`numeric` — library for numerical algorithms

Table of contents

Introduction

The `numeric` package provides algorithms from various areas of numerical mathematics.

The package functions are called using the package name `numeric` and the name of the function. E.g., use

```
>> numeric::solve(equations, unknowns)
```

to call the numerical solver. This mechanism avoids naming conflicts with other library functions. If this is found to be inconvenient, then the routines of the `numeric` package may be exported via `export`. E.g., after calling

```
>> export(numeric, fsolve)
```

the function `numeric::fsolve` may be called directly:

```
>> fsolve(equations, unknowns)
```

All routines of the `numeric` package are exported simultaneously by

```
>> export(numeric)
```

Note, however, that naming conflicts with the functions `indets`, `int`, `linsolve`, `rationalize`, `solve`, and `sort` of the standard library exist. The corresponding functions of the `numeric` package are not exported. Further, if the identifier `fsolve`, say, already has a value, then `export` returns a warning and does not export `numeric::fsolve`. The value of the identifier `fsolve` must be deleted before it can be exported successfully from the `numeric` package.

`numeric::butcher` – Butcher parameters of Runge-Kutta schemes

`numeric::butcher(method)` returns the Butcher parameters of the Runge-Kutta scheme named `method`.

Call(s):

- `numeric::butcher(method)`

Parameters:

`method` — name of the Runge-Kutta scheme, one of *`EULER1`*, *`RKF43`*, *`RK4`*, *`RKF34`*, *`RKF54a`*, *`RKF54b`*, *`DOPRI54`*,*`CK54`*, *`RKF45a`*, *`RKF45b`*, *`DOPRI45`*,*`CK45`*, *`BUTCHER6`*, *`RKF87`*, *`RKF78`*.

Return Value: A list `[s,c,a,b1,b2,order1,order2]` is returned.

Related Functions: `numeric::odesolve`

Details:

- An explicit s-stage Runge-Kutta method for the numerical integration of a dynamical system $dy/dt = f(t, y)$ with stepsize h is a map

$$(t, y) \to (t + h, y + h\,b_1\,k_1 + \cdots + h\,b_s\,k_s)$$

with "intermediate stages" $k_1, \ldots, k_s$ given by

$$\begin{aligned} k_1 &= f(t, y), \\ k_2 &= f(t + c_2\,h, y + h\,a_{21}\,k_1), \\ &\vdots \\ k_s &= f(t + c_s\,h, y + h\,a_{s1}\,k_1 + \cdots + h\,a_{s,s-1}\,k_{s-1}). \end{aligned}$$

Various numerical schemes arise from different choices of the Butcher parameters: the $s \times s$-matrix a_{ij}, the weights $b = [b_1, \ldots, b_s]$ and the abscissae $c = [0, c_2, \ldots, c_s]$.

Embedded pairs of Runge-Kutta methods consist of two methods that share the matrix a_{ij} and the abscissae c_i, but use different weights b_i.

- The returned list `[s,c,a,b1,b2,order1,order2]` are the Butcher data of the method: `s` is the number of stages, `c` is the list of abscissae, `a` is the (strictly lower) Butcher matrix, `b1` and `b2` are lists of weights. The integers `order1` and `order2` are the orders of the scheme when using the weights `b1` or `b2`, respectively, in conjunction with the matrix `a` and the abscissae `c`.

- The methods *EULER1* (order 1), *RK4* (order 4) and *BUTCHER6* (order 6) are single methods with `b1=b2` and `order1=order2`. All other methods are embedded pairs of Runge-Kutta-Fehlberg (`RKFxx`), Dormand-Prince (`DOPRIxx`) or Cash-Karp (`CKxx`) type. The names indicate the orders of the subprocesses, e.g., *CK45* is the Cash-Karp pair of orders 4 and 5. *CK54* is the same pair with reversed ordering of the subprocesses.
- These Butcher data are called by the routines `numeric::odesolve` and `numeric::odesolveGeometric`.

Example 1. The Butcher data of the classical 4 stage, 4th order Runge-Kutta scheme are:

```
>> numeric::butcher(RK4)
```

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

```
                                                                -
-
                                                                  |
                                                                  |
    +-                    -+  +-                    -+          |
    | 1/6, 1/3, 1/3, 1/6 |, | 1/6, 1/3, 1/3, 1/6 |, 4, 4   |
    +-                    -+  +-                    -+          |
                                                                  |
                                                                  |
                                                                -
-
```

Note that the weights `b1` and `b2` coincide: this classical method does not provide an embedded pair.

Example 2. The Butcher data of the embedded Runge-Kutta-Fehlberg pair *RKF34* of orders 3 and 4 are:

```
>> [s, c, a, b1, b2, order1, order2] := numeric::butcher(RKF34):
```

The number of stages `s` of the 4th order subprocess is 5, the abscissae `c` and the matrix `a` are given by:

```
>> s, c, a
```

```
    +-                    -+
 5, | 0, 1/4, 4/9, 6/7, 1 |,
    +-                    -+

    +-                                          -+
    |    0,       0,        0,       0,    0   |
    |                                            |
    |   1/4,      0,        0,       0,    0   |
    |                                            |
    |   4/81,   32/81,      0,       0,    0   |
    |                                            |
    |  57/98, -432/343, 1053/686,    0,    0   |
    |                                            |
    |   1/6,      0,      27/52,  49/156, 0   |
    +-                                          -+
```

Using these parameters with the weights

```
>> b1, b2
```

```
 +-                          -+
 | 1/6, 0, 27/52, 49/156, 0 |,
 +-                          -+

    +-                                      -+
    | 43/288, 0, 243/416, 343/1872, 1/12 |
    +-                                      -+
```

yields a numerical scheme of order 3 or 4, respectively:

```
>> order1, order2
```

```
                              3, 4
```

```
>> delete s, c, a, b1, b2, order1, order2:
```

Background:

- References:

  J.C. Butcher: The Numerical Analysis of Ordinary Differential Equations, Wiley, Chichester (1987).

  E. Hairer, S.P. Nørsett and G. Wanner: Solving Ordinary Differential Equations I, Springer, Berlin (1993).

Changes:

- The Cash-Karp pairs *CK45* and *CK54* were added.

`numeric::complexRound` – round a complex number towards the real or imaginary axis

`numeric::complexRound(z, ..)` discards small real or imaginary parts of complex floating point numbers `z`.

Call(s):

- `numeric::complexRound(z <, eps>)`

Parameters:

`z` — an arbitrary MuPAD object

Options:

`eps` — a real number $\geq 10^{-\texttt{DIGITS}}$.

Return Value: If `z` is a complex floating point number, then a real or complex floating point number is returned. For all other types `z` is returned unchanged.

Side Effects: The function is sensitive to the environment variable `DIGITS`.

Related Functions: `ceil`, `floor`, `frac`, `round`, `trunc`

Details:

- If the real part of `z` satisfies `Re(z)<eps*abs(z)`, then it is replaced by zero and `Im(z)*I` is returned.

  If the imaginary part of `z` satisfies `Im(z)<eps*abs(z)`, then it is replaced by zero and `Re(z)` is returned.

- With the default of `eps`$= 10^{-\texttt{DIGITS}}$ this rounding changes a complex floating point number by less than the relative standard precision.

- This function removes small real or imaginary parts of complex floating points numbers generated by numerical round-off. It is used to simplify the floating point output of `numeric::fsolve`, `numeric::polyroots`, `numeric::polysysroots` and `numeric::sum`.

Option <eps>:

- The default value is `eps`$= 10^{-\texttt{DIGITS}}$.
- Only precisions `eps`$\geq 10^{-\texttt{DIGITS}}$ are accepted.
- Numerical expressions such as `PI*sqrt(2)/10^10` etc. are accepted and converted to floats.

Example 1. Exact numbers are not changed:

```
>> numeric::complexRound(2 + I/10^20)

                  2 + 1/100000000000000000000 I
```

Also the following number has an exact imaginary part and is not rounded:

```
>> numeric::complexRound(2.0 + sqrt(2)*I/10^20)

                                          1/2
              1/100000000000000000000 I 2    + 2.0
```

Rounding occurs for complex floats, if this does not change its value significantly:

```
>> numeric::complexRound(1.0 + 2.0*I/10^10),
   numeric::complexRound(1.0 + 2.0*I/10^11)

                    1.0 + 0.0000000002 I, 1.0
```

Note that rounding is based on relative precision, i.e., only the ratio of real and imaginary parts is relevant:

```
>> numeric::complexRound((1.0 + 2.0*I)/10^100)

                     10.0e-101 + 2.0e-100 I

>> numeric::complexRound((1.0 + 1.0/10^11*I)/10^100)

                              10.0e-101
```

The relative precision for rounding may be reduced by the optional parameter `eps`:

```
>> numeric::complexRound(2.0/10^10 + I),
   numeric::complexRound(2.0/10^10 + I, PI/10^5)

                   0.0000000002 + 1.0 I, 1.0 I
```

Changes:

- `numeric::complexRound` is a new function.

`numeric::cubicSpline` – interpolation by cubic splines

`numeric::cubicSpline([x[0],y[0]], [x[1],y[1]], ..)` returns the cubic spline interpoland through a sequence of coordinate pairs $[x_i, y_i]$.

Call(s):

- `numeric::cubicSpline([x[0],y[0]], .. , [x[n],y[n]] <, BoundaryCondition> <, Symbolic> )`

Parameters:

`x[0],x[1],..,x[n]`	—	numerical real values in ascending order
`y[0],y[1],..,y[n]`	—	arbitrary expressions

Options:

`BoundaryCondition`	—	the type of the boundary condition: either *`NotAKnot`*, *`Natural`*, *`Periodic`*, or *`Complete`*`=[a,b]` with arbitrary arithmetical expressions `a,b`.
`Symbolic`	—	prevents conversion of the input data to floating point numbers. With this option symbolic abscissae x_i are accepted, which are assumed to be ordered.

Return Value: the spline interpoland: a MuPAD procedure.

Related Functions: `numeric::lagrange`

Details:

- The call

  `S:=numeric::cubicSpline([x[0],y[0]],..,[x[n],y[n]] <, Option>)`

  yields the cubic spline function `S` interpolating the data $[x_0, y_0], \ldots, [x_n, y_n]$, i.e, $S(x_i) = y_i$ for $i = 0, \ldots, n$. The spline function is a piecewise polynomial of degree ≤ 3 on the intervals $(-\infty, x_1], [x_1, x_2], \ldots, [x_{n-1}, \infty)$. `S` and its first two derivatives `S'`, `S''` are continuous at the points $x_1, \ldots, x_{n-1}$. Note that `S` extends the polynomial representation on $[x_0, x_1]$, $[x_{n-1}, x_n]$ to $(-\infty, x_1]$ and $[x_{n-1}, \infty)$, respectively.

- By default, `NotAKnot` boundary conditions are assumed, i.e., the third derivative `S'''` is continuous at the points x_1 and x_{n-1}. With this boundary condition `S` is a polynomial on the intervals $(-\infty, x_2]$ and $[x_{n-2}, \infty)$.

- By default, all input data are converted to floating point numbers. This conversion may be suppressed by the option `Symbolic`.

- Without the option `Symbolic` the abscissae x_i must be numerical real values in ascending order. If these data are not ordered, then `numeric::cubicSpline` reorders the abscissae internally, issuing a warning.

- The function `S` returned by `numeric::cubicSpline` may be called with one or two arguments.

  The call `S(z)` returns an explicit expression or a number, if `z` is a real number. Otherwise the unevaluated call `S(z)` is returned.

  The call `S(z,i)` is meant for symbolic arguments `z`. The argument `i` must be an integer. Internally, `z` is assumed to satisfy $x_i \le z \le x_{i+1}$ and `S(z,i)` returns a polynomial expression in `z` representing the spline function on this interval.

- If `S` is generated with symbolic abscissae x_i (necessarily using the option `Symbolic`), then the call `S(z)` with numerical `z` leads to an error. The call `S(z,i)` must be used for symbolic abscissae!

Option <`Symbolic`>:

- With this option no conversion of the input data to floating point numbers occurs.

- Symbolic abscissae x_i are accepted.

- The ordering $x_0 < x_1 < \cdots < x_n$ is assumed by `numeric::cubicSpline`. This ordering is not checked, even if the abscissae are numerical!

Option <`BoundaryCondition`>:

- With the default boundary condition `NotAKnot` the third derivative `S'''` of the spline function is continuous at the points x_1 and x_{n-1}. With this boundary condition `S` is a polynomial on the intervals $(-\infty, x_2]$ and $[x_{n-2}, \infty)$.

- The boundary condition `Natural` produces a spline function `S` satisfying $S''(x_0) = S''(x_n) = 0$.

- The boundary condition `Periodic` produces a spline function `S` satisfying $S(x_0) = S(x_n), S'(x_0) = S'(x_n), S''(x_0) = S''(x_n)$. With this option the input data y_0, y_n must coincide, otherwise an error occurs.

- The boundary condition `Complete=[a,b]` produces a spline function `S` satisfying $S'(x_0) = a$, $S'(x_n) = b$. Symbolic data `a,b` are accepted.

Example 1. We demonstrate some calls with numerical input data:

```
>> data := [i, sin(i*PI/20)] $ i= 0..40:
>> S1 := numeric::cubicSpline(data):
>> S2 := numeric::cubicSpline(data, Natural):
>> S3 := numeric::cubicSpline(data, Periodic):
>> S4 := numeric::cubicSpline(data, Complete = [3, PI]):
```

At the abscissae the corresponding input data are reproduced:

```
>> float(op(data, 6)[2]), S1(5), S2(5), S3(5), S4(5)

 0.7071067812, 0.7071067812, 0.7071067812, 0.7071067812,

    0.7071067812
```

Interpolation between the abscissae depends on the boundary condition:

```
>> S1(4.5), S2(4.5), S3(4.5), S4(4.5)

     0.6494470263, 0.6494470123, 0.6494469992, 0.6517696766
```

These are the cubic polynomials in `z` defining the spline on the interval $x_0 = 0 \le z \le x_1 = 1$:

```
>> expand(S1(z, 0)), expand(S2(z, 0)), expand(S3(z, 0)),
   expand(S4(z, 0))

                                      2                    3
 0.1570962007 z - 0.00002961951081 z  - 0.0006321161139 z ,

                                     3
    0.1570790998 z - 0.0006446347923 z ,

                                      2                     3
    0.157063067 z + 0.00002776961744 z  - 0.0006563716136 z ,

                          2               3
    3.0 z - 4.924083441 z  + 2.080517906 z

>> delete data, S1, S2, S3, S4:
```

Example 2. We demonstrate some calls with symbolic data:

```
>> S := numeric::cubicSpline([i, y.i] $ i=0..3):
>> S(1/2)

          0.3125 y0 + 0.9375 y1 - 0.3125 y2 + 0.0625 y3
```

This is the cubic polynomial in `z` defining the spline on the interval $x_0 = 0 \leq z \leq x_1 = 1$:

```
>> S(z, 0)

 y0 + z (3.0 y1 - 1.833333333 y0 - 1.5 y2 + 0.3333333333 y3 + z

    (1.0 y0 - 2.5 y1 + 2.0 y2 - 0.5 y3 +

    z (0.5 y1 - 0.1666666667 y0 - 0.5 y2 + 0.1666666667 y3)))
```

With the option *`Symbolic`* exact arithmetic is used:

```
>> S := numeric::cubicSpline([i, y.i] $ i=0..3, Symbolic):
>> S(1/2)

                   5 y0   15 y1   5 y2   y3
                   ---- + ----- - ---- + --
                    16     16      16    16
```

Also symbolic boundary data are accepted:

```
>> S := numeric::cubicSpline([i, exp(i)] $ i=0..10,
                             Complete = [a, b]):
>> S(0.1)

      0.08341154273 a + 0.00000005947817812 b + 1.020064753

>> S := numeric::cubicSpline([0, y0], [1, y1], [2, y2],
                             Symbolic, Complete=[a, 5]):
>> collect(S(z, 0), z)

             3 / 3 a   5 y0          3 y2       \
 y0 + a z + z  | --- + ---- - 2 y1 + ---- - 5/4 | +
               \  4     4             4         /

     2 /        9 y0   7 a   3 y2       \
    z  | 3 y1 - ---- - --- - ---- + 5/4 |
       \         4      4     4         /

>> delete S:
```

Example 3. We demonstrate the use of symbolic abscissae. Here the option *Symbolic* is mandatory.

```
>> S := numeric::cubicSpline([x.i, y.i] $ i=0..2, Symbolic):
```

The spline function S can only be called with 2 arguments. This is the cubic polynomial in z defining the spline on the interval $x_0 \le z \le x_1$:

```
>> S(z, 0)

                / (y1 - y0) (x1 - 2 x0 + x2)
 y0 + (z - x0) | -------------------------- -
                \    (x1 - x0) (x2 - x0)

    (x1 - x0) (y2 - y1)
    ------------------- + (z - x0)
    (x2 - x0) (x2 - x1)

    /       y2 - y1               y1 - y0       \ \
    | ------------------- - ------------------- | |
    \ (x2 - x0) (x2 - x1)   (x1 - x0) (x2 - x0) / /

>> delete S:
```

Example 4. Spline functions can be plotted:

```
>> S := numeric::cubicSpline([i, 1/(1 + i^2/100)] $ i=0..100):
>> plotfunc2d(S(x), x = 0..100, Ticks = [10, Steps = 0.2])
>> delete S:
```

Example 5. We demonstrate how to generate a phase plot of the differential equation $x''(t) + x(t)^3 = \sin(t)$, with initial conditions $x(0) = x'(0) = 0$. First, we use `numeric::odesolve` to compute a numerical mesh of solution points $[x_i, y_i] = [x(t_i), x'(t_i)]$ with $n+1$ equidistant time nodes $t_0, \ldots, t_n$ in the interval $[0, 20]$:

```
>> DIGITS := 4: n := 100:
>> for i from 0 to n do t[i] := 20/n*i: end_for:
>> f := (t, x) -> [x[2], sin(t) - x[1]^3]:
>> x[0] := 0: y[0] := 0:
>> for i from 1 to n do
     [x[i], y[i]] :=
     numeric::odesolve(t[i-1]..t[i], f, [x[i-1], y[i-1]]):
   end_for:
```

The mesh of the $(x(t), x'(t))$ phase plot consists of the following points:

```
>> Plotpoints := [point(x[i], y[i]) $ i=0..n]:
```

We wish to connect these points by a spline curve. We define a spline interpoland $Sx(t)$ approximating the solution $x(t)$ by interpolating the data $[t_0, x_0], \ldots, [t_n, x_n]$. A spline interpoland $Sy(t)$ approximating $x'(t)$ is obtained by interpolating the data $[t_0, y_0], \ldots, [t_n, y_n]$:

```
>> Sx := numeric::cubicSpline([t[i], x[i]] $ i=0..n):
>> Sy := numeric::cubicSpline([t[i], y[i]] $ i=0..n):
```

Finally, we plot the mesh points together with the interpolating spline curve:

```
>> plot2d([Mode = List, Plotpoints, PointWidth = 30],
          [Mode = Curve,[Sx(z), Sy(z)], z = [0, 20], Grid = [5*n]])
```

The function `plot::ode` serves for displaying numerical solutions of ODEs. In fact, it is implemented as indicated by the previous commands. The following call produces the same plot:

```
>> plot(plot::ode(
    [t[i] $ i=0..n], f, [x[0], y[0]],
    [(t, x) -> [x[1], x[2]], Style = Points, Color = RGB::Red],
    [(t, x) -> [x[1], x[2]], Style = Splines, Color = RGB::Blue])):

>> delete DIGITS, n, i, t, f, x, y, Plotpoints, Sx, Sy:
```

Changes:

- `numeric::cubicSpline` used to be `spline`.
- The routine was completely redesigned: the functionality was extended, performance improved.

`numeric::det` – determinant of a matrix

`numeric::det(A, ..)` returns the determinant of the matrix `A`.

Call(s):

- `numeric::det(A <, Symbolic> <, MinorExpansion>)`

Parameters:

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

Options:

Symbolic	— prevents conversion of input data to floats
MinorExpansion	— computes the determinant by a minor expansion along the first column

Return Value: By default the determinant is returned as a floating point number. With the option *Symbolic* an expression is returned.

Side Effects: Without the option *Symbolic* the function is sensitive to the environment variable DIGITS, which determines the numerical working precision.

Related Functions: linalg::det

Details:

- Without the option *Symbolic* all entries of A must be numerical. Numerical expressions such as exp(PI), sqrt(2) etc. are accepted and converted to floats. If symbolic entries are found in the matrix, then numeric::det automatically switches to *Symbolic*, issuing a warning.
- Option *Symbolic* should be used, if the matrix contains symbolic objects that cannot be converted to floating point numbers.
- Matrices A of a matrix domain such as Dom::Matrix(..) or Dom::SquareMatrix(..) are internally converted to arrays over expressions via A::dom::expr(A). Note that linalg::det must be used, when the determinant is to be computed over the component domain. Cf. example 2. Note that the option *Symbolic* should be used, if the entries cannot be converted to numerical expressions.

Option <*Symbolic*>:

- This option prevents conversion of the input data to floats. With this option symbolic entries are accepted.

