegerMod(29)):
   A := MatZ29([[1, 0, 0], [0, 1, 2], [0, 0, 1]]);
   b := MatZ29(3, 1, [1, 2, 3]):
```

```
+-                              -+
|  1 mod 29, 0 mod 29, 0 mod 29  |
|                                |
|  0 mod 29, 1 mod 29, 2 mod 29  |
|                                |
|  0 mod 29, 0 mod 29, 1 mod 29  |
+-                              -+
```

For this particular matrix, it is easy to define an efficient multiplication method:

```
>> mult := proc(dummy, y)
   begin
       y[2]:=y[2]+2*y[3];
       y
   end:
   linalg::wiedemann(A, b, mult)
```

```
-- +-          -+        --
|  |   1 mod 29  |         |
|  |             |         |
|  |  25 mod 29  |, TRUE   |
|  |             |         |
|  |   3 mod 29  |         |
-- +-          -+        --
```

Background:

- The expected running time for the probabilistic algorithm is $O\left(n^2 + nM\right)$, and the running time for the deterministic variant is $O\left(n^2M\right)$ in the worst case, but only $O\left(n^2 + nM\right)$ on average. Here, M is the number of field operations that the matrix-vector multiplication routine `mult` uses.

- The basic idea of the algorithm is to solve a linear system $A\vec{x} = \vec{b}$ by finding the minimal polynomial $f(y)$ that solves $f(A)\vec{b} = \vec{0}$. If the constant coefficient $c = f(0)$ is nonzero and $g(y) := f(y) - c$, the equality $g(A)\vec{b} = -c\vec{b}$ implies that $\vec{x} = -\frac{1}{c}(g/y)(A)$ is the solution.

  The polynomial f is found by looking for the minimal polynomial h satisfying $\vec{u}h(A)\vec{b} = \vec{0}$ for some randomly chosen row vector $\vec{u}$. This may yield $h \neq f$ in unlucky cases, but in general the probability for this is small.

- Reference: Douglas Wiedemann: *Solving Sparse Linear equations over Finite Fields*, IEEE Transactions on Information Theory, vol. 32, no.1, Jan. 1986.

Changes:

- `linalg::wiedemann` is a new function.