Option <*MinorExpansion*>:

- With this option recursive minor expansion along the first column is used. This option may be useful for small matrices with symbolic entries.
- With this option symbolic entries are accepted, even if the option *Symbolic* is not used.

Example 1. Numerical matrices can be processed with or without the option *Symbolic*:

```
>> A := array(1..3, 1..3,[[1, 1, I], [1, exp(1), I], [1, 2, 2]]):

>> numeric::det(A), numeric::det(A, Symbolic)

      3.436563657 - 1.718281829 I, (2 - I) exp(1) - (2 - I)
```

Option *Symbolic* must be used, when the matrix has non-numerical entries:

```
>> A := array(1..2, 1..2, [[1/(x + 1),  1], [1/(x + 2), PI]]):

>> numeric::det(A, Symbolic)

                      2 PI - x + x PI - 1
                      -------------------
                                2
                         3 x + x  + 2
```

If the option *MinorExpansion* is used, then symbolic entries are accepted, even if the option *Symbolic* is not specified:

```
>> numeric::det(A, MinorExpansion),
   numeric::det(A, Symbolic, MinorExpansion)

             3.141592654     1.0     PI       1
             ----------- - -------, ----- - -----
               x + 1.0     x + 2.0  x + 1   x + 2

>> delete A:
```

Example 2. The following matrix has domain components:

```
>> A := Dom::Matrix(Dom::IntegerMod(7))([[6, -1], [1, 6]])

                     +-                   -+
                     |  6 mod 7, 6 mod 7  |
                     |                    |
                     |  1 mod 7, 6 mod 7  |
                     +-                   -+
```

Note that `numeric::det` computes the determinant of the following matrix:

```
>> A::dom::expr(A), numeric::det(A)

                        +-      -+
                        |  6, 6  |
                        |        |, 30.0
                        |  1, 6  |
                        +-      -+
```

The routine `linalg::det` must be used, if the determinant is to be computed over the component domain `Dom::IntegerMod(7)`:

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

$$2 \bmod 7$$

```
>> delete A:
```

Background:

- Without the option *`Symbolic`* QR-factorization of A via Householder transformations is used. With *`Symbolic`* LU-factorization of A is used.

Changes:

- Conversion of `Cat::Matrix` objects now uses the method `"expr"` of the matrix domain.

`numeric::eigenvalues` – numerical eigenvalues of a matrix

`numeric::eigenvalues(A)` returns numerical eigenvalues of the matrix `A`.

Call(s):

- `numeric::eigenvalues(A)`

Parameters:

`A` — a numerical square matrix of domain type `DOM_ARRAY` or of category `Cat::Matrix`

Return Value: an ordered list of numerical eigenvalues

Side Effects: The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Related Functions: `linalg::eigenvalues`, `linalg::eigenvectors`, `numeric::eigenvectors`, `numeric::singularvalues`, `numeric::singularvectors`, `numeric::spectralradius`

Details:

- All entries of `A` must be numerical. Numerical expressions such as `exp(PI)`, `sqrt(2)` etc. are accepted and converted to floats. Non-numerical symbolic entries lead to an error.
- Matrices `A` of a matrix domain such as `Dom::Matrix(..)` or `Dom::SquareMatrix(..)` are internally converted to arrays over expressions via `A::dom::expr(A)`. Note that `linalg::eigenvalues` must be used, when the eigenvalues are to be computed over the component domain. Cf. example 2.

- The eigenvalues are sorted by `numeric::sort`.
- Eigenvalues are approximated with an *absolute* precision of $10^{-DIGITS}\,r$, where r is the spectral radius of `A` (i.e., r is the maximum of the absolute values of the eigenvalues). Consequently, large eigenvalues should be computed correctly to `DIGITS` decimal places. The numerical approximations of the small eigenvalues are less accurate.

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

```
>> numeric::eigenvalues(linalg::hilbert(3))

          [0.002687340356, 0.1223270659, 1.408318927]
```

Precision goal and working precision are set by `DIGITS`:

```
>> A := array(1..3, 1..3,
              [[   I     ,     PI   ,  exp(1)  ],
               [   2     ,  10^100  ,    1     ],
               [10^(-100), 10^(-100), 10^(-100)]
              ]):
>> DIGITS := 10: numeric::eigenvalues(A)

                   [1.0 I, 5.0e-101, 10.0e99]
```

Note that small eigenvalues may be influenced by round-off. We increase the working precision. The previous numerical eigenvalue 5.0×10^{-101} is improved to $(1.0 + 2.718...\,I) \times 10^{-100}$:

```
>> DIGITS := 200: eigenvals := numeric::eigenvalues(A):
>> DIGITS := 10: eigenvals

 [- 6.283185307e-100 + 1.0 I, 1.0e-100 + 2.718281829e-100 I,

    1.0e100 + 2.031919862e-102 I]

>> delete A, eigenvals:
```

Example 2. The following matrix has domain components:

```
>> A := Dom::Matrix(Dom::IntegerMod(7))(
       [[6, -1, 4], [0,  3, 3], [0,  0, 3]])
```

```
                +-                           -+
                |  6 mod 7, 6 mod 7, 4 mod 7  |
                |                             |
                |  0 mod 7, 3 mod 7, 3 mod 7  |
                |                             |
                |  0 mod 7, 0 mod 7, 3 mod 7  |
                +-                           -+
```

Note that `numeric::eigenvalues` computes the eigenvalues of the following matrix:

```
>> A::dom::expr(A), numeric::eigenvalues(A)
```

```
                +-         -+
                |  6, 6, 4  |
                |           |
                |  0, 3, 3  |, [3.0, 3.0, 6.0]
                |           |
                |  0, 0, 3  |
                +-         -+
```

If the eigenvalues are to be computed over the component domain `Dom::IntegerMod(7)`, then `linalg::eigenvalues` should be used:

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

```
                 [[6 mod 7, 1], [3 mod 7, 2]]
```

```
>> delete A:
```

Background:

- The function implements standard numerical algorithms from the Handbook of Automatic Computation by Wilkinson and Reinsch.

Changes:

- Conversion of `Cat::Matrix` objects now uses the method `"expr"` of the matrix domain. Triangular matrices are now processed numerically.

`numeric::eigenvectors` – numerical eigenvalues and eigenvectors of a matrix

`numeric::eigenvectors(A, ..)` returns numerical eigenvalues and eigenvectors of the matrix `A`.

Call(s):

- `numeric::eigenvectors(A <, NoErrors>)`

Parameters:

`A` — a numerical square matrix of domain type `DOM_ARRAY` or of category `Cat::Matrix`

Options:

`NoErrors` — suppresses the computation of error estimates

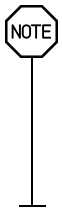
Return Value: a list `[d, X, res]`. The sorted list `d=[d[1],d[2],..]` contains the numerical eigenvalue. The array `X` is the matrix of eigenvectors, i.e., the *i*-th column of `X` is the eigenvector associated with the eigenvalue `d[i]`. The list of residues `res=[res[1],res[2],..]` provides error estimates for the numerical eigenvalues.

Side Effects: The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Related Functions: `linalg::eigenvalues`, `linalg::eigenvectors`, `numeric::eigenvalues`, `numeric::singularvalues`, `numeric::singularvectors`, `numeric::spectralradius`

Details:

- All entries of the matrix must be numerical. Numerical expressions such as `exp(PI), sqrt(2)` etc. are accepted and converted to floats. Non-numerical symbolic entries lead to an error.
- The eigenvalues are sorted by `numeric::sort`.
- Eigenvalues are approximated with an *absolute* precision of $10^{-DIGITS}\, r$, where r is the spectral radius of `A` (i.e., r is the maximum of the absolute values of the eigenvalues). Consequently, large eigenvalues should be computed correctly to `DIGITS` decimal places. The numerical approximations of the small eigenvalues are less accurate.



- The array `X` is the matrix of eigenvectors, i.e., the *i*-th column of `X` is a numerical eigenvector corresponding to the eigenvalue `d[i]`. Each column is either zero or normalized to the Euclidean length 1.0.

- For matrices with multiple eigenvalues and an insufficient number of eigenvectors some of the eigenvectors may coincide or may be zero, i.e., `X` is not necessarily invertible.

- The list of residues $res = [res_1, res_2, ..]$ provides some control over the quality of the numerical spectral data. The residues are given by

$$res_i = \|A\,x_i - d_i\,x_i\|_2\ ,$$

  where x_i is the normalized eigenvector (the i-th column of `X`) associated with the numerical eigenvalue d_i. For Hermitean matrices res_i provides an upper bound for the absolute error of d_i.

- With the option *`NoErrors`* the computation of the residues is suppressed, the returned value is `NIL`.

- Matrices `A` of a matrix domain such as `Dom::Matrix(..)` or `Dom::SquareMatrix(..)` are internally converted to arrays over expressions via `A::dom::expr(A)`. Note that `linalg::eigenvectors` must be used, when the eigenvalues/vectors are to be computed over the component domain. Cf. example 3.

- For a numerical algorithm it is not possible to distinguish between badly separated distinct eigenvalues and multiple eigenvalues. For this reason `numeric::eigenvectors` and `linalg::eigenvectors` use different return formats: the latter can provide information on the multiplicity of eigenvalues due to its internal exact arithmetic.

- Use `numeric::eigenvalues`, if only eigenvalues are to be computed.

Example 1. We compute the spectral data of the 2×2 Hilbert matrix:

```
>> A := linalg::hilbert(2)
```

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

```
>> [d, X, res] := numeric::eigenvectors(A):
```

The eigenvalues:

```
>> d
```

```
                    [0.06574145409, 1.267591879]
```

The eigenvectors:

```
>> X
```

```
                +-                                -+
                |   0.4718579255, -0.8816745988  |
                |                                 |
                |  -0.8816745988, -0.4718579255  |
                +-                                -+
```

Hilbert matrices are Hermitean, i.e., computing the spectral data is a numerically stable process. This is confirmed by the small residues:

```
>> res
```

```
                [3.965706585e-20, 5.421010863e-20]
```

For further processing we convert the data to matrices of the domain `Dom::Matrix()`:

```
>> M := Dom::Matrix(): X := M(X): d := M(2, 2, d, Diagonal):
```

We reconstruct the matrix from its spectral data:

```
>> X*d/X
```

```
                      +-                    -+
                      |  1.0,      0.5       |
                      |                      |
                      |  0.5, 0.3333333333   |
                      +-                    -+
```

We extract an eigenvector from the matrix `X`:

```
>> eigenvector1 := X::dom::col(X, 1)
```

```
                        +-                -+
                        |   0.4718579255  |
                        |                 |
                        |  -0.8816745988  |
                        +-                -+
```

```
>> delete A, d, X, res, M, eigenvector1:
```

Example 2. We demonstrate some numerically ill-conditioned cases. The following matrix has only one eigenvector and cannot be diagonalized. Only one numerical eigenvector is found:

```
>> A := array(1..2, 1..2, [[5, -1], [4, 1]]):
>> numeric::eigenvectors(A)
```

```
 --             +-                 -+                            -
-
 |              |  0, 0.4472135955  |                              |
 |  [3.0, 3.0], |                   |, [0.0, 1.084202173e-
19]  |
 |              |  0,  0.894427191  |                              |
 --             +-                 -+                            -
-
```

Dividing `A` by 3 leads to slightly increased internal round-off. This time two badly separated eigenvalues are computed. The two corresponding numerical eigenvectors are almost collinear. Both represent the same exact eigenvector:

```
>> B := map(A, _divide, 3):
>> numeric::eigenvectors(B)
```

```
 --                      +-                                -+
 |                       |  0.4472135954, -0.4472135957  |
 |  [0.9999999997, 1.0], |                                |,
 |                       |   0.894427191, -0.8944271909  |
 --                      +-                                -+

                                          --
                                           |
    [1.084202173e-19, 6.060874398e-20]     |
                                           |
                                          --
```

```
>> delete A:
```

Example 3. The following matrix has domain components:

```
>> A := Dom::Matrix(Dom::IntegerMod(7))([[6, -1], [0,  3]])
```

```
                     +-                  -+
                     |  6 mod 7, 6 mod 7  |
                     |                    |
                     |  0 mod 7, 3 mod 7  |
                     +-                  -+
```

Note that `numeric::eigenvectors` computes the spectral data of the following matrix:

```
>> A::dom::expr(A)
```

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

```
>> numeric::eigenvectors(A, NoErrors)
```

```
        --             +-                    -+      --
        |              |  -0.894427191, 1.0  |       |
        |  [3.0, 6.0], |                     |, NIL  |
        |              |  0.4472135955, 0.0  |       |
        --             +-                    -+      --
```

If the spectral data are to be computed over the component domain `Dom::IntegerMod(7)`, then `linalg::eigenvectors` should be used:

```
>> linalg::eigenvectors(A)
```

```
 -- --             -- +-          -+ -- --
 |  |              |  |  1 mod 7  |  |  |
 |  |  6 mod 7, 1, |  |           |  |  |,
 |  |              |  |  0 mod 7  |  |  |
 -- --             -- +-          -+ -- --

    --             -- +-          -+ -- -- --
    |              |  |  5 mod 7  |  |  |  |
    |  3 mod 7, 1, |  |           |  |  |  |
    |              |  |  1 mod 7  |  |  |  |
    --             -- +-          -+ -- -- --
```

```
>> delete A:
```

Background:

- The function implements standard numerical algorithms from the Handbook of Automatic Computation by Wilkinson and Reinsch.

Changes:

- Conversion of `Cat::Matrix` objects now uses the method `"expr"` of the matrix domain. Triangular matrices are now processed numerically.
- The eigenvector matrix is now returned as an array.

`numeric::expMatrix` – the exponential of a matrix

`numeric::expMatrix(A, ..)` returns the exponential $\exp(A)$ of a square matrix A.

`numeric::expMatrix(A, x, ..)` with a vector x returns the vector $\exp(A)\,x$.

`numeric::expMatrix(A, X, ..)` with a matrix X returns the matrix $\exp(A)\,X$.

Call(s):

- `numeric::expMatrix(A <, method>)`
- `numeric::expMatrix(A, x <, method>)`
- `numeric::expMatrix(A, X <, method>)`

Parameters:

`A` — a numerical square matrix of domain type `DOM_ARRAY` or of category `Cat::Matrix`

`x` — a vector represented by a list `[x[1],..,x[n]]` or a 1-dimensional array `array(1..n,[x[1],..,x[n]])`

`X` — an $n \times m$ matrix of domain type `DOM_ARRAY` or `Dom::Matrix(Ring)` with a suitable coefficient ring `Ring`

Options:

`method` — specifies the numerical method used for computing the result. The available methods are *`Diagonalization`*, *`Interpolation`*, *`Krylov`* and *`TaylorExpansion`*.

Return Value: All results are float matrices/vectors.

The call `numeric::expMatrix(A <, method>)` returns $\exp(A)$ as a matrix of domain type `DOM_ARRAY`.

The call `numeric::expMatrix(A, x, <, method>)` returns $\exp(A)\,x$ as a vector of the same domain type as the input vector `x`, i.e., either as a list or as a 1-dimensional array `array(1..n,[..])`.

The call `numeric::expMatrix(A, X, <, method>)` returns $\exp(A)\,X$ as an $n \times m$ matrix of domain type `DOM_ARRAY`.

Side Effects: The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Related Functions: `exp`

Details:

- The components of the matrix A must not contain symbolic objects which cannot be converted to numerical values via `float`. Numerical symbolic expressions such as `PI`, `sqrt(2)`, `exp(-1)` etc. are accepted. They are converted to floats.
- The methods *`Diagonalization`* and *`Interpolation`* do not work for all matrices (see below).
- Special algorithms are implemented for traceless 2×2 matrices and skew symmetric 3×3 matrices. Specification of a particular method does not have any effect on such matrices.

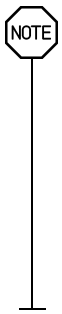
- If $\exp(A)\,x$ or $\exp(A)\,X$ is required, then you should not compute $\exp(A)$ first and then multiply the resulting matrix with the vector/matrix x/X. In general the call `numeric::expMatrix(A, x)/numeric::expMatrix(A, X)` is faster.

Option `<method>`:

- The method *`TaylorExpansion`* is the default algorithm. It produces fast results for matrices with small norms.

- The default method *`TaylorExpansion`* computes *each individual component* of $\exp(A)$, $\exp(A)\,x$, $\exp(A)\,X$ to a relative precision of about `10^(-DIGITS)`, unless numerical roundoff prevents reaching this precision goal. Roughly speaking: all digits of all components of the result are reliable up to roundoff effects.

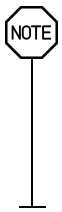
- The methods *`Diagonalization`*, *`Interpolation`* and *`Krylov`* compute the result to a relative precision w.r.t. the norm:



$$\| \mathit{error} \| \leq 10^{-\mathit{DIGITS}} \| \mathit{result} \| \,.$$

  Consequently, if the result has components of different orders of magnitude, then the smaller components have larger relative errors than the large components. Not all digits of the small components are reliable! Cf. example 2.

- The method *`Diagonalization`* only works for diagonalizable matrices. For matrices without a basis of eigenvectors `numeric::expMatrix` may either produce an error or the returned result is dominated by roundoff effects. For symmetric/Hermitean or skew/skew-Hermitean matrices this method produces reliable results.



- The method *`Interpolation`* may become numerically unstable for certain matrices. The algorithm tries to detect such unstabilities and stops with an error message.

- The method *`Krylov`* is only available for computing $\exp(A)\,x$ with a vector x. Also vectors represented by $n \times 1$ matrices are accepted.

  This method is fast when x is spanned by few eigenvectors of A. Further, if A has only few clusters of similar eigenvalues, then this method can be much faster than the other methods. Cf. example 3.

Example 1. We consider the matrix

```
>> A := array(1..2, 1..2, [[1, 0] , [1, PI]]):
>> expA := numeric::expMatrix(A)
```

```
+-                            -+
|  2.718281829,      0         |
|                              |
|  9.536085572, 23.14069263    |
+-                            -+
```

We consider a vector given by a list `x1` and by an equivalent 1-dimensional array `x2`, respectively:

```
>> x1 := [1, 1]: x2 := array(1..2, [1, 1]):
```

The constructor `M` of the matrix domain `Dom::Matrix()` converts the list `x1` to an equivalent 2×1 matrix `X` (a column):

```
>> M := Dom::Matrix(): X := M(x1):
```

The first call yields a list, the second a 1-dimensional array, the third a 2×1 array:

```
>> numeric::expMatrix(A, x1), numeric::expMatrix(A, x2, Krylov),
   numeric::expMatrix(A, X, Diagonalization)
```

```
                            +-                          -+
 [2.718281829, 32.6767782], | 2.718281829, 32.6767782 |,
                            +-                          -+

    +-              -+
    |  2.718281829   |
    |                |
    |   32.6767782   |
    +-              -+
```

For further processing the array `expA` is converted to an element of the matrix domain:

```
>> expA := M(expA):
```

Now the overloaded arithmetical operators `+`, `*`, `^` etc. can be used for further computations:

```
>> expA*X
```

```
                         +-              -+
                         |  2.718281829   |
                         |                |
                         |   32.6767782   |
                         +-              -+
```

```
>> delete A, expA, x1, x2, M, X:
```

Example 2. We demonstrate the different precision goals of the methods.

```
>> A := array(1..3, 1..3, [[ 1000,    1,    0 ],
                           [   0,     1,    1 ],
                           [1/10^100, 0, -1000]]):
```

The default method `TaylorExpansion` computes *each component* of $\exp(A)$ correctly:

```
>> numeric::expMatrix(A)

     +-                                                    -
+
     |  1.970071114e434, 1.972043157e431, 9.860215786e427  |
     |                                                     |
     |  9.860215786e327, 9.870085872e324, 4.935042936e321  |
     |                                                     |
     |   9.85035557e330, 9.860215786e327, 4.930107893e324  |
     +-                                                    -
+
```

The method `Diagonalization` produces a result, which is accurate in the sense that $\| \mathit{error} \| \leq 10^{-DIGITS} \| \exp(A) \|$ holds. Indeed, the largest components of $\exp(A)$ are correct. However, `Diagonalization` does not even get the right order of magnitude of the smaller components:

```
>> numeric::expMatrix(A, Diagonalization)

    +-                                                    -
+
    |  1.970071114e434, 1.972043157e431,         0         |
    |                                                      |
    |         0,          2.718281829,           0         |
    |                                                      |
    |         0,               0,        5.075958898e-435  |
    +-                                                    -
+
```

Note that $\exp(A)$ is very sensitive to small changes in A. After elimination of the small lower triangular element both methods yield the same result with correct digits for all entries:

```
>> B := array(1..3, 1..3, [[ 1000, 1,    0 ],
                           [   0 , 1,    1 ],
                           [   0 , 0, -1000]]):
>> numeric::expMatrix(B)
```

```
    +-                                                          -
+
    |  1.970071114e434, 1.972043157e431,  9.860215786e427  |
    |                                                          |
    |         0,          2.718281829,    0.002715566262   |
    |                                                          |
    |         0,               0,        5.075958897e-435  |
    +-                                                          -
+

>> numeric::expMatrix(B, Diagonalization)

    +-                                                          -
+
    |  1.970071114e434, 1.972043157e431,  9.860215786e427  |
    |                                                          |
    |        0.0,         2.718281829,    0.002715566262   |
    |                                                          |
    |        0.0,             0.0,       5.075958898e-435  |
    +-                                                          -
+

>> delete A, B:
```

Example 3. Hilbert matrices $H_{ij} = (i + j - 1)^{-1}$ have real positive eigenvalues. For large dimension most of these eigenvalues are small and may be regarded as a single cluster. Consequently, the option `Krylov` is useful:

```
>> numeric::expMatrix(linalg::hilbert(100), [1 $ 100], Krylov)

 [25.47080919, 18.59337041, ... , 2.863083064, 2.848538965]
```

Background:

- The method `TaylorExpansion` sums the usual Taylor series

$$\exp(A) = 1 + A + A^2/2 + \cdots$$

  in a suitable numerically stable way.

- The method `Diagonalization` computes $A = T\,\mathrm{diag}(e^{\lambda_1}, e^{\lambda_2}, \dots)\,T^{-1}$ by a diagonalization $A = T\,\mathrm{diag}(\lambda_1, \lambda_2, \dots)\,T^{-1}$.

- The method `Interpolation` computes a polynomial P interpolating the function exp at the eigenvalues of A. Evaluation of the matrix polynomial yields $\exp(A) = P(A)$.

- The method *Krylov* reduces A to a Hessenberg matrix H and computes an approximation of $\exp(A)\,x$ from $\exp(H)$. Depending on A and x the dimension of H may be smaller than the dimension of A. Reference: Y. Saad, "Analysis of some Krylov Subspace Approximations to the Matrix Exponential Operator", SIAM Journal of Numerical Analysis **29** (1992).
- `numeric::expMatrix` uses polynomial arithmetic to multiply matrices and vectors. Thus sparse matrices are handled efficiently based on MuPADs internal sparse representation of polynomials.

Changes:

- `numeric::expMatrix` is a new function.

`numeric::factorCholesky` – **Cholesky factorization of a matrix**

`numeric::factorCholesky(A, ..)` returns a Cholesky factorization $A = LL^H$ of a positive definite Hermitean matrix A.

`numeric::factorCholesky(A, Symmetric, ..)` returns a Cholesky factorization $A = LL^T$ of a symmetric matrix A.

Call(s):

- `numeric::factorCholesky(A <, Symmetric> <, Symbolic> <, NoCheck>)`

Parameters:

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

Options:

Symmetric — makes `numeric::factorCholesky` compute a symmetric factorization $A = LL^T$ rather than a Hermitean factorization $A = LL^H$

Symbolic — prevents `numeric::factorCholesky` from using floating point arithmetic

NoCheck — prevents `numeric::factorCholesky` from checking that the matrix is Hermitean and positive definite

Return Value: The lower triangular Cholesky factor L is returned as a matrix of domain type `DOM_ARRAY`. Its components are real or complex floats, unless the option *Symbolic* is used. Without the option *NoCheck* an error is returned, if the matrix is not Hermitean or not positive definite.

Side Effects: Without the option *Symbolic* the function is sensitive to the environment variable DIGITS, which determines the numerical working precision.

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

Details:

- The Cholesky factorization of a square Hermitean matrix is $A = LL^H$, where L is a regular complex lower triangular matrix and L^H is the Hermitean transpose of L (i.e., the complex conjugate of the transpose of L). Such a factorization only exists, if A is positive definite.
- By default a numerical factorization is computed. If the option *Symbolic* is not used, then all components of the matrix are converted to floating point numbers. In this case the matrix must not contain symbolic objects that cannot be converted to floats. Numerical symbolic expressions such as PI, sqrt(2), exp(-1) etc. are accepted.
- The Cholesky factor returned by numeric::factorCholesky is normalized such that its diagonal elements are real and positive.

Option <*Symmetric*>:

- The symmetric Cholesky factorization of a square symmetric matrix is $A = LL^T$, where L is a regular complex lower triangular matrix and L^T is the transpose of L. The matrix A does not have to be positive definite. Consequently, with the option *Symmetric* no internal check is performed whether A is positive definite. Note that the symmetric factorization with regular L does not exist for all matrices.
- For real symmetric positive definite matrices A the Cholesky factor L is real and the Hermitean factorization $A = LL^H$ coincides with the symmetric factorization $A = LL^T$.

Option <*Symbolic*>:

- This option prevents conversion of the matrix entries to floats. The usual arithmetic for MuPAD expressions is used. With this option the matrix A may contain symbolic objects. Note that the option *NoCheck* must be used for the Hermitean factorization when non-numerical symbolic objects are present.

Option <*NoCheck*>:

- Without the option *Symmetric* `numeric::factorCholesky` checks that the matrix A is Hermitean and positive definite. The option *NoCheck* may be used to suppress these checks. It must be used when the matrix contains symbolic objects. Elements in the upper triangular part of the matrix will never be touched by the algorithm!
- This option is dangerous! It returns a result for matrices that are not Hermitean or not positive definite (i.e., no Cholesky factorization exists)!

- This option has no effect when the option *Symmetric* is used.

Example 1. We consider the matrix

```
>> A := array(1..2, 1..2, [[1, I] , [-I, PI]]):
```

We compute a numerical factorization

```
>> numeric::factorCholesky(A)
```

```
                      +-                       -+
                      |    1.0,         0       |
                      |                         |
                      |  - 1.0 I, 1.46341814    |
                      +-                       -+
```

and a symbolic factorization:

```
>> L := numeric::factorCholesky(A, Symbolic, NoCheck)
```

```
                       +-                    -+
                       |   1,        0        |
                       |                      |
                       |                1/2   |
                       |  - I, (PI - 1)       |
                       +-                    -+
```

For further processing the Cholesky factor (of domain type `DOM_ARRAY`) is converted to an element of the matrix domain `Dom::Matrix()`:

```
>> L := Dom::Matrix()(L):
```

Now the overloaded arithmetical operators `+`, `*`, `^` etc. can be used for further computations:

```
>> L*linalg::transpose(map(L, conjugate))
```

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

```
>> delete A, L:
```

Example 2. The following matrix is not positive definite:

```
>> A := array(1..2, 1..2, [[-2, sqrt(2)], [sqrt(2), 1]]):
>> numeric::factorCholesky(A)

 Error: matrix is not positive definite within working precision\
  [numeric::factorCholesky]
```

However, a symmetric factorization with a complex Cholesky factor does exist:

```
>> numeric::factorCholesky(A, Symmetric)
```

```
                +-                              -+
                |  1.414213562 I,      0         |
                |                                |
                |     - 1.0 I,    1.414213562    |
                +-                              -+
```

```
>> delete A:
```

Example 3. The option *NoCheck* should be used, when the matrix contains symbolic objects:

```
>> assume(x > 0): assume(z > 0):
>> A := array(1..2, 1..2, [[x, conjugate(y)], [y, z]]):

>> numeric::factorCholesky(A, Symbolic, NoCheck)
```

```
                 +-                            -+
                 |   1/2                        |
                 |  x   ,          0            |
                 |                              |
                 |        /           2 \1/2    |
                 |   y    |     abs(y)  |       |
                 |  ----, | z - ------- |       |
                 |   1/2  \        x    /       |
                 |  x                           |
                 +-                            -+
```

Note that with *NoCheck* it is assumed that the matrix is Hermitean and positive definite! All upper triangular entries ignored. The following result implicitly assumes `u=conjugate(y)`:

```
>> A := array(1..2, 1..2, [[x, u], [y, z]]):
>> numeric::factorCholesky(A, Symbolic, NoCheck)
```

$$\begin{bmatrix} x^{1/2} & 0 \\ \frac{y}{x^{1/2}} & z - \frac{\mathrm{abs}(y)^2}{x} \end{bmatrix}$$

```
>> delete A:
```

Changes:

- The new option *Symmetric* allows factorization of in-definite symmetric matrices.
- The return type was changed to `DOM_ARRAY`.

`numeric::factorLU` – *LU* factorization of a matrix

`numeric::factorLU(A, ..)` returns a *LU* factorization $PA = LU$ of the matrix A.

Call(s):

- `numeric::factorLU(A <, Symbolic>)`

Parameters:

`A` — an $m \times n$ matrix of domain type `DOM_ARRAY` or of category `Cat::Matrix`

Options:

Symbolic — prevents `numeric::factorLU` from using floating point arithmetic

Return Value: A list `[L,U,p]` is returned. The matrices `L` and `U` are of domain type `DOM_ARRAY`, `p` is a list of integer numbers $1, \ldots, m$ representing the row exchanges in pivoting steps. The components of `L` and `U` are real or complex floats, unless the option *`Symbolic`* is used.

Side Effects: Without the optional argument *`Symbolic`* the function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

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

Details:

- The LU factorization of a real or complex $m \times n$ matrix is $PA = LU$. The $m \times m$ matrix L is lower triangular normalized to 1 along the diagonal. The $m \times n$ matrix U is upper triangular, i.e., $U_{ij} = 0$ for $j < i$. The list `p=[p[1],..,p[m]]` returned by `numeric::factorLU` is a permutation of the numbers $1, \ldots, m$ corresponding to row exchanges of A. It represents the $m \times m$ permutation matrix P:

$$P_{ij} = \delta_{p[i],j} = \begin{cases} 1, \text{ if } j = p[i], \\ 0, \text{ if } j \neq p[i], \end{cases}$$

  Multiplication of P with matrices and vectors is performed easily using the permutation list `p`:

  `Y[i,j]:=X[p[i],j]` defines the permutation $Y = PX$ of a matrix X,

  `y[i]:=x[p[i]]` defines the permutation $y = Px$ of a vector x.

- By default a numerical factorization with partial pivoting is computed. If the option *`Symbolic`* is not used, then all components of the matrix are converted to floating point numbers. In this case the matrix must not contain symbolic objects that cannot be converted to floats. Numerical symbolic expressions such as `PI`, `sqrt(2)`, `exp(-1)` etc. are accepted.

- The factorization depends on the pivoting strategy. The results obtained with/without the option *`Symbolic`* may differ. Cf. example 2.

Option <*`Symbolic`*>:

- This option prevents conversion of the matrix entries to floats. The usual arithmetic for MuPAD expressions is used. With this option the matrix A may contain symbolic objects.

- With this option no row exchanges are performed in the internal Gaussian elimination unless necessary.

Example 1. We consider the matrix

```
>> A := array(1..3, 1..3, [[1, 2, 3], [2, 4, 6], [4, 8, 9]]):
>> [L, U, p] := numeric::factorLU(A)
```

```
     -- +-            -+  +-                -+            -
-
     |  |    1,  0, 0  |  |  4.0, 8.0,  9.0  |            |
     |  |              |  |                  |            |
     |  |   0.5, 1, 0  |, |   0,   0,   1.5  |, [3, 2, 1] |
     |  |              |  |                  |            |
     |  |  0.25, 0, 1  |  |   0,   0,  0.75  |            |
     -- +-            -+  +-                -+            -
-
```

The factors (of domain type `DOM_ARRAY`) are converted to elements of the matrix domain `Dom::Matrix()` for further processing:

```
>> M := Dom::Matrix(): L := M(L): U := M(U):
```

Now the overloaded arithmetical operators `+`, `*`, `^` etc. can be used for further computations:

```
>> L*U
```

```
                        +-               -+
                        |  4.0, 8.0, 9.0  |
                        |                 |
                        |  2.0, 4.0, 6.0  |
                        |                 |
                        |  1.0, 2.0, 3.0  |
                        +-               -+
```

The product LU coincides with A after exchanging the rows according to the permutation `p`:

```
>> PA := array(1..3, 1..3, [[A[p[i], j] $ j=1..3] $ i=1..3])
```

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

```
>> delete A, L, U, p, M, PA:
```

Example 2. We consider a non-square matrix:

```
>> A := array(1..3, 1..2, [[3*I, 10], [I, 1], [I, 1]]):
>> numeric::factorLU(A)
```

$$\left[\begin{bmatrix} 1, & 0, & 0 \\ 3.0, & 1, & 0 \\ 1.0, & 0, & 1 \end{bmatrix}, \begin{bmatrix} 1.0\ I, & 1.0 \\ 0, & 7.0 \\ 0, & 0 \end{bmatrix}, [2, 1, 3]\right]$$

Note that the symbolic factorization is different, because a different pivoting strategy is used:

```
>> numeric::factorLU(A, Symbolic)
```

$$\left[\begin{bmatrix} 1, & 0, & 0 \\ 1/3, & 1, & 0 \\ 1/3, & 1, & 1 \end{bmatrix}, \begin{bmatrix} 3\ I, & 10 \\ 0, & -7/3 \\ 0, & 0 \end{bmatrix}, [1, 2, 3]\right]$$

```
>> delete A:
```

Changes:

- Factorization was extended to non-square matrices.
- The type of the returned matrix factors was changed to `DOM_ARRAY`.

`numeric::factorQR` – QR factorization of a matrix

`numeric::factorQR(A, ..)` returns a QR factorization $A = QR$ of the matrix A.

Call(s):

- `numeric::factorQR(A <, Symbolic>)`

Parameters:

`A` — an $m \times n$ matrix of domain type `DOM_ARRAY` or of category `Cat::Matrix`

Options:

Symbolic — prevents `numeric::factorQR` from using floating point arithmetic

Return Value: The list `[Q,R]` with `Q` and `R` of domain type `DOM_ARRAY` is returned. The components of `Q` and `R` are real or complex floats, unless the option *Symbolic* is used.

Side Effects: The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

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

Details:

- The QR factorization of a real/complex $m \times n$ matrix is $A = QR$, where the $m \times m$ matrix Q is orthogonal/unitary and the $m \times n$ matrix R is upper triangular (i.e., $R_{ij} = 0$ for $j < i$).
- By default a numerical factorization is computed. The matrix must not contain symbolic objects that cannot be converted to floats. Numerical symbolic expressions such as `PI`, `sqrt(2)`, `exp(-1)` etc. are accepted. They will be converted to floats, unless the option *Symbolic* is used.
- The R factor is normalized such that its diagonal elements R_{ii} with $i = 1, \ldots, \min(m, n)$ are real and nonnegative.

Option <*Symbolic*>:

- This option prevents conversion of the matrix entries to floats. The usual arithmetic for MuPAD expressions is used. With this option the matrix A may contain symbolic objects.

Example 1. We consider the matrix

```
>> A := array(1..2, 1..2, [[1, 0] , [1, PI]]):
```

First we compute a numerical factorization:

```
>> [Q1, R1] := numeric::factorQR(A)
```

```
-- +-                                -+
|  |  0.7071067812, -0.7071067812  |
|  |                                |,
|  |  0.7071067812,  0.7071067812  |
-- +-                                -+

   +-                            -+ --
   |  1.414213562, 2.221441469  |  |
   |                            |  |
   |       0,      2.221441469  |  |
   +-                            -+ --
```

Next the symbolic factorization is computed:

```
>> [Q2, R2] := numeric::factorQR(A, Symbolic)
```

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

For further processing the factors (of domain type `DOM_ARRAY`) are converted to elements of the matrix domain `Dom::Matrix()`:

```
>> M := Dom::Matrix():
>> Q1 := M(Q1): R1 := M(R1): Q2 := M(Q2): R2 := M(R2):
```

Now the overloaded arithmetical operators `+`, `*`, `^` etc. can be used for further computations:

```
>> Q1*R1, Q2*R2
```

```
             +-                        -+  +-        -+
             |  1.0, -4.33680869e-19    |  |  1,  0  |
             |                          |, |         |
             |  1.0,   3.141592654      |  |  1, PI  |
             +-                        -+  +-        -+
```

We finally verify the othogonality of the factors `Q1` and `Q2`:

```
>> Q1 * M::transpose(Q1), Q2 * M::transpose(Q2)
```

```
      +-                                        -+  +-      -+
      |         1.0,       -5.421010863e-20      |  |  1, 0  |
      |                                          |, |        |
      |  -5.421010863e-20,        1.0            |  |  0, 1  |
      +-                                        -+  +-      -+

>> delete A, Q1, R1, Q2, R2, M:
```

Example 2. We consider a non-square matrix of rank 1:

```
>> A := array(1..3, 1..2, [[0, 0], [I, 1], [I, 1]]):
>> numeric::factorQR(A, Symbolic)

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

Note that the numerical factorization yields different factors:

```
>> numeric::factorQR(A)

 -- +-                                                  -+
 |  |         0,       0.7071067812 I, 0.7071067812 I   |
 |  |                                                    |
 |  |  0.7071067812 I,       0.5,           -0.5        |,
 |  |                                                    |
 |  |  0.7071067812 I,      -0.5,            0.5        |
 -- +-                                                  -+

    +-                                -+ --
    |  1.414213562, - 1.414213562 I    |  |
    |                                  |  |
    |       0,             0           |  |
    |                                  |  |
    |       0,             0           |  |
    +-                                -+ --
```

```
>> delete A:
```

Background:

- Householder transformations are used to compute the numerical factorization. With the option *Symbolic* Gram-Schmidt orthonormalization of the columns of A is used.
- For an invertible square matrix A the QR factorization is unique up to to scaling factors of modulus 1. The normalization of R to real positive diagonal elements determines the factorization uniquely. Consequently, the results obtained with/without the option *Symbolic* coincide for invertible square matrices.
- For singular or non-square matrices the factorization is not unique and the results obtained with/without the option *Symbolic* may differ (cf. example 2).

Changes:

- Factorization was extended to non-square matrices.
- The type of the returned matrix factors was changed to DOM_ARRAY.

---

numeric::fft, numeric::invfft – Fast Fourier Transform

`numeric::fft(data, ..)` returns the discrete Fourier transform of the `data`.

`numeric::invfft(data, ..)` returns the inverse discrete Fourier transform.

Call(s):

- `numeric::fft(L <, Symbolic>)`
- `numeric::fft(A <, Symbolic>)`
- `numeric::invfft(L <, Symbolic>)`
- `numeric::invfft(A <, Symbolic>)`

Parameters:

- `L` — a list of arithmetical expressions. The length of the list must be an integer power of 2.
- `A` — a d-dimensional `array`$(1..n_1, \ldots, 1..n_d, [\ldots])$ of arithmetical expressions. The values $n_1, \ldots, n_d$ must all be integer powers of 2.

Options:

Symbolic — Without this option the floating point converter `float` is applied to all input data. Use this option if no such conversion is desired.

Return Value: a list/array of the same length/format as the first input parameter `L/A`.

Side Effects: Without the option *Symbolic* the function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Details:

- The 1-dimensional discrete Fourier transform $F = \texttt{fft}(L)$ of N data elements L_j stored in the list $L = [L_1, \ldots, L_N]$ is the list $F = [F_1, \ldots, F_N]$ given by

$$F_k = \sum_{j=1}^{N} L_j \, e^{-i\,2\,\pi\,(j-1)\,(k-1)/N} \;,\; k = 1, \ldots, N \;.$$

  The inverse transformation $L = \texttt{invfft}(F)$ is given by

$$L_j = \frac{1}{N} \sum_{k=1}^{N} F_k \, e^{i\,2\,\pi\,(j-1)\,(k-1)/N} \;,\; j = 1, \ldots, N.$$

  `fft` and `invfft` transform the data by the Fast Fourier Transform (FFT) algorithm with $O(N \log_2(N))$ operations.

- The d-dimensional discrete Fourier transform $F = \texttt{fft}(A)$ of $N = n_1 \times \cdots \times n_d$ data elements $(A_{j_1,\ldots,j_d})$ stored in the array A is the array $F = (F_{k_1,\ldots,k_d})$ given by

$$F_{k_1,\ldots,k_d} = \sum_{j_1=1}^{n_1} \cdots \sum_{j_d=1}^{n_d} A_{j_1,\ldots,j_d} \, e^{-i\,2\,\pi\left(\frac{(j_1-1)\,(k_1-1)}{n_1} + \cdots + \frac{(j_d-1)\,(k_d-1)}{n_d}\right)}$$

  with $k_1 = 1, \ldots, n_1 \;,\; \ldots \;,\; k_d = 1, \ldots, n_d$. The inverse transformation $A = \texttt{invfft}(F)$ is given by

$$A_{j_1,\ldots,j_d} = \frac{1}{N} \sum_{k_1=1}^{n_1} \cdots \sum_{k_d=1}^{n_d} F_{k_1,\ldots,k_d} \, e^{i\,2\,\pi\left(\frac{(j_1-1)\,(k_1-1)}{n_1} + \cdots + \frac{(j_d-1)\,(k_d-1)}{n_d}\right)}$$

  with $j_1 = 1, \ldots, n_1 \;,\; \ldots \;,\; j_d = 1, \ldots, n_d$. `fft` and `invfft` transform the data by the Fast Fourier Transform (FFT) algorithm with $O(N \log_2(N))$ operations.

Example 1. We give a demonstration of 1-dimensional transformations using lists. By default, numerical expressions are converted to floats:

```
>> L := [1, 2^(1/2), 3, PI]: numeric::fft(L)

 [8.555806216, - 2.0 + 1.727379091 I, -0.5558062159,

    - 2.0 - 1.727379091 I]

>> numeric::invfft(%)

 [1.0, 1.414213562 - 1.084202173e-19 I, 3.0,

    3.141592654 + 1.084202173e-19 I]
```

Exact arithmetic is used with the option *Symbolic*:

```
>> numeric::fft(L, Symbolic)

        1/2                 1/2           1/2
 [PI + 2    + 4, I PI - I 2    - 2, 4 - 2    - PI,

       1/2
    I 2    - I PI - 2]

>> numeric::invfft(%, Symbolic)

                            1/2
                      [1, 2   , 3, PI]
```

Symbolic expressions are accepted:

```
>> L := [x, 2, 3, x]: numeric::fft(L)

 [2 x + 5.0, (1.0 + 1.0 I) x - (3.0 + 2.0 I), 1.0,

    (1.0 - 1.0 I) x - (3.0 - 2.0 I)]

>> numeric::fft(L, Symbolic)

   [2 x + 5, (1 + I) x - (3 + 2 I), 1, (1 - I) x - (3 - 2 I)]

>> delete L:
```

Example 2. We give a demonstration of multi-dimensional transformations. First, a 2-dimensional transformation is computed by using an array with 2 indices:

```
>> A := array(1..2, 1..4, [[1, 2, 3, 4], [a, b, c, d]]):
```

```
>> numeric::fft(A, Symbolic)

            array(1..2, 1..4,
              (1, 1) = a + b + c + d + 10,
              (1, 2) = a - I b - c + I d - (2 - 2 I),
              (1, 3) = a - b + c - d - 2,
              (1, 4) = a + I b - c - I d - (2 + 2 I),
              (2, 1) = 10 - b - c - d - a,
              (2, 2) = I b - a + c - I d - (2 - 2 I),
              (2, 3) = b - a - c + d - 2,
              (2, 4) = c - I b - a + I d - (2 + 2 I)
            )

>> numeric::invfft(%, Symbolic)

                        +-            -+
                        |  1, 2, 3, 4  |
                        |              |
                        |  a, b, c, d  |
                        +-            -+
```

The next example is 3-dimensional as indicated by the format of the array:

```
>> A := array(1..2, 1..4, 1..2,
              [[[sin(j1*PI/2)*cos(j2*3*PI/4)*sin(j3*PI/2)
              $ j3 = 1..2 ] $ j2 = 1..4 ] $ j1 = 1..2]):
>> numeric::fft(A)

              array(1..2, 1..4, 1..2,
                (1, 1, 1) = -1.0,
                (1, 1, 2) = -1.0,
                (1, 2, 1) = - 1.414213562 - 1.0 I,
                (1, 2, 2) = - 1.414213562 - 1.0 I,
                (1, 3, 1) = 1.0,
                (1, 3, 2) = 1.0,
                (1, 4, 1) = - 1.414213562 + 1.0 I,
                (1, 4, 2) = - 1.414213562 + 1.0 I,
                (2, 1, 1) = -1.0,
                (2, 1, 2) = -1.0,
                (2, 2, 1) = - 1.414213562 - 1.0 I,
                (2, 2, 2) = - 1.414213562 - 1.0 I,
                (2, 3, 1) = 1.0,
                (2, 3, 2) = 1.0,
                (2, 4, 1) = - 1.414213562 + 1.0 I,
                (2, 4, 2) = - 1.414213562 + 1.0 I
              )

>> delete A:
```

Changes:

- These functions used to be called `fft` and `ifft`, respectively, in previous MuPAD versions.
- The length of input lists needs not be specified by a second parameter any more.
- Multi-dimensional transformations are now possible by using appropriately formatted input arrays.
- The option *`Symbolic`* was added.

`numeric::fMatrix` – functional calculus for numerical square matrices

`numeric::fMatrix(f, A, ..)` computes the matrix $f(A, ..)$ with a function f and a square matrix A.

Call(s):

- `numeric::fMatrix(f, A, p1, p2, ..)`

Parameters:

`f`	—	a procedure representing a scalar function $f : \mathbb{C} \mapsto \mathbb{C}$ or $f : \mathbb{C} \times P \times P \times \cdots \mapsto \mathbb{C}$, where P is a set of parameters
`A`	—	a square matrix of domain type `DOM_ARRAY` or of category `Cat::Matrix`
`p1, p2, ..`	—	arbitrary MuPAD objects accepted by f as parameters

Return Value: The matrix $f(A, ..)$ is returned as an array.

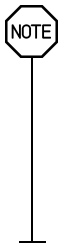
Side Effects: The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Related Functions: `numeric::expMatrix`, `numeric::inverse`

Details:

- The components of A must not contain symbolic objects which cannot be converted to numerical values via `float`. Numerical symbolic expressions such as `PI`, `sqrt(2)`, `exp(-1)` etc. are accepted. They are converted to floats.

- The matrix A must be diagonalizable. `numeric::fMatrix` aborts with an error message, if it detects numerically that A is not diagonalizable. For most non-diagonalizable matrices, however, the numerical algorithm fails to detect this fact and the returned matrix is dominated by roundoff effects. It is the user's responsibility to make sure that the diagonalization is feasible and well conditioned.



- Symmetric/Hermitean and skew/skew Hermitean matrices can always be diagonalized in a numerically stable way. `numeric::fMatrix` produces reliable numerical results for such matrices.

- The procedure f must accept complex floating point numbers as first argument. It may return arbitrary MuPAD expressions, provided these can be multiplied with floating point numbers.

- The parameters `p1, p2, ..` may be numerical or symbolic objects. They must be accepted by `f` as 2nd argument, 3rd argument etc.

- In contrast to the components of A, numerical symbolic objects such as `PI`, `sqrt(2)` etc. passed as parameters `p1, p2,..` are not converted to floats.

- Inversion or exponentiation of a matrix may be realized with the functions $f : a \mapsto 1/a$ and $f = \exp$, respectively. However, it is recommended to use the specialized algorithms `numeric::inverse` and `numeric::expMatrix` instead. Also matrix evaluation of low degree polynomials should be done with standard matrix arithmetic rather than with `numeric::fMatrix`.

Example 1. We compute the matrix power A^{100}:

```
>> A := array(1..2, 1..2, [[2, PI], [exp(-10), 0]]):
>> numeric::fMatrix(x -> x^100, A)
```

```
                +-                                  -+
                |  1.272133133e30, 1.998190806e30    |
                |                                    |
                |  2.887634784e25, 4.535724387e25    |
                +-                                  -+
```

Alternatively you may use the function `_power` which takes the exponent as a second parameter.

```
>> numeric::fMatrix(_power, A, 100)
```

```
>> delete A:
```

Example 2. We compute the square root of a matrix:

```
>> A := array(1..2, 1..2, [[0, 1], [-1, 0]]):
>> B := numeric::fMatrix(sqrt, A)
```

```
         array(1..2, 1..2,
           (1, 1) = 0.7071067812 - 1.084202173e-19 I,
           (1, 2) = 0.7071067812 + 2.710505431e-20 I,
           (2, 1) = - 0.7071067812 - 1.084202173e-19 I,
           (2, 2) = 0.7071067812 - 5.421010863e-20 I
         )
```

The small imaginary parts are caused by numerical roundoff. We eliminate them by extracting the real parts of the components:

```
>> B := map(B, Re)
```

```
               +-                                -+
               |   0.7071067812, 0.7071067812  |
               |                                 |
               |  -0.7071067812, 0.7071067812  |
               +-                                -+
```

We verify that `B^2` matrix is `A`. For convenience we convert `B` to an element of a matrix domain and compute the square by the overloaded operator `^`:

```
>> B := Dom::Matrix(Dom::Complex)(B): B^2
```

```
            +-                                      -+
            |  5.421010863e-20,        1.0          |
            |                                       |
            |        -1.0,      -1.084202173e-19    |
            +-                                      -+
```

This coincides with `A` up to numerical roundoff.

```
>> delete A, B:
```

Example 3. We compute $\exp(t\,\pi\,A)$ with a symbolic parameter t:

```
>> A := array(1..2,1..2,[[0,1],[-1,0]]):
>> numeric::fMatrix(exp@_mult, A, t*PI)
```

```
   array(1..2, 1..2,
     (1, 1) = 0.5 exp(-1.0 I t PI) + 0.5 exp(1.0 I t PI),
     (1, 2) = 0.5 I exp(-1.0 I t PI) - 0.5 I exp(1.0 I t PI),
     (2, 1) = 0.5 I exp(1.0 I t PI) - 0.5 I exp(-1.0 I t PI),
     (2, 2) = 0.5 exp(-1.0 I t PI) + 0.5 exp(1.0 I t PI)
   )
```

```
>> delete A:
```

Background:

- A numerical diagonalization $A = X\,\mathrm{diag}(\lambda_1, \lambda_2, \ldots)\,X^{-1}$ is computed. The columns of X are the (right) eigenvectors of A, the diagonal entries $\lambda_1, \lambda_2, \ldots$ are the corresponding eigenvalues. The function f is mapped to the eigenvalues, the matrix result is computed by

  $$f(A, p_1, p_2, ..) = X\,\mathrm{diag}(f(\lambda_1, p_1, p_2, ..), f(\lambda_2, p_1, p_2, ..), \ldots)\,X^{-1}\,.$$

  The eigenvector matrix X may be obtained via `numeric::eigenvectors(A)[2]`.

- The condition number $\|X\|\,\|X\hat{}-1\|$ of the eigenvector matrix is a measure indicating how well conditioned the diagonalization of the matrix A is. If this number is larger than 10^{DIGITS}, then not a single digit of the diagonalization data is trustworthy.

- The call `numeric::fMatrix(exp, A)` is equivalent to
  `numeric::expMatrix(A, Diagonalization)`

Changes:

- `numeric::fMatrix` is a new function.

`numeric::fsolve` – search for a numerical root of a system of equations

`numeric::fsolve(eqs, ..)` returns a numerical approximation of a solution of the system of equations `eqs`.

Call(s):

- `numeric::fsolve(eq, x <, Options>)`
- `numeric::fsolve(eq, x = a <, Options>)`
- `numeric::fsolve(eq, x = a..b <, Options>)`
- `numeric::fsolve(eqs, [x1, x2, ..] <, Options>)`
- `numeric::fsolve(eqs, {x1, x2, ..} <, Options>)`
- `numeric::fsolve(eqs, [x1 = a1, x2 = a2, ..] <, Options>)`
- `numeric::fsolve(eqs, {x1 = a1, x2 = a2, ..} <, Options>)`
- `numeric::fsolve(eqs, [x1 = a1..b1, x2 = a2..b2, ..] <, Options>)`
- `numeric::fsolve(eqs, {x1 = a1..b1, x2 = a2..b2, ..} <, Options>)`

Parameters:

`eq`	— an arithmetical expression or an equation in one indeterminate `x`. An expression `eq` is interpreted as the equation `eq = 0`.
`x`	— an identifier or an indexed identifier to be solved for.
`a`	— real or complex numerical starting value for the internal search. Typically, a crude approximation of the solution.
`a..b`	— a range of numerical values defining a search interval for the numerical root.
`eqs`	— a list or a set of expressions or equations in several indeterminates `x1, x2, ...`. Expressions are interpreted as homogeneous equations.
`x1, x2, ..`	— identifiers or indexed identifiers to be solved for.
`a1, a2, ..`	— real or complex numerical starting values for the internal search. Typically, crude approximations of solution.
`a1..b1, a2..b2, ..`	— ranges of numerical values defining search intervals for the numerical root.

Options:

`RestrictedSearch`	— makes `numeric::fsolve` return only numerical roots in the user-defined search range `x = a..b` and `[x1 = a1..b1, x2 = a2..b2, ..]`, respectively. This is the default search strategy, if a search range with real range parameters is specified for at least one of the unknowns.
`UnrestrictedSearch`	— allows `numeric::fsolve` to find and return solutions outside the specified search range. With this option, the search range is only used to choose random starting points for the internal numerical search.
`MultiSolutions`	— makes `numeric::fsolve` return all solutions found in the internal search
`Random`	— With this option, several calls to `numeric::fsolve` with the same input parameters may produce different roots.

Return Value: A single numerical root is returned as a list of equations `[x = value]` or `[x1 = value1, x2 = value2, ..]`, respectively. `FAIL` is returned, if no solution is found. With the option *`MultiSolutions`*, sequences of solutions may be returned.

Side Effects: The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Related Functions: `linsolve`, `numeric::linsolve`, `numeric::realroot`, `numeric::realroots`, `numeric::polyroots`, `numeric::polysysroots`, `numeric::solve`, `polylib::realroots`, `solve`

Details:

- This is MuPAD's numerical solver for non-linear systems of equations. By default, it returns only one numerical solution.
- The equations must not contain symbolic objects other than the unknowns that cannot be converted to numerical values via `float`. Symbolic objects such as `PI` or `sqrt(2)` etc. are accepted. The same holds true for starting values and search ranges. Search ranges may contain $\pm$`infinity`. Cf. example 2.
- `numeric::fsolve` implements a purely numerical Newton type root search with a working precision set by the environment variable `DIGITS`. Well separated simple roots should be exact within this precision. However, multiple roots or badly separated roots may be computed with a restricted precision. Cf. example 3.
- For systems of equations, the expressions defining the equations must have a symbolic derivative.
- Overdetermined systems (i.e., more equations than indeterminates) are not accepted. However, there may be more indeterminates than equations. Cf. example 4.
- Specifying indeterminates `[x1, x2, ..]` without starting values or search ranges is equivalent to the search ranges

  $$[\texttt{x1} = -\texttt{infinity}..\texttt{infinity},\ \texttt{x2} = -\texttt{infinity}..\texttt{infinity}, ..].$$

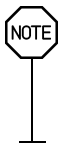
  Note, however, that the user should assist `numeric::fsolve` by providing specific search ranges whenever possible!
- For real equations and real starting points or search ranges, the internal Newton iteration will usually produce real values, i.e., `numeric::fsolve` searches for real roots only (unless square roots, logarithms etc. happen to produce complex values from real input). Use complex starting points or search ranges to search for complex roots of real equations. Cf. example 5.
- Starting values and search ranges can be mixed. Cf. example 6.

- Search ranges should only be provided, if a solution is known to exist inside the search range. Otherwise, the search may take some time before `numeric::fsolve` gives up.

- Specification of a search range primarily means that starting points from this range are used for the internal Newton search. For sufficiently small search ranges enclosing a solution the search will usually pick out this solution. However, it may also happen that the Newton iteration drifts towards other solutions. With the default search strategy *`RestrictedSearch`*, only solutions from the search range are accepted, even if solutions outside the search range are found internally. With the search strategy *`UnrestrictedSearch`*, any solution outside the search range is accepted and returned. Cf. example 7.

- If starting values for all indeterminates are provided, then a *single* Newton iteration with these initial data is launched. It either leads to a solution or `numeric::fsolve` gives up and returns `FAIL`. The same holds true if search ranges `x = a..a` or `[x1 = a1..a1, x2 = a2..a2, ..]` of zero length are specified.

  The risc of failure is high when providing bad starting values! Starting values are appropriate only if a sufficiently good approximation of the solution is known! On the other hand, providing good starting values is the fastest way to a solution. Cf. example 8.



- If at least one of the indeterminates has a non-trivial search range, then `numeric::fsolve` uses *several* Newton iterations with different starting values from the search range. Cf. example 9. Search ranges in conjunction with the option *`UnrestrictedSearch`* provide a higher chance of detecting roots than (bad) starting values!

- User defined assumptions such as `assume(x > 0)` are not taken into account in the numerical search! Provide search ranges instead! Cf. example 2.

- Convergence may be slow for multiple roots. Furthermore, `numeric::fsolve` may fail to detect such roots.

- `setuserinfo(numeric::fsolve, 3)` provides detailed information on the internal search.

- Use `linsolve` or `numeric::linsolve` for systems of *linear* equations. Use `numeric::realroots`, if *all real roots* of a single non-polynomial real equation in a finite range are desired. Use `polylib::realroots`, if *all real roots* of a real univariate polynomial are desired. Use `numeric::polyroots`, if *all real and complex roots* of a univariate polynomial are desired. Use `numeric::solve`, if *all roots* of a multivariate polynomial system are desired.

Option <`RestrictedSearch`>:

- This is the default search strategy, so there is no need to specify this option explicitly. With this search strategy, only solutions matching search ranges `x = a..b` or `[x1 = a1..b1, x2 = a2..b2, ..]` with real range parameters are returned. A real starting point is regarded as the search range `[-infinity, infinity]`. Complex starting points or complex range parameters automatically switch to the search strategy `UnrestrictedSearch`.

- Once a root with components $(r_1, r_2, \ldots)$ is found, it is checked whether $a_i \leq r_i \leq b_i$ is satisfied. If the root is not inside the search range, the search is continued. Note that solutions outside the search range may be found internally. These may be accessed with the option `MultiSolutions`. Cf. example 7.

Option <`UnrestrictedSearch`>:

- This option switches off the search strategy `RestrictedSearch`. With `UnrestrictedSearch`, `numeric::fsolve` stops its internal search whenever a root is found, even if the root is not inside the specified search range. Starting points for the internal Newton search are taken from the search range.

Option <`MultiSolutions`>:

- This option only has an effect when used with the default search strategy `RestrictedSearch`. A sequence of all roots found in the internal search is returned. Cf. example 7.

Option <`Random`>:

- With this option, random starting values are chosen for the internal search. Consequently, calling `numeric::fsolve` several times with the same parameters may lead to different solutions. This may be useful when several roots of one and the same equation or set of equations are desired.

Example 1. We compute roots of the sine function:

```
>> numeric::fsolve(sin(x) = 0, x)

                        [x = -226.1946711]
```

With the option *Random*, several calls may result in different roots:

```
>> numeric::fsolve(sin(x), x, Random)

                         [x = 97.38937226]

>> numeric::fsolve(sin(x), x, Random)

                         [x = 53.40707511]
```

Particular solutions can be chosen by an appropriate starting point close to the wanted solution, or by a search interval:

```
>> numeric::fsolve(sin(x), x = 3),
   numeric::fsolve(sin(x), x = -4..-3)

              [x = 3.141592653], [x = -3.141592654]
```

The solutions found by `numeric::fsolve` can be used in `subs` and `assign` to substitute or assign the indeterminates:

```
>> eqs := [x^2 = sin(y), y^2 = cos(x)]:
>> solution := numeric::fsolve(eqs, [x, y])

              [x = -0.8517004887, y = 0.8116062152]

>> eval(subs(eqs, solution))

   [0.7253937224 = 0.7253937224, 0.6587046485 = 0.6587046485]

>> assign(solution): x, y

                   -0.8517004887, 0.8116062152

>> delete eqs, solution, x, y:
```

Example 2. We demonstrate the use of search ranges. The following system has solutions with positive and negative `x`. The solution with `x > 0` is obtained with the search interval `x = 0..infinity`:

```
>> numeric::fsolve([x^2 = exp(x*y), x^2 = y^2],
                   [x = 0..infinity, y])

              [x = 0.7530891649, y = -0.753089165]

>> numeric::fsolve([x^2 = exp(x*y), x^2 = y^2],
                   [x = -infinity..0, y])

              [x = -0.753089165, y = 0.7530891649]
```

Example 3. Multiple roots can only be computed with a restricted precision:

```
>> numeric::fsolve(expand((x - 1/3)^5), x = 0.3)

                        [x = 0.3333929968]
```

Example 4. The following system of equations is degenerate and has a 1-parameter family of solutions. Each call to `numeric::fsolve` picks out one random solution:

```
>> numeric::fsolve([x^2 - y^2, x^2 - y^2], [x, y], Random) $ i=1..3

 [x = -140.1698476, y = 140.1698476],

    [x = 34.70258251, y = 34.70258251],

    [x = -29.16650501, y = 29.16650501]
```

The equation may also be specified as an underdetermined system:

```
>> numeric::fsolve([x^2 - y^2], [x, y])

                [x = -140.1698476, y = 140.1698476]
```

Example 5. The following equation has no real solution. Consequently, the numerical search with real starting values fails:

```
>> numeric::fsolve(sin(x) + cos(x)^2 = 3, x)

                                FAIL
```

With a complex starting value, a solution is found:

```
>> numeric::fsolve(sin(x) + cos(x)^2 = 3, x = I)

                [x = 0.2972513613 + 1.128383965 I]
```

Also complex search ranges may be specified. In the following, the internal starting point is a random value on the line from `2 + I` to `3 + I`:

```
>> numeric::fsolve(sin(x) + cos(x)^2 = 3, x = 2 + I..3 + I)

                 [x = 2.844341292 + 1.128383965 I]
```

Example 6. Starting values and search intervals can be mixed:

```
>> numeric::fsolve([x^2 + y^2 = 1, y^2 + z^2 = 1, x^2 + z^2 = 1],
                   [x = 1, y = 0..10, z])

     [x = 0.7071067812, y = 0.7071067812, z = 0.7071067812]
```

Example 7. With *UnrestrictedSearch*, search intervals are only used for choosing starting values for the internal Newton search. The numerical iteration may drift towards a solution outside the search range:

```
>> eqs := [x*sin(10*x) = y^3, y^2 = exp(-2*x/3)]:
>> numeric::fsolve(eqs, [x = 0..1, y = -1..0],
                   UnrestrictedSearch)

               [x = 1.232766202, y = -0.663038602]
```

With the default strategy *RestrictedSearch*, only solutions inside the search range are accepted:

```
>> numeric::fsolve(eqs, [x = 0..1, y = -1..0])

              [x = 0.9816416007, y = -0.7209295436]
```

In the last search, also the previous solution outside the search range was found. With the option *MultiSolutions*, `numeric::fsolve` returns a sequence of all solutions that were found in the internal search:

```
>> numeric::fsolve(eqs, [x = 0..1, y = -1..0], MultiSolutions)

 [x = 0.9816416007, y = -0.7209295436],

    [x = 1.232766202, y = -0.663038602]

>> delete eqs:
```

Example 8. Usually, most of the time is spent internally searching for some (crude) approximations of the root. If high precision roots are required, it is recommended to compute first approximations with moderate values of `DIGITS` and use them as starting values for a refined search:

```
>> eq := exp(-x) = x:
>> DIGITS := 10: firstApprox := numeric::fsolve(eq, x)

                        [x = 0.5671432904]
```

This output is suitable as input defining a starting value for x:

```
>> DIGITS := 1000: numeric::fsolve(eq, firstApprox)

 [x = 0.567143290409783872999968662210355549753 8...]

>> delete eq, firstApprox, DIGITS:
```

Example 9. Specifying starting values for the indeterminates launches a *single* Newton iteration. This may fail, if the starting values are not sufficiently close to the solution:

```
>> eq := [x*y = x + y - 4, x/y = x - y + 4]:
>> numeric::fsolve(eq, [x = 1, y = 1])

                                FAIL
```

If a search range is specified for at least one of the unknowns, then *several* Newton iterations with random starting values in the search range are used, until a solution is found or until `numeric::fsolve` gives up:

```
>> numeric::fsolve(eq, [x = 1, y = 0..10])

                 [x = 4.026449604e-14, y = 4.0]

>> delete eq:
```

Background:

- Internally the set of equations $f(x) = 0$ is solved by a modified Newton iteration $x \to x - t\,(f'(x))^{-1}\,f(x)$ with some adaptively chosen stepsize t. For degenerate or ill-conditioned Jacobians f' a minimization strategy for $\langle f, f\rangle$ is implemented. For scalar real equations, `numeric::realroot` is used, if a real finite search range is specified.

Changes:

- `numeric::fsolve` is a new function.
- The functionality of the function `numeric::fsolve` of previous MuPAD versions was moved to `numeric::realroots`.

`numeric::gldata` – weights and abscissae of Gauss-Legendre quadrature

`numeric::gldata(n, ..)` returns the weights and the abscissae of the Gauss-Legendre quadrature rule with n nodes.

Call(s):

- `numeric::gldata(n, digits)`

Parameters:

`n`	— the number of nodes: a positive integer
`digits`	— the number of decimal digits: a positive integer

Return Value: A list `[b,c]` is returned. The lists `b=[b[1],..,b[n]]` and `c=[c[1],..,c[n]]` are numerical approximations of the weights and abscissae with `digits` significant digits.

Side Effects: The function uses `option remember`. It is not sensitive to the environment variable `DIGITS`, because the numerical working precision is specified by the second argument `digits`.

Related Functions: `numeric::gtdata`, `numeric::ncdata`, `numeric::quadrature`

Details:

- The Gauss-Legendre quadrature rule $\sum_{i=1}^{n} b_i f(c_i)$ produces the exact integral $\int_0^1 f(x)\,dx$ for all polynomial integrands $f(x)$ through degree $2n-1$. The weights b_i and abscissae c_i are related to the roots of the n-th Legendre polynomial.
- The weights and abscissae are computed by a straightforward numerical algorithm with a working precision set by the argument `digits`. The resulting floating point numbers are correct to `digits` decimal places.
- Typically, the argument `digits` should coincide with the actual value of `DIGITS`.
- The data for `n=20,40,80,160` with `digits`≤ 200 are stored internally. They are returned immediately without any computational costs.
- Due to the internal remember mechanism only the first call to `numeric::gldata` leads to computational costs. For any further call with the same arguments the data are returned immediately.
- For odd n the abscissa $c_{(n+1)/2} = 1/2$ and the corresponding weight $b_{(n+1)/2}$ are rational numbers.

Example 1. The following call computes the Gaussian data with the default precision of `DIGITS=10` decimal digits:

```
>> numeric::gldata(4, DIGITS)

 [[0.1739274226, 0.3260725774, 0.3260725774, 0.1739274226],

    [0.0694318442, 0.3300094782, 0.6699905218, 0.9305681558]]
```

Example 2. For odd n exact rational data for $c_{(n+1)/2}$ and $b_{(n+1)/2}$ are returned:

```
>> DIGITS := 4: numeric::gldata(5, DIGITS)

 [[0.1185, 0.2393, 64/225, 0.2393, 0.1185],

    [0.04691, 0.2308, 1/2, 0.7692, 0.9531]]

>> delete DIGITS:
```

Background:

- The numerical integrator `numeric::quadrature` calls `numeric::gldata` to provide the data for Gaussian quadrature.

Changes:

- Efficiency was improved.

`numeric::gtdata` – weights and abscissae of Gauss-Tschebyscheff quadrature

`numeric::gtdata(n)` returns the weights and the abscissae of the Gauss-Tschebyscheff quadrature rule with n nodes.

Call(s):

- `numeric::gtdata(n)`

Parameters:

`n` — the number of nodes: a positive integer

Return Value: A list `[b,c]` is returned. The lists `b=[b[1],..,b[n]]` and `c=[c[1],..,c[n]]` are the exact weights and abscissae of the Gauss-Tschebyscheff quadrature rule, respectively.

Side Effects: The function uses `option remember`. It is not sensitive to the environment variable `DIGITS`.

Related Functions: `numeric::gldata`, `numeric::ncdata`, `numeric::quadrature`

Details:

- The Gauss-Tschebyscheff quadrature rule $\sum_{i=1}^{n} b_i f(c_i)$ produces the exact integral $\int_0^1 f(x)\,dx$ for all integrands of the form $f(x) = p(x)/\sqrt{x(1-x)}$ with polynomials $p(x)$ through degree $2n-1$.
- The exact weights $b = [b_1, \ldots, b_n]$ and abscissae $c = [c_1, \ldots, c_n]$ are given by

$$b_i = \frac{\pi}{2n}\ \sin\left(\frac{(2i-1)\,\pi}{2n}\right)\ , \quad c_i = \frac{1}{2}\ \left(1 + \cos\left(\frac{(2i-1)\,\pi}{2n}\right)\right)\ .$$

Example 1. The following call produces exact data for the quadrature rule with two nodes:

```
>> numeric::gtdata(2)

   -- --     1/2      1/2 --  --  1/2               1/2 --
--
   |  |  PI 2     PI 2     |  |  2                 2     |  |
   |  |  -------, -------  |, |  ---- + 1/2, 1/2 - ----  |  |
   -- --    8        8    --  --  4                  4   --
--
```

Background:

- The numerical integrator `numeric::quadrature` calls `numeric::gtdata` to provide the data for Gauss-Tschebyscheff quadrature.

Changes:

- No changes.

`numeric::indets` – search for indeterminates

`numeric::indets(object)` returns a set of the indeterminates contained in `object`.

Call(s):

- `numeric::indets(object)`

Parameters:

`object` — an arbitrary MuPAD object

Return Value: A set of indeterminates is returned, if the argument is an object of some basic data type of the kernel. The empty set is returned, if the object is from some library domain.

Related Functions: `indets`

Details:

- This is an auxiliary routine used by `numeric::polyroots`, `numeric::quadrature`, `numeric::realroots`, `numeric::solve` to find indeterminates.
- It recursively searches the operands of `object` for indeterminates. In particular, the search is applied to the elements of lists, sets, arrays, tables, etc.
- Following objects are regarded as indeterminates: identifiers, indexed identifiers and the indeterminates of `DOM_POLY` objects. Also coefficients of such polynomials are searched for indeterminates.
- Following objects are not regarded as indeterminates: the numerical constants `PI` and `EULER` (of type `Type::ConstantIdents`) and zero operands of expressions and subexpressions (i.e., the function names in unevaluated function calls such as `f(2)`). Also integration variables in unevaluated calls of `int` and `numeric::quadrature` and summation indices in unevaluated calls of `sum` are not considered.
- An object of a library domain, characterized by

  $$\texttt{domtype}(\texttt{extop}(\texttt{object}, 0)) = \texttt{DOM\_DOMAIN},$$

  is not searched for indeterminates. The empty set is returned. Cf. example 3.

Example 1. Identifiers, indexed identifiers are regarded as indeterminates:

```
>> numeric::indets([{a + b*PI}, sin(c + sqrt(2) + EULER),
                    table(1 = d - cos(e), 2 = f + 0.1*I),
                    array(1..2, [g, h]), F(i[2], i[2]),
                    D([1], G)(j[1]), k[3 + L[4]]])

        {a, b, c, d, e, f, g, h, i[2], j[1], k[L[4] + 3]}
```

Both indeterminates as well as symbolic coefficients are considered in polynomials of domain type `DOM_POLY`:

```
>> numeric::indets(poly(a[1]*x^2 + a[2]*x +a, [x, y]))

                        {a, x, y, a[1], a[2]}
```

Example 2. The zero operands of unevaluated function calls such as `f(..)` or `exp(..)` are not regarded as indeterminates:

```
>> numeric::indets(f(a + exp(b) + PI + EULER))

                                {a, b}
```

Integration variables and summation indices are not regarded as indeterminates:

```
>> numeric::indets({int(f(x), x = a..b),
                     sum(f(i), i = c..infinity)})

                              {a, b, c}
```

Example 3. Only objects of basic kernel types such as lists, sets, arrays, tables, expressions etc. are searched. This does not include matrices of domain type `matrix` or various polynomial types:

```
>> numeric::indets(Dom::Matrix()([a,b])),
   numeric::indets(Dom::DistributedPolynomial()(x^2 +a*x))

                                {}, {}
```

Changes:

- `numeric::indets` is a new function.

`numeric::inverse` – the inverse of a matrix

`numeric::inverse(A, ..)` returns the inverse of the matrix `A`.

Call(s):

- `numeric::inverse(A <, Symbolic>)`

Parameters:

A — a square matrix of domain type DOM_ARRAY or of category Cat::Matrix

Options:

Symbolic — prevents conversion of input data to floats

Return Value: a matrix of domain type DOM_ARRAY. FAIL is returned, if the inverse cannot be computed.

Side Effects: Without the option *Symbolic* the function is sensitive to the environment variable DIGITS, which determines the numerical working precision.

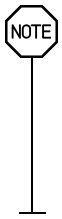
Related Functions: linsolve, linalg::matlinsolve, numeric::linsolve, numeric::matlinsolve, solve

---

Details:

- Option *Symbolic* should be used, if the matrix contains symbolic objects that cannot be converted to floating point numbers.
- Without the option *Symbolic* all entries of A must be numerical. Floating point arithmetic is used, the working precision is set by the environment variable DIGITS. etc. are accepted and converted to floats. If symbolic entries are found in the matrix, then numeric::inverse automatically switches to *Symbolic*, issuing a warning.
- Invertibility of the matrix can only be detected with exact arithmetic, i.e., using *Symbolic*. Cf. example 2.

- Matrices A of a matrix domain such as Dom::Matrix(..) or Dom::SquareMatrix(..) are internally converted to arrays over expressions via A::dom::expr(A). Note that 1/A must be used, when the inverse is to be computed over the component domain. Cf. example 3. Note that *Symbolic* should be used, if the entries cannot converted to numerical expressions.



- We recommend to use numeric::linsolve, if a system of linear equations is to be solved. In particular, this routine is more efficient than numeric::inverse for large sparse systems. It uses sparse input and output via symbolic equations and features internal sparse arithmetic.

Option <*Symbolic*>:

- This option prevents conversion of the input data to floats. With this option symbolic entries are accepted.
- This option should not be used for floating point matrices! No internal pivoting is used, unless necessary. Consequently, numerical instabilities may occur in floating point operations. Cf. example 4.

Example 1. Numerical matrices can be processed with or without the option *Symbolic*:

```
>> A := array(1..2, 1..2, [[1, 2], [3, PI]]):
>> numeric::inverse(A), numeric::inverse(A, Symbolic)
```

```
                                      +-                    -
+
                                      |     PI         2     |
   +-                              -+  |   ------,  - -----
-  |
   |  -1.099071012,  0.6996903372  |  |   PI - 6     PI - 6  |
   |                               |, |                      |
   |   1.049535506, -0.3498451686  |  |      3        1      |
   +-                              -+  |  - ------,  ------
|
                                      |    PI - 6   PI - 6   |
                                      +-                    -
+
```

Matrices of category `Cat::Matrix` are accepted. Note, however, that the inverse is returned as an array:

```
>> A := Dom::Matrix()([[2, PI], [0, 1]]):
>> numeric::inverse(A); domtype(%)
```

```
                     +-                    -+
                     |  0.5, -1.570796327  |
                     |                     |
                     |   0,       1.0      |
                     +-                    -+

                            DOM_ARRAY
```

```
>> delete A:
```

Example 2. The following matrix is not invertible:

```
>> A := array(1..2, 1..2, [[PI, PI^2], [PI^2, PI^3]]):
```

With exact arithmetic `numeric::inverse` detects this fact:

```
>> numeric::det(A, Symbolic), numeric::inverse(A, Symbolic)

                              0, FAIL
```

Due to internal round-off the matrix is regarded as invertible, if float arithmetic is used:

```
>> numeric::det(A), numeric::inverse(A)

                       +-                                    -
+
                       |   1.896479859e18, -6.036682882e17   |
     1.796747287e-17, |                                      |
                       |  -6.036682882e17,  1.921535841e17   |
                       +-                                    -
+

>> delete A:
```

Example 3. The following matrix has domain components:

```
>> A := Dom::Matrix(Dom::IntegerMod(7))([[6, -1], [1, 6]])

                     +-                  -+
                     |  6 mod 7, 6 mod 7  |
                     |                    |
                     |  1 mod 7, 6 mod 7  |
                     +-                  -+
```

Note that `numeric::inverse` computes the inverse of the following matrix:

```
>> A::dom::expr(A), numeric::inverse(A)

             +-      -+  +-                        -+
             |  6, 6  |  |        0.2,      -0.2   |
             |        |, |                         |
             |  1, 6  |  |  -0.03333333333,  0.2   |
             +-      -+  +-                        -+
```

The overloaded arithmetic should be used, if the inverse is to be computed over the component domain `Dom::IntegerMod(7)`:

```
>> 1/A
```

```
                    +-                  -+
                    |  3 mod 7, 4 mod 7  |
                    |                    |
                    |  3 mod 7, 3 mod 7  |
                    +-                  -+
```

```
>> delete A:
```

Example 4. The option *Symbolic* should not be used for float matrices, because no internal pivoting is used to stabilize the numerical algorithm:

```
>> A := array(1..2, 1..2, [[1.0/10^20, 1.0], [1.0, 1.0]]):
>> bad = numeric::inverse(A, Symbolic),
   good = numeric::inverse(A)
```

```
          +-                 -+         +-                  -
+
          |  0.0,    1.0      |         |  -1.0,    1.0      |
    bad = |                    |, good = |                    |
          |  1.0, -10.0e-21   |         |   1.0, -10.0e-21   |
          +-                 -+         +-                  -
+
```

```
>> delete A, bad, good:
```

Background:

- Gaussian elimination with partial pivoting is used. Partial pivoting is switched off by the option *Symbolic*.

Changes:

- Conversion of `Cat::Matrix` objects now uses the method `"expr"` of the matrix domain.
- The return type was changed to `DOM_ARRAY`.

`numeric::int` – numerical integration (the float attribute of int)

`numeric::int(f(x), x = a..b, ..)` computes a numerical approximation of $\int_a^b f(x)\,dx$.

Call(s):

- `numeric::int(f(x), x = a..b <, options>)`
- `float(hold(int)(f(x), x = a..b <, options>))`
- `float(freeze(int)(f(x), x = a..b <, options>))`

Parameters:

`f(x)` — expression in x
`x` — identifier or indexed identifier.
`a,b` — arbitrary expressions.

Options:

`options` — all options of `numeric::quadrature` can be used.

Return Value: a floating point number or an unevaluated `int(f(x), x = a..b <, options>)`, if the integral cannot be evaluated numerically.

Related Functions: `int`, `numeric::quadrature`

Details:

- The calls `numeric::int(arguments)`, `float(freeze(int)(arguments))` and `float(hold(int)(arguments))` are equivalent.
- The calls `numeric::int(arguments)` and `numeric::quadrature(arguments)` are almost equivalent: `numeric::int` calls `numeric::quadrature`. A numerical result produced by `numeric::quadrature` is returned as is. Otherwise `hold(int)(arguments)` is returned.
- See the help page of `numeric::quadrature` for details.

Example 1. We demonstrate some equivalent calls for numerical integration:

```
>> numeric::int(exp(x^2), x = -1..1),
   float(hold(int)(exp(x^2), x = -1..1)),
   float(freeze(int)(exp(x^2), x = -1..1)),
   numeric::quadrature(exp(x^2), x = -1..1)

      2.925303492, 2.925303492, 2.925303492, 2.925303492

>> numeric::int(max(1/10, cos(PI*x)), x = -2..0.0123),
   float(hold(int)(max(1/10, cos(PI*x)), x = -2..0.0123)),
   float(freeze(int)(max(1/10, cos(PI*x)), x = -2..0.0123)),
   numeric::quadrature(max(1/10, cos(PI*x)), x = -2..0.0123)

    0.7521024709, 0.7521024709, 0.7521024709, 0.7521024709
```

```
>> numeric::int(exp(-x^2), x = -2..infinity),
   float(hold(int)(exp(-x^2), x = -2..infinity)),
   float(freeze(int)(exp(-x^2), x = -2..infinity)),
   numeric::quadrature(exp(-x^2), x = -2..infinity)

       1.768308316, 1.768308316, 1.768308316, 1.768308316

>> numeric::int(sin(x)/x, x = -1..10, GaussLegendre = 5),
   float(hold(int)(sin(x)/x, x = -1..10, GaussLegendre = 5)),
   float(freeze(int)(sin(x)/x, x = -1..10, GaussLegendre = 5)),
   numeric::quadrature(sin(x)/x, x = -1..10, GaussLegendre = 5)

       2.604430665, 2.604430665, 2.604430665, 2.604430665
```

The calls `numeric::int(..)`, `float(hold(int)(..))` and `numeric::quadrature(..)` are equivalent in multiple numerical integrations, too:

```
>> numeric::int(numeric::int(x*y, x = 0..y), y = 0..1),
   numeric::int(numeric::quadrature(x*y, x = 0..y), y = 0..1),
   float(freeze(int)(numeric::int(x*y, x = 0..y), y = 0..1)),
   float(hold(int)(numeric::quadrature(x*y, x = 0..y), y = 0..1)),
   numeric::quadrature(numeric::int(x*y, x = 0..y), y = 0..1),
   numeric::quadrature(numeric::quadrature(x*y, x = 0..y), y = 0..1)

            0.125, 0.125, 0.125, 0.125, 0.125, 0.125
```

Example 2. The following integral do not exist. Consequently, numerical integration runs into problems:

```
>> numeric::quadrature(exp(x^2), x = 0..infinity)

 Error: Overflow/underflow in arithmetical operation;
 during evaluation of 'exp::float'
```

Note that `numeric::int` handles errors produced by `numeric::quadrature` and returns an unevaluated call to `int`:

```
>> numeric::int(exp(x^2), x = 0..infinity)

                              2
                  int(exp(x ), x = 0..infinity)
```

Changes:

- `numeric::int` used to be `numeric::fint`.
- This is a new function unifying the float attribute of `int` as well as `numeric::fint` of MuPAD 1.4. Symbolic analysis of the integrand was disabled, every aspect of the quadrature is now handled numerically.
- All options of `numeric::quadrature` are now allowed.

`numeric::lagrange` – polynomial interpolation

`numeric::lagrange` computes an interpolating polynomial through data over a rectangular grid.

Call(s):

- `numeric::lagrange(nodes, values, ind <, F>)`

Parameters:

`nodes` — a list $[L_1, \ldots, L_d]$ of d lists L_i defining a d-dimensional rectangular grid

$$\{[x_1, \ \ldots \ , x_d] \ ; \ x_1 \in L_1 \ , \ \ldots \ , \ x_d \in L_d\} \ .$$

The lists L_i may have different lengths $n_i = \texttt{nops}(L_i)$. The elements of each L_i must be distinct.

`values` — a d-dimensional `array`$(1..n_1, \ldots, 1..n_d, [\ldots])$ associating a value with each grid point:

$$[L_1[i_1], \ldots, L_d[i_d]] \quad \longrightarrow \quad \texttt{values}[i_1, \ldots, i_d] \ ,$$
$$i_1 = 1, .., n_1 \ , \ \ldots \ , \ i_d = 1, .., n_d \ .$$

`ind` — a list of d indeterminates or arithmetical expressions. Indeterminates are either identifiers (of domain type `DOM_IDENT`) or indexed identifiers (of type `"_index"`).

Options:

`F` — either *`Expr`* or any field of category `Cat::Field`

Return Value: An interpolating polynomial P of domain type `DOM_POLY` in the indeterminates specified by `ind` over the coefficient field `F` is returned. The elements in `ind` that are not indeterminates but arithmetical expressions are not used as indeterminates in P, but enter its coefficients: the polynomial is "evaluated" at these points. If no element of `ind` is an indeterminate, then the value of the polynomial at the point specified by `ind` is returned. This is an element of the field `F` or an expression, if `F=`*`Expr`*.

Related Functions: `genpoly`, `numeric::cubicSpline`, `poly`

Details:

- Assume that indeterminates `ind`$= [X_1, \ldots, X_d]$ are specified. The interpolating polynomial $P =$ `poly`$(\cdots, [X_1, \ldots, X_d], F)$ satisfies

$$\texttt{evalp}(P, X_1 = L_1[i_1], \ldots, X_d = L_d[i_d]) = \texttt{value}[i_1, \ldots, i_d]$$

  for all points $[L_1[i_1], \ldots, L_d[i_d]]$ in the grid. P is the polynomial of minimal degree satisfying the interpolation conditions, i.e., `degree`$(P, X_i) < n_i$.

- If only interpolating values at concrete numerical points $X_1 = v_1, \ldots, X_d = v_d$ are required, then we recommend not to compute P with symbolic indeterminates `ind`$= [X_1, \ldots, X_d]$ and then evaluate $P(v_1, \ldots, v_d)$. It is faster to compute this value directly by `numeric::lagrange` with `ind`$= [v_1, \ldots, v_d]$. Cf. examples 1 and 3.

Option <F>:

- The returned polynomial is of type `poly(.., F)`.

- For the default field *Expr* all input data may be arbitrary MuPAD expressions. Standard arithmetic over such expressions is used to compute the polynomial.

- For `F`$\neq$*Expr* the grid nodes as well as the entries of `values` must be elements of `F` or must be convertible to such elements. Conversion of the input data to elements of `F` is done automatically.

Example 1. We consider a 1-dimensional interpolation problem. To each node x_i a value y_i is associated. The interpolation polynomial P with $P(x_i) = y_i$ is:

```
>> L1 := [1, 2, 3]:
   values := array(1..3, [y1, y2, y3]):
   P := numeric::lagrange([L1], values, [X])
```

```
     / / y1        y3 \  2   /   5 y1          3 y3 \
 poly| | -- - y2 + -- | X  + | - ---- + 4 y2 - ---- | X +
     \ \ 2         2  /      \    2             2   /

                              \
    (3 y1 - 3 y2 + y3), [X] |
                              /
```

The evaluation of P at the point $X = 5/2$ is given by:

```
>> evalp(P, X = 5/2)
```

$$\frac{3\,y2}{4} - \frac{y1}{8} + \frac{3\,y3}{8}$$

It can also be computed directly without the symbolic polynomial:

```
>> numeric::lagrange([L1], values, [5/2])
```

$$\frac{3\,y2}{4} - \frac{y1}{8} + \frac{3\,y3}{8}$$

```
>> delete L1, values, P:
```

Example 2. We demonstrate multi-dimensional interpolation. Consider data over the following 2×3 grid:

```
>> XList := [1, 2]: YList := [1, 2, 3]:
   values := array(1..2, 1..3, [[1, 2, 3], [3, 2, 1]]):
   P := numeric::lagrange([XList, YList], values, [X, Y])

              poly(- 2 X Y + 4 X + 3 Y - 4, [X, Y])
```

Next, interpolation over a $2 \times 3 \times 2$ grid is demonstrated:

```
>> L1 := [1, 2]: L2 := [1, 2, 3]: L3 := [1, 2]:
   values := array(1..2, 1..3, 1..2,
      [[[1, 4], [1, 2], [3, 3]], [[1, 4], [1, 3], [4, 0]]]):
   numeric::lagrange([L1, L2, L3], values, [X, Y, Z])

             2           2
 poly(- 3 X Y  Z + 7/2 X Y  + 10 X Y Z - 23/2 X Y - 7 X Z +

               2        2
    8 X + 7/2 Y  Z - 3 Y  - 27/2 Y Z + 12 Y + 13 Z - 11,

    [X, Y, Z])
```

```
>> delete XList, P, L1, L2, L2, values:
```

Example 3. We interpolate data over a 2-dimensional grid:

```
>> n1 := 4: L1 := [i $ i = 1..n1]:
   n2 := 5: L2 := [i $ i = 1..n2]:
   f := (X, Y) -> 1/(1 + X^2 + Y^2):
   values := array(1..n1, 1..n2,
                   [[f(L1[i], L2[j]) $ j=1..n2] $ i=1..n1]):
```

First we compute the symbolic polynomial:

```
>> P := numeric::lagrange([L1, L2], values, [X, Y])

                        3  4                   3  3
 poly(- 5563/23108085 X  Y  + 16376/4621617 X  Y  -

    ... -

    4401895/3081078 Y + 4199983/2567565, [X, Y])
```

Fixing the value $Y = 2.5$ this yields a polynomial in X. It can also be computed directly by using an evaluation point for the indeterminate Y:

```
>> numeric::lagrange([L1, L2], values, [X, 2.5])

                        3                    2
 poly(0.0007372500794 X  - 0.002155538175 X  -

    0.03076935248 X + 0.1533997618, [X])
```

If all indeterminates are replaced by evaluation points, then the corresponding interpolation value is returned:

```
>> numeric::lagrange([L1, L2], values, [1.2, 2.5])

                           0.1146465319
```

```
>> delete n1, n2, f, values, P:
```

Example 4. We demonstrate interpolation over a non-standard field. Consider the following data over a 2×3 grid:

```
>> XList := [3, 4]: YList := [1, 2, 3]:
   values := array(1..2, 1..3, [[0, 1, 2], [3, 2, 1]]):
```

With the following call these data are converted to integers modulo 7. Arithmetic over this field is used:

```
>> F := Dom::IntegerMod(7):
   P := numeric::lagrange([XList, YList], values, [X, Y], F)
```

```
poly(5 X Y + 5 X + 5, [X, Y], Dom::IntegerMod(7))
```

Evaluation of P at grid points reproduces the associated values converted to the field:

```
>> evalp(P, X = XList[2], Y = YList[3]) = F(values[2, 3])
                          1 mod 7 = 1 mod 7
>> delete XList, YList, values, F, P:
```

Background:

- For a d-dimensional rectangular grid

  $$\{[x_1, \ldots, x_d]\,;\; x_1 \in L_1\,,\; \ldots\,,\; x_d \in L_d\}$$

  specified by the lists

  $$L_j = [x_{j1}, \ldots, x_{jn_j}]\,, \quad j = 1, \ldots, d$$

  with associated values

  $$P(x_{1i_1}, \ldots, x_{di_d}) = v_{i_1,\ldots,i_d}$$

  the interpolating polynomial in the indeterminates $X_1, \ldots, X_d$ is given by

  $$P(X_1, \ldots, X_d) = \sum_{i_1=1}^{n_1} \cdots \sum_{i_d=1}^{n_d} v_{i_1,\ldots,i_d} \times p_{1i_1}(X_1) \times \cdots \times p_{di_d}(X_d)$$

  with the Lagrange polynomials

  $$p_{jk}(X) = \prod_{\substack{l=1,\ldots,n_j \\ l \neq k}} \frac{X - x_{jl}}{x_{jk} - x_{jl}}\,, \quad j = 1, \ldots, d, \quad k = 1, \ldots, n_j$$

  associated with the k-th node of the j-th coordinate.

Changes:

- `numeric::lagrange` used to be `lagrange`.
- Also 1-dimensional grids must now be specified as a list with a list of nodes. The values must now be specified by an array. Also for 1-dimensional interpolation the indeterminate must now be specified by a list.
- Interpolation is now possible over grids of arbitrary dimension. Further, it is now possible to specify indeterminates as well as evaluation points.

`numeric::linsolve` – solve a system of linear equations

`numeric::linsolve(eqs, ..)` solves a system of linear equations.

Call(s):

- `numeric::linsolve(eqs <, vars> <, Symbolic> <, ShowAssumptions>)`

Parameters:

`eqs` — a list or set of linear equations or expressions

Options:

`vars`	— a list or set of unknowns to solve for. Unknowns may be identifiers or indexed identifiers or expressions.
`Symbolic`	— prevents conversion of input data to floating point numbers.
`ShowAssumptions`	— returns information on internal assumptions on symbolic parameters in `eqs`.

Return Value: Without the option `ShowAssumptions` a list of simplified equations is returned. It represents the general solution of the system `eqs`. `FAIL` is returned, if the system is not solvable.

With `ShowAssumptions` a list `[Solution, Constraints, Pivots]` is returned. `Solution` is a list of simplified equations representing the general solution of `eqs`. The lists `Constraints` and `Pivots` contain equations and inequalities involving symbolic parameters in `eqs`. Internally these were assumed to hold true when solving the system. `[FAIL,[],[]]` is returned, if the system is not solvable.

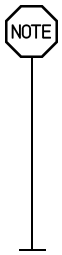
Side Effects: Without the option `Symbolic` the function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Related Functions: `linalg::matlinsolve`, `linsolve`, `numeric::fsolve`, `numeric::inverse`, `numeric::matlinsolve`, `numeric::polyroots`, `numeric::polysysroots`, `numeric::realroots`, `polylib::realroots`, `solve`

Details:

- `numeric::linsolve` is a fast numerical linear solver. It is also a recommended solver for linear systems with exact or symbolic coefficients (using `Symbolic`).
- Expressions are interpreted as homogeneous equations. E.g., the input `[x=y-1,x-y]` is interpreted as the system of equations `[x=y-1,x-y=0]`.

- Without the option *Symbolic* the input data are converted to floating point numbers. The coefficient matrix A of the system $A\,x = b$ represented by `eqs` must not contain non-convertible parameters, unless the option *Symbolic* is used! If such objects are found, then `numeric::linsolve` automatically switches to its symbolic mode, issuing a warning. Symbolic parameters in the "right hand side" b are accepted without warning.



- The numerical working precision is set by the environment variable `DIGITS`.

- The solutions are returned as a list of solved equations of the form

$$[x_1 = value_1, x_2 = value_2, \ldots]\,,$$

  where $x_1, x_2, \ldots$ are the unknowns. These simplified equations should be regarded as constraints on the unknowns. E.g., if an unknown x_1, say, does not turn up in the form $[x_1 = \ldots, \ldots]$ in the solution, then there is no constraint on this unknown and it is an arbitrary parameter. Generally, all unknowns that do not turn up on the left hand side of the solved equations are arbitrary parameters spanning the solution space. Cf. example 9.

  In particular, if the empty list is returned as the solution, then there are no constraints whatsoever on the unknowns, i.e., the system is trivial.

- The ordering of the solved equations corresponds to the ordering of the unknowns `vars`. It is recommended that the user specifies `vars` by a *a list* of unknowns. This guarantees that the solved equations are returned in the expected order. If `vars` are specified by a set, or if no `vars` are specified at all, then an internal ordering is used.

- `numeric::linsolve` returns the general solution of the system `eqs`. It is valid for arbitrary complex values of the symbolic parameters which may be present in `eqs`. If no such solution exists, then `FAIL` is returned. Solutions that are valid only for special values of the symbolic parameters may be obtained with the option *ShowAssumptions*. Cf. examples 2, 3, 4, 10.

- The solved equations representing the solution are suitable as input for `assign` and `subs`. Cf. example 8.

- `numeric::linsolve` is suitable for solving large sparse systems. Cf. example 6.

- If `eqs` represents a system with a banded coefficient matrix, then this is detected and used by `numeric::linsolve`. Note that in this case it is important to enter `eqs` as a list and to specify the unknowns as a list in order to guarantee the desired form of the coefficient matrix. The elements of sets may be reordered internally, leading to a loss of band structure and, consequently, of efficiency. Cf. example 6.

- `numeric::linsolve` is tuned for speed. For this reason it does not check systematically that the equations `eqs` are indeed linear in the unknowns! For non-linear equations strange things may happen, `numeric::linsolve` might even return wrong results! Cf. example 5.

- `numeric::linsolve` does not react to any assumptions on the unknowns or on symbolic parameters that are set via `assume`.

- Gaussian elimination with partial pivoting is used. Without the option *`Symbolic`*, floating point arithmetic is used and the pivoting strategy takes care of numerical stabilization. With *`Symbolic`* exact data are assumed and the pivoting strategy tries do maximize speed, not taking care of numerical stabilization! Cf. example 7.

Option <`vars`>:

- If no unknowns are specified by `vars`, then `numeric::linsolve` solves for all symbolic objects in `eqs`. The unknowns are determined internally by `indets(eqs,PolyExpr)`.

Option <*`Symbolic`*>:

- This option prevents conversion of the input data to floats.
- This option *must* be used, if the coefficients of the equations contain symbolic parameters that cannot be converted to floating point numbers.
- This option should not be used for equations with floating point coefficients! Numerical instabilities may occur in floating point operations. Cf. example 7.

Option <*`ShowAssumptions`*>:

- This option is only useful, if the equations contain symbolic parameters. Consequently, it should only be used in conjunction with the option *`Symbolic`*.
- The format of the return value is changed to `[Solution, Constraints, Pivots]`.

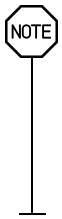
- `Solution` is a set of simplified equations representing the general solution subject to `Constraints` and `Pivots`.

- `Constraints` is a list of equations for symbolic parameters in `eqs`, which are necessary and sufficient to make the system solvable.

  Such constraints arise, if Gaussian elimination of the original equations leads to equations of the form $0 = c$, where c is some expression involving symbolic parameters in the "right hand side" of the system. All such equations are collected in `Constraints`. `numeric::linsolve` assumes that these equations are satisfied and returns a solution.

  If no such constraints arise, then the return value of `Constraints` is the empty list.

- This option changes the return strategy for "unsolvable" systems. Without the option *`Symbolic`* `FAIL` is returned, whenever Gaussian elimination produces an equation $0 = c$ with non-zero c. With *`ShowAssumptions`* such equations are returned via `Constraints`, provided c involves symbolic parameters.
  If c is a purely numerical value, then `[FAIL,[],[]]` is returned.



- `Pivots` is a list of inequalities involving symbolic parameters in the coefficient matrix A of the linear system $A\,x = b$ represented by `eqs`. Internally, division by pivot elements occurs in the Gaussian elimination. The expressions collected in `Pivots` are the numerators of those pivot elements that contain symbolic parameters. If only numerical pivot elements were used, then the return value of `Pivots` is the empty list.

- Cf. examples 2, 3, 4, 10.

Example 1. Equations and variables may be entered as sets or lists:

```
>> numeric::linsolve({x = y - 1, x + y = z}, {x, y}),
   numeric::linsolve([x = y - 1, x + y = z], {x, y}),
   numeric::linsolve({x = y - 1, x + y = z}, [x, y]),
   numeric::linsolve([x = y - 1, x + y = z], [x, y])

 [y = 0.5 z + 0.5, x = 0.5 z - 0.5],

    [y = 0.5 z + 0.5, x = 0.5 z - 0.5],

    [x = 0.5 z - 0.5, y = 0.5 z + 0.5],

    [x = 0.5 z - 0.5, y = 0.5 z + 0.5]
```

With the option *`Symbolic`* exact arithmetic is used. The following system has a 1-parameter set of solution, the unknown `x[3]` is arbitrary:

```
>> numeric::linsolve([x[1] + x[2] = 2, x[1] - x[2] = 2*x[3]],
                     [x[1], x[2], x[3]], Symbolic)
```

```
[x[1] = x[3] + 1, x[2] = 1 - x[3]]
```

The unknowns may be expressions:

```
>> numeric::linsolve([f(0) - sin(x + 1) = 2, f(0) = 1 - sin(x + 1)],
                     [f(0), sin(x + 1)])

                  [f(0) = 1.5, sin(x + 1) = -0.5]
```

The following system does not have a solution:

```
>> numeric::linsolve([x + y = 1, x + y = 2], [x, y])

                               FAIL
```

Example 2. We demonstrate some examples with symbolic coefficients. Note that the option *Symbolic* has to be used:

```
>> eqs := [x + a*y = b, x + A*y = b]:
>> numeric::linsolve(eqs, [x, y], Symbolic)

                           [x = b, y = 0]
```

Note that for $a = A$ this is not the general solution. Using the option *ShowAssumptions* it turns out, that the above result is the general solution subject to the assumption $a \neq A$:

```
>> numeric::linsolve(eqs, [x, y], Symbolic, ShowAssumptions)

               [[x = b, y = 0], [], [A - a <> 0]]

>> delete eqs:
```

Example 3. We give a further demonstration of the option *ShowAssumptions*. The following system does not have a solution for all values of the parameter `a`:

```
>> numeric::linsolve([x + y = 1, x + y = a], [x, y], Symbolic)

                               FAIL
```

With *ShowAssumptions* `numeric::linsolve` investigates, under which conditions (on `a`) there is a solution:

```
>> numeric::linsolve([x + y = 1, x + y = a], [x, y], Symbolic,
                     ShowAssumptions)
```

```
[[x = 1 - y], [a - 1 = 0], []]
```

We conclude that there is a 1-parameter set of solutions for $a = 1$. The constraint in `a` is a linear equation, since the parameter `a` enters the equations linearly. If `a` is regarded as an unknown rather than as a parameter, then the constraint becomes part of the solution:

```
>> numeric::linsolve([x + y = 1, x + y = a], [x, y, a], Sym-
bolic,
                    ShowAssumptions)

                [[x = 1 - y, a = 1], [], []]
```

Example 4. With exact arithmetic `PI` is regarded as a symbolic parameter. The following system has a solution subject to the constraint `PI=1`:

```
>> numeric::linsolve([x = x - y + 1, y = PI], [x, y],
                    Symbolic, ShowAssumptions)

                [[y = PI], [1 - PI = 0], []]
```

With floating point arithmetic `PI` is converted to `3.1415...`. The system has no solution:

```
>> numeric::linsolve([x = x - y + 1, y = PI], [x, y],
                    ShowAssumptions)

                        [FAIL, [], []]
```

Example 5. Since `numeric::linsolve` does not do a systematic internal check for non-linearities, the user should make sure that the equations to be solved are indeed linear in the unknowns. Otherwise strange things may happen. Garbage is produced for the following non-linear systems:

```
>> a := sin(x):
>> numeric::linsolve([y = 1 - a, x = y], [x, y], Symbolic)

               [x = 1 - sin(0), y = 1 - sin(0)]

>> numeric::linsolve([a*x + y = 1, x = y], [x, y], Symbolic)

          --           1                   1       --
          |  x = -------------, y = -------------  |
          |              3                   3      |
          --     sin(x16 ) + 1       sin(x16 ) + 1 --
```

Polynomial non-linearities are usually detected. Regarding `x,y,a` as unknowns the following quadratic system yields an error:

```
>> delete a: numeric::linsolve([x*a + y = 1, x = y], Symbolic)

 Error: this system does not seem to be linear
 [numeric::linsolve]
```

This system is linear in `x,y`, if `a` is regarded as a parameter:

```
>> numeric::linsolve([x*a + y = 1, x = y], [x, y], Symbolic)

                  --       1          1   --
                  |  x = -----, y = -----  |
                  --     a + 1      a + 1 --
```

Example 6. We solve a large sparse system. The coefficient matrix has only 3 diagonal bands. Note that both the equations as well as the variables are passed as lists. This guarantees that the band structure is not lost internally:

```
>> n := 100: x[0] := 0: x[n + 1] := 0:
   eqs := [x[i-1] - 2*x[i] + x[i+1] = 1 $ i = 1..n]:
   vars := [x[i] $ i = 1..n]:
   numeric::linsolve(eqs, vars)

 [x[1] = -50.0, x[2] = -99.0, x[3] = -147.0, x[4] = -194.0,

    ..., x[98] = -147.0, x[99] = -99.0, x[100] = -50.0]
```

The band structure is lost, if the equations or the unknowns are specified by sets. The following call takes more time than the previous call:

```
>> numeric::linsolve({op(eqs)}, {x[i] $ i = 1..n})

 [x[86] = -645.0, x[49] = -1274.0, x[12] = -534.0,

    ... , x[87] = -609.0, x[50] = -1275.0, x[13] = -572.0]

>> delete n, x:
```

Example 7. The option *`Symbolic`* should not be used for equations with floating point coefficients, because the symbolic pivoting strategy favors efficiency instead of numerical stability.

```
>> eqs := [x + 10^20*y = 10^20, x + y = 0]:
```

The float approximation of the exact solution is:

```
>> map(numeric::linsolve(eqs, [x, y], Symbolic), map, float)

                        [x = -1.0, y = 1.0]
```

We now convert the exact coefficients to floating point numbers:

```
>> feqs := map(eqs, map, float)

              [x + 1.0e20 y = 1.0e20, x + y = 0.0]
```

The default pivoting strategy stabilizes floating point operations. Consequently, one gets a correct result:

```
>> numeric::linsolve(feqs, [x, y])

                        [x = -1.0, y = 1.0]
```

With *Symbolic* the pivoting strategy optimizes speed, assuming exact arithmetic. Numerical instabilities may occur, if floating point coefficients are involved. The following incorrect result is caused by internal round-off effects ("cancellation"):

```
>> numeric::linsolve(feqs, [x, y], Symbolic)

                         [x = 0, y = 1.0]

>> delete eqs, feqs:
```

Example 8. We demonstrate that the simplified equations representing the solution can be used for further processing with `subs`:

```
>> eqs := [x + y = 1, x + y = a]:
>> [Solution, Constraints, Pivots] :=
   numeric::linsolve(eqs, [x, y], ShowAssumptions)

             [[x = 1.0 - 1.0 y], [a - 1.0 = 0], []]

>> subs(eqs, Solution)

                        [1.0 = 1, 1.0 = a]
```

The solution can be assigned to the unknowns via `assign`:

```
>> assign(Solution): x, y, eqs

               1.0 - 1.0 y, y, [1.0 = 1, 1.0 = a]

>> delete eqs, Solution, Constraints, Pivots, x, y:
```

Example 9. If the solution of the linear system is not unique, then some of the unknowns are used as "free parameters" spanning the solution space. In the following example the unknown `z` is such a parameter. It does not turn up on the left hand side of the solved equations:

```
>> eqs := [x + y = z, x + 2*y = 0, 2*x - z = -3*y, y + z = 0]:
>> vars := [w, x, y, z]:
>> Solution := numeric::linsolve(eqs, vars, Symbolic)
```

```
                          [x = 2 z, y = -z]
```

You may define a function such as the following `NewSolutionList` to rename your free parameters to "myName1", "myName2" etc. and fill up your list of solved equations accordingly:

```
>> NewSolutionList :=
   proc(Solution : DOM_LIST, vars : DOM_LIST, myName : DOM_STRING)
   local i, solvedVars, newEquation;
   begin
     solvedVars := map(Solution, op, 1);
     for i from 1 to nops(vars) do
        if not has(solvedVars, vars[i]) then
           newEquation := vars[i] = genident(myName);
           Solution := listlib::insertAt(
               subs(Solution, newEquation), newEquation, i)
        end_if
     end_for:
     Solution
   end_proc:
>> NewSolutionList(Solution, vars, "FreeParameter")
```

```
 [w = FreeParameter1, x = 2 FreeParameter2,

    y = -FreeParameter2, z = FreeParameter2]
```

```
>> delete eqs, vars, Solution, NewSolutionList:
```

Example 10. We demonstrate, how a complete solution of the following linear system in `x, y` with symbolic parameters may be found:

```
>> eqs := [x + y = A, a*x + b*y = 1, x + c*y = 1]:
>> numeric::linsolve(eqs, [x, y], Symbolic, ShowAssumptions)
```

```
 -- --     A b - 1      1 - A a --
 |  |  x = -------, y = -------  |,
 -- --      b - a        b - a  --

                                                        --
    [b - a - c - A b + A a c + 1 = 0], [b - a <> 0]      |
                                                        --
```

This is the general solution, assuming $a \neq b$. We now set $b = a$ to investigate further solution branches:

```
>> eqs := subs(eqs, b = a):
>> numeric::linsolve(eqs, [x, y], Symbolic, ShowAssumptions)

 -- --     A c - 1      1 - A --                                  -
-
 |  |  x = -------, y = -----  |, [1 - A a = 0], [c - 1 <> 0]  |
 -- --      c - 1       c - 1 --                                  -
-
```

This is the general solution for $a = b$, assuming $c \neq 1$. We finally set $c = 1$ to obtain the last solution branch:

```
>> eqs := subs(eqs, c = 1):
>> numeric::linsolve(eqs, [x, y], Symbolic, ShowAssumptions)

           [[x = A - y], [1 - A = 0, 1 - A a = 0], []]
```

From the constraints on the symbolic parameters `a` and `A` we conclude that there is a special 1-parameter solution $x = 1 - y$ for $a = b = c = A = 1$.

```
>> delete eqs:
```

Changes:

- `numeric::linsolve` is a new function.

`numeric::matlinsolve` – solve a linear matrix equation

`numeric::matlinsolve(A, B, ..)` returns the matrix solution X of the matrix equation $A\,X = B$.

Call(s):

- `numeric::matlinsolve(A, B <, Symbolic> <, ShowAssumptions>)`

Parameters:

`A` — an $m \times n$ matrix of domain type `DOM_ARRAY` or of category `Cat::Matrix`.

`B` — an $m \times p$ matrix of domain type `DOM_ARRAY` or of category `Cat::Matrix`. Column vectors `B` may also be represented by a 1-dimensional `array(1..m,[B1,B2,..])` or by a list `[B1,B2,..]`.

Options:

Symbolic	— prevents conversion of input data to floating point numbers
ShowAssumptions	— returns information on internal assumptions on symbolic parameters in A and B

Return Value: Without the option *ShowAssumptions* a list [X,KernelBasis] is returned. The solution X is an $n \times p$ array. KernelBasis is an $n \times d$ array. Its columns span the kernel of A. [FAIL,NIL] is returned, if the system is not solvable.

With *ShowAssumptions* a list [X,KernelBasis,Constraints,Pivots] is returned. The lists Constraints and Pivots contain equations and inequalities involving symbolic parameters in A and B. Internally these were assumed to hold true when solving the system. [FAIL,NIL,[],[]] is returned, if the system is not solvable.

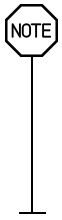
Side Effects: Without the option *Symbolic* the function is sensitive to the environment variable DIGITS, which determines the numerical working precision.

Related Functions: linalg::matlinsolve, linsolve, numeric::inverse, numeric::linsolve, solve

---

Details:

- numeric::matlinsolve is a fast numerical linear solver. It is also a recommended solver for linear systems with exact or symbolic coefficients (use option *Symbolic*).
- Without *Symbolic* the input data are converted to floating point numbers. The matrix A must not contain non-convertible parameters, unless *Symbolic* is used! If such objects are found, then numeric::matlinsolve automatically switches to its symbolic mode, issuing a warning. Symbolic parameters in B are accepted without warning.



- The numerical working precision is set by the environment variable DIGITS.
- X is a *special* solution of $A\,X = B$.
- The $n \times d$ array KernelBasis is the most general solution of $A\,X = 0$. Its columns span the d-dimensional kernel of A.
- If the kernel is empty, then the return value of KernelBasis is the null vector represented by array(1..n,1..1,[0,..,0]).
- If KernelBasis=array(1..n,1..1,[0,..,0]), then the dimension d of the kernel of A is d:=0. Otherwise it is d:=op(KernelBasis,[0,3,2]]).

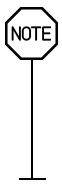
- Due to roundoff errors some or all basis vectors in the kernel of `A` may be missed in the numeric mode.

- The special solution `X` in conjunction with `KernelBasis` provides the general solution of $A\,X = B$. Solutions generated without the option *`ShowAssumptions`* are valid for arbitrary complex values of the symbolic parameters which may be present in `A` and `B`. If no such solution exists, then `[FAIL,NIL]` is returned. Solutions that are valid only for special values of the symbolic parameters may be obtained with *`ShowAssumptions`*. Cf. examples 3, 4, 7.

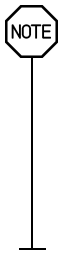
- `numeric::matlinsolve` internally uses a sparse representation of the matrices. It is suitable for solving large sparse systems. Cf. example 5. Note, however, that the input requires specification of entire matrices. For this reason you should consider using `numeric::linsolve` which allows to input the linear system as a sparse list of symbolic equations.

- `numeric::matlinsolve` does not react to any assumptions on symbolic parameters in `A,B` that are set via `assume`.

- Gaussian elimination with partial pivoting is used. Without option *`Symbolic`* the pivoting strategy takes care of numerical stabilization. With *`Symbolic`* exact data are assumed. The symbolic pivoting strategy tries do maximize speed and does not take care of numerical stabilization! Cf. example 6.



- `Cat::Matrix` objects `A` from matrix domains such as `Dom::Matrix(..)` or `Dom::SquareMatrix(..)` are internally converted to arrays over expressions via `A::dom::expr(A)`. Note that `linalg::matlinsolve` must be used, when the solution is to be computed over the component domain. Cf. example 8. Note that the option *`Symbolic`* should be used, if the entries cannot be converted to numerical expressions.



Option <*`Symbolic`*>:

- This option prevents conversion of the input data to floats.

- This option *must* be used, if the matrix A contains symbolic parameters that cannot be converted to floating point numbers.

- This option should not be used for matrices with floating point entries! Numerical instabilities may occur in floating point operations. Cf. example 6.

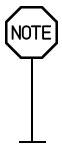
Option <`ShowAssumptions`>:

- This option is only useful, if the matrices contain symbolic parameters. Consequently, it should only be used in conjunction with the option `Symbolic`.
- This option changes the format of the return value to `[X, KernelBasis, Constraints, Pivots]`.

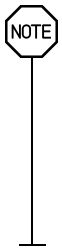
- `X` and `KernelBasis` represent the general solution subject to `Constraints` and `Pivots`.
- `Constraints` is a list of equations for symbolic parameters in B which are necessary and sufficient for $A\,X = B$ to be solvable.

  Such constraints arise, if Gaussian elimination leads to equations of the form $0 = c$, where c is some expression involving symbolic parameters contained in B. All such equations are collected in `Constraints`. `numeric::matlinsolve` assumes that these equations are satisfied and returns a special solution `X`.

  If no such constraints arise, then the return value of `Constraints` is the empty list.
- `Constraints` usually is a *non-linear* list of equation for the symbolic parameters. It is not investigated by `numeric::matlinsolve`, i.e., solutions may be returned, even if the `Constraints` cannot be satisfied. Cf. example 3.



- This option changes the return strategy for "unsolvable" systems. Without the option `ShowAssumptions` the result `[FAIL,NIL]` is returned, whenever Gaussian elimination produces an equation $0 = c$ with non-zero c. With `ShowAssumptions` such equations are returned via `Constraints`, provided c involves symbolic parameters. If c is a purely numerical value, then `[FAIL,NIL,[],[]]` is returned.



- `Pivots` is a list of inequalities involving symbolic parameters in A. Internally, division by pivot elements occurs in the Gaussian elimination. The expressions collected in `Pivots` are the numerators of those pivot elements that involve symbolic parameters contained in A. If only numerical pivot elements are used, then the return value of `Pivots` is the empty list.
- Cf. examples 3, 4, 7.

Example 1. We use equivalent input formats `B1`, ... ,`B4` to represent a vector with components $[a, \pi]$. First, this vector is defined as a 2-dimensional array:

```
>> A := array(1..2, 1..3, [[1, 2, 3],[1, 1, 2]]):
>> B1 := array(1..2, 1..1, [[a], [PI]]):
>> numeric::matlinsolve(A, B1)
```

$$\begin{pmatrix} 6.283185307 - 1.0\,a & -1.0 \\ 1.0\,a - 3.141592654 & -1.0 \\ 0 & 1 \end{pmatrix}$$

Next, we use a 1-dimensional array and compute an exact solution:

```
>> B2 := array(1..2, [a, PI]):
>> numeric::matlinsolve(A, B2, Symbolic)
```

$$\begin{pmatrix} 2\,PI - a & -1 \\ a - PI & -1 \\ 0 & 1 \end{pmatrix}$$

Now a list is used to specify the vector. No internal assumptions were used by `numeric::matlinsolve` to obtain the solution:

```
>> B3 := [a, PI]:
>> numeric::matlinsolve(A, B3, ShowAssumptions)
```

$$\begin{pmatrix} 6.283185307 - 1.0\,a & -1.0 \\ 1.0\,a - 3.141592654 & -1.0 \\ 0 & 1 \end{pmatrix}, [], []$$

Finally, we use `Dom::Matrix` objects to specify the system. Note that the result are still arrays:

```
>> A := Dom::Matrix()([[1, 2, 3],[1, 1, 2]]):
>> B4 := Dom::Matrix()([a, PI]):
```

```
>> numeric::matlinsolve(A, B4)
```

$$\begin{bmatrix} 6.283185307 - 1.0\,a \\ 1.0\,a - 3.141592654 \\ 0 \end{bmatrix}, \begin{bmatrix} -1.0 \\ -1.0 \\ 1 \end{bmatrix}$$

```
>> delete A, B1, B2, B3, B4:
```

Example 2. We invert a matrix by solving $A\,X = 1$:

```
>> A := array(1..3, 1..3, [[1, 1, 0], [0, 1, 1], [0, 0, 1]]):
>> B := array(1..3, 1..3, [[1, 0, 0], [0, 1, 0], [0, 0, 1]]):
>> InverseOfA := numeric::matlinsolve(A, B, Symbolic)[1]
```

$$\begin{bmatrix} 1, & -1, & 1 \\ 0, & 1, & -1 \\ 0, & 0, & 1 \end{bmatrix}$$

```
>> delete A, B, InverseOfA:
```

Example 3. We solve an equation with a symbolic parameter x:

```
>> M := Dom::Matrix():
>> A := M(3, 3, [[2, 2, 3], [1, 1, 2], [3, 3, 5]]):
>> B := M(3, 1, [sin(x)^2, cos(x)^2, 0]):
>> [X, Kernel, Constraints, Pivots] :=
   numeric::matlinsolve(A, B, Symbolic, ShowAssumptions)
```

$$\begin{bmatrix} 5\sin(x)^2 \\ 0 \\ -3\sin(x)^2 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}, [\cos(x)^2 + \sin(x)^2 = 0], []$$

This solution holds subject to the constraint $\sin(x)^2 + \cos(x)^2 = 0$ on the parameter x. `numeric::matlinsolve` does not investigate the `Constraints` and does not realize that they cannot be satisfied. We check the consistency of the "result" by inserting the solution into the original system. For convenience we convert the arrays `X` and `Kernel` to `Dom::Matrix` objects and use the overloaded operators `*` and `-` for matrix multiplication and subtraction:

```
>> A*M(X) - B, A*M(Kernel)
```

$$\begin{pmatrix} 0 & \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \\ -\cos(x)^2 - \sin(x)^2 & \end{pmatrix}$$

```
>> delete M, A, B, X, Kernel, Constraints, Pivots:
```

Example 4. We give a further demonstration of the option *`ShowAssumptions`*. The following system does not have a solution for all values of the parameter `a`:

```
>> A := array(1..2, 1..2, [[1, 1], [1, 1]]):
>> B := array(1..2, 1..1, [[1], [a]]):
>> numeric::matlinsolve(A, B, Symbolic)

                          [FAIL, NIL]
```

With *`ShowAssumptions`* `numeric::matlinsolve` investigates, under which conditions (on `a`) there is a solution:

```
>> numeric::matlinsolve(A, B, Symbolic, ShowAssumptions)
```

$$\left[\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 1 \end{pmatrix}, [a - 1 = 0], [] \right]$$

We conclude that there is a 1-dimensional solution space for $a = 1$.

```
>> delete A, B:
```

Example 5. We solve a large sparse system with 3 diagonal bands:

```
>> n := 100: A := Dom::Matrix()(n, n, [1, -2, 1], Banded):
>> B := array(1..n, [1 $ n]): numeric::matlinsolve(A, B)
```

```
                  -- +-          -+  +-    -+ --
                  |  |   -50.0    |  |  0  |  |
                  |  |            |  |     |  |
                  |  |   -99.0    |  |  0  |  |
                  |  |            |  |     |  |
                  |  |    ...     |  |  ...|  |
                  |  |            |  |     |  |
                  |  |   -50.0    |  |  0  |  |
                  -- +-          -+  +-    -+ --
```

```
>> delete n, A, B:
```

Example 6. The option *Symbolic* should not be used for equations with floating point coefficients, because the symbolic pivoting strategy favors efficiency instead of numerical stability.

```
>> A := array(1..2, 1..2, [[1, 10^20], [1, 1]]):
>> B := array(1..2, 1..1, [[10^20], [0]]):
```

The float approximation of the exact solution is:

```
>> map(numeric::matlinsolve(A, B, Symbolic)[1], float)
```

```
                          +-      -+
                          |  -1.0  |
                          |        |
                          |   1.0  |
                          +-      -+
```

We now convert the exact input data to floating point approximations:

```
>> A := map(A, float): B := map(B, float):
```

The default pivoting strategy stabilizes floating point operations. Consequently, one gets a correct result:

```
>> numeric::matlinsolve(A, B)[1]
```

```
                          +-      -+
                          |  -1.0  |
                          |        |
                          |   1.0  |
                          +-      -+
```

With the option *Symbolic* the pivoting strategy optimizes speed, assuming exact arithmetic. Numerical instabilities may occur, if floating point coefficients are involved. The following result is caused by internal round-off effects ("cancellation"):

```
>> numeric::matlinsolve(A, B, Symbolic)[1]
```

$$\begin{pmatrix} 0 \\ 1.0 \end{pmatrix}$$

```
>> delete A, B:
```

Example 7. We demonstrate, how a complete solution of the following linear system with symbolic parameters may be found:

```
>> A := array(1..3, 1..2, [[1, 1], [a, b], [1, c]]):
>> B := array(1..3, 1..1, [[1], [1], [1]]):
>> numeric::matlinsolve(A, B, Symbolic, ShowAssumptions)
```

$$\left[\begin{pmatrix} \frac{b-1}{b-a} \\ \frac{1-a}{b-a} \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix}, [a\,c - c - a + 1 = 0], [b - a \neq 0] \right]$$

This is the general solution, assuming $a \neq b$. We now set $b = a$ to investigate further solution branches:

```
>> A := subs(A, b = a):
>> numeric::matlinsolve(A, B, Symbolic, ShowAssumptions)
```

$$\left[\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix}, [1 - a = 0], [c - 1 \neq 0] \right]$$

This is the general solution for $a = b$, assuming $c \neq 1$. We finally set $c = 1$ to obtain the last solution branch:

```
>> A := subs(A, c = 1):
>> numeric::matlinsolve(A, B, Symbolic, ShowAssumptions)
```

```
            -- +-   -+  +-    -+                   --
            |  |  1  |  |  -1  |                    |
            |  |     |, |      |, [1 - a = 0], []  |
            |  |  0  |  |   1  |                    |
            -- +-   -+  +-    -+                   --
```

From the constraint on `a` we conclude that there is a 1-dimensional solution space for the special values $a = b = c = 1$ of the symbolic parameters.

```
>> delete A, B:
```

Example 8. Matrices from a domain such as `Dom::Matrix(..)` are converted to arrays with numbers or expressions. Hence `numeric::matlinsolve` finds no solution for the following system:

```
>> M := Dom::Matrix(Dom::IntegerMod(7)):
>> A := M([[1, 4], [6, 3], [3, 2]]): B := M([[9], [5], [0]]):
>> numeric::matlinsolve(A, B)
```

```
                          [FAIL, NIL]
```

Use `linalg::matlinsolve` to solve the system over the coefficient field of the matrices. A solution does exist over the field `Dom::IntegerMod(7)`:

```
>> linalg::matlinsolve(A, B)
```

```
                          +-         -+
                          |  1 mod 7  |
                          |           |
                          |  2 mod 7  |
                          +-         -+
```

```
>> delete M, A, B:
```

Changes:

- `numeric::matlinsolve` is a new function.

`numeric::ncdata` – weights and abscissae of Newton-Cotes quadrature

`numeric::ncdata(n)` returns the weights and the abscissae of the Newton-Cotes quadrature rule with n equidistant nodes.

Call(s):

- `numeric::ncdata(n)`

Parameters:

`n` — the number of nodes: a positive integer

Return Value: A list `[b,c]` is returned. The lists `b=[b[1],..,b[n]]` and `c=[c[1],..,c[n]]` are the rational weights and abscissae of the Newton-Cotes quadrature rule, respectively.

Side Effects: The function uses `option remember`. It is not sensitive to the environment variable `DIGITS`.

Related Functions: `numeric::gldata`, `numeric::gtdata`, `numeric::quadrature`

Details:

- The Newton-Cotes quadrature rule $\sum_{i=1}^{n} b_i f(c_i)$ produces the exact integral $\int_0^1 f(x)\,dx$ for all polynomials f through degree $n-1$. If n is odd, then the quadrature rule is exact through degree n.
- The equidistant abscissae $c = [c_1, \ldots, c_n]$ are given by $c_i = (i-1)/(n-1)$.

Example 1. The following call produces exact data for the quadrature rule with four nodes:

```
>> numeric::ncdata(4)
```

```
            [[1/8, 3/8, 3/8, 1/8], [0, 1/3, 2/3, 1]]
```

Background:

- The numerical integrator `numeric::quadrature` calls `numeric::ncdata` to provide the data for Newton-Cotes quadrature.

Changes:

- No changes.

`numeric::odesolve` – numerical solution of an ordinary differential equation

`numeric::odesolve(t0..t, f, Y0, ..)` returns a numerical approximation of the solution $Y(t)$ of the first order differential equation (dynamical system)

$$\frac{dY}{dt} = f(t, Y) , \quad Y(t_0) = Y_0 , \quad t_0, t \in R , \quad Y_0, Y(t) \in C^n .$$

Call(s):

```
numeric::odesolve(t0..t, f, Y0 <, method> <, RelativeError = tol> <, Stepsize = h>
                  <, Alldata = n> <, Symbolic> )
```

Parameters:

- `t0` — numerical real value for the initial time t_0
- `t` — numerical real value (the "time")
- `f` — procedure representing the vector field
- `Y0` — list or 1-dimensional array of numerical values representing the initial condition Y_0

Options:

- `method` — name of the numerical scheme
- `RelativeError = tol` — forces internal numerical Runge-Kutta steps to use stepsizes with local discretization errors below `tol`. This tolerance must be a numerical real value $\geq 10^{-\text{DIGITS}}$.
- `Stepsize = h` — switches off the internal error control and uses a Runge-Kutta iteration with constant stepsize `h`. `h` must be a numerical real value.
- `Alldata = n` — makes `numeric::odesolve` return a list of numerical mesh points generated by the internal Runge-Kutta iteration. The integer `n` controls the size of the output list.
- `Symbolic` — makes `numeric::odesolve` return a vector of symbolic expressions representing a single symbolic step of the Runge-Kutta iteration.

Return Value: The solution vector $Y(t)$ is returned as a list or as a 1-dimensional array of floating point values. The type of the result vector coincides with the type of the input vector `Y0`. With the option `Alldata` a list of mesh data is returned.

Side Effects: The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Related Functions: `numeric::butcher`, `numeric::odesolve2`, `plot::ode`

Details:

- `numeric::odesolve` is a general purpose solver able to deal with initial value problems of various kinds of (non-stiff) ordinary differential equations. Equations $y^{(p)} = f(t, y, y', \ldots, y^{(p-1)})$ of order p can be solved by `numeric::odesolve` after reformulation to dynamical system form. This can alway be achieved by writing the equation as a first order system

$$\frac{d}{dt}\begin{pmatrix} Y_1 \\ \vdots \\ Y_{p-1} \\ Y_p \end{pmatrix} = \begin{pmatrix} Y_2 \\ \vdots \\ Y_p \\ f(t, Y_1, \ldots, Y_p) \end{pmatrix}$$

  for the vector $[Y_1, \ldots, Y_p] = [y, y', \ldots, y^{(p-1)}]$. Cf. example 4.

- The input data `t0`, `t` and `Y0` must not contain symbolic objects which cannot be converted to floating point values via `float`. Numerical expressions such as `exp(PI)`, `sqrt(2)` etc. are accepted.

- The vector field `f` defining the dynamical system $Y' = f(t, Y)$ must be represented by a procedure with two input parameters: the scalar time `t` and the vector `Y`. `numeric::odesolve` internally calls this function with real floating point values `t` and a list `Y` of floating point values. It has to return the vector $f(t, Y)$ either as a list or as a 1-dimensional array. The output of `f` may contain numerical expressions such as `PI`, `exp(2)` etc. However, all values must be convertible to real or complex floating point numbers by `float`.

- Also autonomous systems, where $f(t, Y)$ does not depend on t, must be represented by a procedure with 2 arguments `t` and `Y`.

- Also for scalar equations `Y` has to be represented by a list or an array with one element. For instance, the input data for the scalar initial value problem $y' = t\,\sin(y),\ y(0) = 1$ may be of the form

```
f := proc(t,Y)  /* Y is a 1-dimensional vector */
local y;        /* represented by a list with */
begin           /* one element:  Y = [y]. */
  y := Y[1];
  [t*sin(y)]    /* the output is a list with 1 element */
end_proc:
Y0 := [1]:      /* the initial value */
```

- The numerical precision is controlled by the global variable `DIGITS`: an adaptive control of the stepsize keeps local relative discretization errors below `tol`$= 10^{-\texttt{DIGITS}}$, unless a different tolerance is specified via the option *`RelativeError`*=`tol`. The error control may be switched off by specifying a fixed *`Stepsize`*= `h`.

- Only local errors are controlled by the adaptive mechanism. No control of the global error is provided!

- With `Y := t -> numeric::odesolve(t0..t, f, Y0 <,options>)` the numerical solution can be repesented by a MuPAD function: the call `Y(t)` will start the numerical integration from `t0` to `t`. A more sophisticated form of this function may be generated via

  `Y := numeric::odesolve2(f, t0, Y0 <, options>)`

  This equips `Y` with a remember mechanism that uses previously computed values to speed up the computation. Cf. example 2.

- For systems of the special form $Y' = f(t, Y)Y$ with a matrix valued function $f(t, Y)$ there is a special solver `numeric::odesolveGeometric` which preserves geometric features of the system more faithfully than `numeric::odesolve`.

Option <`method`>:

- Presently the following single step Runge-Kutta type methods are implemented:

`EULER1` (order 1),	*`RKF43`* (order 3),	*`xRKF43`* (order 3),
`RKF34` (order 4),	*`xRKF34`* (order 4),	*`RK4`* (order 4),
`RKF54a` (order 4),	*`RKF54b`* (order 4),	*`DOPRI54`* (order 4),
`xDOPRI54`(order 4),	*`CK54`* (order 4),	*`xRKF54a`* (order 4),
`xRKF54b` (order 4),	*`xCK54`* (order 4),	*`RKF45a`* (order 5),
`RKF45b` (order 5),	*`DOPRI45`* (order 5),	*`CK45`* (order 5),
`xRKF45a` (order 5),	*`xRKF45b`* (order 5),	*`xDOPRI45`*(order 5),
`xCK45` (order 5),	*`BUTCHER6`* (order 6),	*`RKF87`* (order 7),
`xRKF87` (order 7),	*`RKF78`* (order 8),	*`xRKF78`* (order 8).

  The Background below provides some information on these methods.

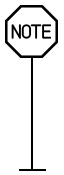
- There is usually no need to change the default method *`RKF78`* by this option. This method is an embedded Runge-Kutta-Fehlberg pair of orders 7 and 8.

- Cf. example 6.

Option <`RelativeError = tol`>:

- The internal control mechanism estimates the local relative discretization error of a Runge-Kutta step and adjusts the stepsize adaptively to keep this error below `tol`.
- The default setting of `tol` $= 10^{-\texttt{DIGITS}}$ ensures that the local discretization errors are of the same order of magnitude as numerical roundoff.

  Usually there is no need to use this option to change this setting. However, occasionally the numerical evaluation of the Runge-Kutta steps may be ill-conditioned or stepsizes are so small that the time parameter cannot be incremented by the stepsize within working precision. In such a case this option may be used to bound the local discretization error by `tol` and use a higher working precision given by `DIGITS`.
- Only real numerical values `tol` $\geq 10^{-\texttt{DIGITS}}$ are accepted.
- The global error of the result returned by `numeric::odesolve` is usually larger than the local errors bounded by `tol`. Although the result is displayed with `DIGITS` decimal places one should not expect that all of them are correct. The relative precision of the final result is `tol` at best!



- Cf. example 7.

Option <`Stepsize = h`>:

- By default, `numeric::odesolve` uses an adaptive stepsize control mechanism in the numerical iteration. The option *`Stepsize`*=`h` switches off this adaptive mechanism and uses the Runge-Kutta method specified by `method` (or the default method *`RKF78`*) with constant stepsize `h`.
- A final step with smaller stepsize is used to match the end `t` of the integration interval `t0..t`, if `(t-t0)/h` is not integer.
- With this option there is no automatic error control! Depending on the problem and on the order of the method the result may be a poor numerical approximation of the exact solution.

- There is usually no need to invoke this option. However, occasionally the built-in adaptive error control mechanism may fail when integrating close to a singularity. In such a case this option may be used to customize a control mechanism for the global error by using different stepsizes and investigating the convergence of the corresponding results.
- Cf. example 8.

Option <`Alldata = n`>:

- With this option `numeric::odesolve` returns a list of numerical mesh points

  ```
  [[t0, Y0], [t1, Y1], .., [t, Y(t)]]
  ```

  generated by the internal Runge-Kutta iteration.

- The integer `n` controls the size of the output list. For $n = 1$ all internal mesh points are returned. This case may also be invoked by entering the simplified option `Alldata`, which is equivalent to `Alldata=1`. For $n > 1$ only each n-th mesh point is stored in the list. The list always contains the initial point `[t0,Y0]` and the final point `[t,Y(t)]`. For $n \leq 0$ only the data `[[t0,Y0],[t,Y(t)]]` are returned.

- The output list may be useful to inspect the internal numerical process. Also further graphical processing of the mesh data may be useful.

- Cf. example 9.

Option <`Symbolic`>:

- The call `numeric::odesolve(t0..t, f, Y0, <method>, Symbolic)` returns a vector (list or array) of expressions representing a single step of the numerical scheme with stepsize `t-t0`. In this mode symbolic values for `t0`, `t` and the components of `Y0` are accepted.

- This option may be useful, if the specified numerical method applied to a given differential equation is to be investigated symbolically.

- Cf. example 10.

Example 1. We compute the numerical solution $y(10)$ of the initial value problem $y' = t \sin(y)$, $y(0) = 2$:

```
>> f := proc(t, Y) begin [t*sin(Y[1])] end_proc:
>> numeric::odesolve(0..10, f, [2])

                          [3.141592654]

>> delete f:
```

Example 2. We consider $y' = y,\ y(0) = 1$. The numerical solution may be represented by the function

```
>> Y := t -> numeric::odesolve(0..t, (t,Y) -> Y, [1]):
```

Calling `Y(t)` starts the numerical integration:

```
>> Y(-5), Y(0), Y(1), Y(PI)

      [0.006737946999], [1.0], [2.718281828], [23.14069263]

>> delete Y:
```

Example 3. We compute the numerical solution $Y(\pi) = [x(\pi), y(\pi)]$ of the system

$$x' = x + y\ , \quad y' = x - y\ , \quad x(0) = 1\ , \quad y(0) = \sqrt{-1}\ .$$

```
>> f := (t, Y) -> [Y[1] + Y[2], Y[1] - Y[2]]: Y0 := [1, I]:
>> numeric::odesolve(0..PI, f, Y0)

   [72.57057162 + 30.05484302 I, 30.05484302 + 12.46088558 I]
```

The solution of a linear dynamical system $Y' = A\,Y$ with a constant matrix A is $Y(t) = \exp(t\,A)\,Y_0$. The solution of the system above can also be computed by:

```
>> t := PI: tA := array(1..2, 1..2, [[t, t], [t, -t]]):
>> numeric::expMatrix(tA, Y0)

   [72.57057163 + 30.05484303 I, 30.05484303 + 12.46088558 I]

>> delete f, Y0, t, tA:
```

Example 4. We compute the numerical solution $y(1)$ of the differential equation $y'' = y^2$ with initial conditions $y(0) = 0$, $y'(0) = 1$. The second order equation is converted to a first order system for $Y = [y, y'] = [y, z]$:

$$y' = z\ , \quad z' = y^2\ , \quad y(0) = 0\ , \quad z(0) = 1\ .$$

```
>> f := proc(t, Y) begin [Y[2], Y[1]^2] end_proc: Y0 := [0, 1]:
>> numeric::odesolve(0..1, f, Y0)

                    [1.087473533, 1.362851121]

>> delete f, Y0:
```

Example 5. We demonstrate how numerical data can be obtained on a user defined time mesh `t[i]`. The initial value problem is $y' = \sin(t) - y$, $y(0) = 1$, the sample points are `t[0]=0.0`, `t[1]=0.1`, ..., `t[100]=10.0`. First, we define the differential equation and the initial condition:

```
>> f := (t, Y) -> [sin(t) - Y[1]]: Y[0] := [1]:
```

We define the time mesh:

```
>> for i from 0 to 100 do t[i] := i/10 end_for:
```

The equation is integrated iteratively from `t[i-1]` to `t[i]` with a working precision of 4 significant decimal places:

```
>> DIGITS := 4:
>> for i from 1 to 100 do
     Y[i] := numeric::odesolve(t[i-1]..t[i], f, Y[i-1])
   end_for:
```

The following mesh data are produced:

```
>> [t[i], Y[i]] $ i = 0..100

 [[0, [1]], [1/10, [0.9097]], [1/5, [0.8374]], [3/10, [0.7813]],

    [2/5, [0.7397]], ... , [99/10, [0.2159]], [10, [0.1476]]]
```

These data can be displayed by a list plot:

```
>> plotpoints := [point(t[i], op(Y[i])) $ i = 0..100]:
>> plot2d([Mode = List, plotpoints])
```

The same plot can be obtained directly via `plot::ode`:

```
>> plot(plot::ode(
       [t[i] $ i = 0..100], f, Y[0],
       [(t, Y) -> [t, Y[1]], Style = Points]))
>> delete f, t, DIGITS, Y, plotpoints:
```

Example 6. We compute the numerical solution $y(1)$ of $y' = y$, $y(0) = 1$ by the classical 4-th order Runge-Kutta method *RK4*. By internal local extrapolation, its effective order is 5:

```
>> f := (t, Y) -> Y: DIGITS := 13:
>> numeric::odesolve(0..1, f, [1], RK4)

                        [2.718281828459]
```

Next we use local extrapolation *xRKF78* of the 8-th order submethod of the Runge-Kutta-Fehlberg pair *RKF78*. This scheme has effective order 9:

```
>> numeric::odesolve(0..1, f, [1], xRKF78)

                          [2.718281828459]
```

Both methods yield the same result because of the internal adaptive error control. However, due to its higher order, the method *xRKF78* is faster.

```
>> delete f, DIGITS:
```

Example 7. We consider the initial value problem $y' = -10^{10}\,y\,(1-\cos(y))$, $y(0) = 1$. We note that the numerical evaluation of the right hand side of the equation suffers from cancellation effects, when $|y|$ is small.

```
>> f := (t, Y) -> [-10^10*Y[1]*(1 - cos(Y[1]))]: Y0 := [1]:
```

We first attempt to compute $y(1)$ with a working precision of 4 digits using the default setting `RelativeError` $= 10^{\texttt{DIGITS}} = 10^{-4}$:

```
>> DIGITS := 4: numeric::odesolve(0..1, f, Y0)

                              [2.931e-5]
```

Due to numerical roundoff in the internal steps no digit of this result is correct. Next we use a working precision of 20 significant decimal places to eliminate roundoff effects:

```
>> DIGITS := 20:
>> numeric::odesolve(0..1, f, Y0, RelativeError = 10^(-4))

                  [0.00000999999975776021931 32]
```

Since relative local discretization errors are of the magnitude 10^{-4}, not all displayed digits are trustworthy. We finally use a working precision of 20 digits and constrain the local relative discretization errors by the tolerance 10^{-10}:

```
>> numeric::odesolve(0..1, f, Y0, RelativeError = 10^(-10))

                  [0.00001000000000004937459 06]

>> delete f, Y0, DIGITS:
```

Example 8. We compute the numerical solution $y(1)$ of $y' = y$, $y(0) = 1$ with various methods and various constant stepsizes. We compare the result with the exact solution $y(1) = \exp(1) = 2.718281828\ldots$.

```
>> f := (t, Y) -> Y: Y0 := [1]:
```

We first use the Euler method of order 1 with two different stepsizes:

```
>> Y := numeric::odesolve(0..1, f, Y0, EULER1, Stepsize = 0.1):
>> Y, globalerror = float(exp(1)) - Y[1]

            [2.59374246], globalerror = 0.1245393684
```

Decreasing the stepsize by a factor of 10 should reduce the global error by a factor of 10. Indeed:

```
>> Y := numeric::odesolve(0..1, f, Y0, EULER1, Stepsize = 0.01):
>> Y, globalerror = float(exp(1)) - Y[1]

            [2.70481383], globalerror = 0.01346799904
```

Next we use the classical Runge-Kutta method of order 4 with two different stepsizes:

```
>> Y := numeric::odesolve(0..1, f, Y0, RK4, Stepsize = 0.1):
>> Y, globalerror = float(exp(1)) - Y[1]

         [2.718279744], globalerror = 0.000002084323879
```

Decreasing the stepsize by a factor of 10 in a 4-th order scheme should reduce the global error by a factor of 10^4. Indeed:

```
>> Y := numeric::odesolve(0..1, f, Y0, RK4, Stepsize = 0.01):
>> Y, globalerror = float(exp(1)) - Y[1]

       [2.718281828], globalerror = 0.0000000002246438649

>> delete f, Y0, Y:
```

Example 9. We integrate $y' = y^2$, $y(0) = 1$ over the interval $t \in [0, 0.99]$ with a working precision of 4 digits. The exact solution is $y(t) = 1/(1-t)$. Note the singularity at $t = 1$.

```
>> DIGITS := 4: f := (t, Y) -> [Y[1]^2]: Y0 := [1]:
```

The option `Alldata`, equivalent to `Alldata`=1, yields all mesh data generated during the internal adaptive process:

```
>> numeric::odesolve(0..0.99, f, Y0, Alldata)
```

```
 [[0.0, [1.0]], [0.5668, [2.308]], [0.7784, [4.513]],

    [0.8842, [8.636]], [0.9371, [15.9]], [0.9636, [27.43]],

    [0.9768, [43.05]], [0.99, [99.99]]]
```

With *Alldata*=2, only each second point is returned:

```
>> numeric::odesolve(0..0.99, f, Y0, Alldata = 2)

 [[0.0, [1.0]], [0.7784, [4.513]], [0.9371, [15.9]],

    [0.9768, [43.05]], [0.99, [99.99]]]
```

One can control the time mesh using the option *Stepsize*=h:

```
>> numeric::odesolve(0..0.99, f, Y0, Stepsize=0.1, Alldata = 1)

 [[0.0, [1.0]], [0.1, [1.111]], [0.2, [1.25]], [0.3, [1.429]],

    [0.4, [1.667]], [0.5, [2.0]], [0.6, [2.5]], [0.7, [3.333]],

    [0.8, [5.0]], [0.9, [10.0]], [0.99, [131.2]]]
```

However, with the option *Stepsize*=h, no automatic error control is provided by `numeric::odesolve`. Note the poor approximation $y(t) = 131.1$ for $t = 0.99$ (the exact value is $y(0.99) = 100.0$). The next computation with smaller stepsize yields better results:

```
>> numeric::odesolve(0..0.99, f, Y0, Stepsize = 0.01, All-
data = 10)

 [[0.0, [1.0]], [0.1, [1.111]], [0.2, [1.25]], [0.3, [1.429]],

    [0.4, [1.667]], [0.5, [2.0]], [0.6, [2.5]], [0.7, [3.333]],

    [0.8, [5.0]], [0.9, [10.0]], [0.99, [100.0]]]
```

Example 5 demonstrates how accurate numerical data on a user defined time mesh can be generated using the automatic error control of `numeric::odesolve`.

```
>> delete DIGITS, f, Y0:
```

Example 10. The second order equation $y'' + \sin(y) = 0$ is written as the dynamical system $y' = z$, $z' = -\sin(y)$ for the vector $Y = [y, z]$. A single symbolic step

$$[y(t_0), z(t_0)] \mapsto [y(t_0 + h), z(t_0 + h)]$$

of the Euler method is computed:

```
>> f := proc(t, Y) begin [Y[2], -sin(Y[1])] end_proc:
>> numeric::odesolve(t0..t0+h, f, [y0, z0], EULER1, Symbolic)

                    [y0 + h z0, z0 - h sin(y0)]

>> delete f:
```

Background:

- All methods presently implemented are adaptive versions of Runge-Kutta type single step schemes. References:

  J.C. Butcher: "The Numerical Analysis of Ordinary Differential Equations", Wiley, Chichester (1987).

  E. Hairer, S.P. Nørsett and G. Wanner: "Solving Ordinary Differential Equations I", Springer, Berlin (1993).

- The methods *RKF43*, *RKF34*, *RKF54a*, *RKF54b*, *RKF45a*, *RKF45b*, *RKF87*, *RKF78*, *DOPRI54*, *DOPRI45*, *CK54*, *CK45* are embedded pairs of Runge-Kutta-Fehlberg, Dormand-Prince and Cash-Karp type, respectively. Estimates of the local discretization error are obtained in the usual way by comparing the results of the two submethods of the pair. The names indicate the orders of the subprocesses. For instance, *RKF34* and *RKF43* denote the same embedded Runge-Kutta-Fehlberg pair with orders 3 and 4. In *RKF34* the result of the fourth order submethod is accepted, whereas *RKF43* advances using the result of the third order submethod. In both cases the discretization error of the lower order subprocess is estimated and controlled.

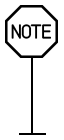
- For the single methods *EULER1* (the first order Euler scheme), *RK4* (the classical fourth order Runge-Kutta scheme) and *BUTCHER6* (a Runge-Kutta scheme of order 6) the relative local error is controlled by comparing steps with different stepsizes. The effective order of these methods is increased by one through local extrapolation.

- Local extrapolation is also available for the submethods of the embedded pairs. For instance, the method *xRKF78* uses extrapolation of the 8-th order subprocess of *RKF78*, yielding a method of effective order 9. The 7-th order subprocess is ignored. The cheap error estimate based on the embedded pair is not used implying some loss of efficiency when comparing *RKF78* (order 8) and *RKF78* (effective order 9).

- The call `numeric::butcher(method)` returns the Butcher data of the methods used in `numeric::odesolve`. Here `method` is one of *EULER1*, *RKF43*, *RK4*, *RKF34*, *RKF54a*, *RKF54b*, *DOPRI54*, *CK54*, *RKF45a*, *RKF45b*, *DOPRI45*, *CK45*, *BUTCHER6*, *RKF87*, *RKF78*.

- Only local errors are controlled by the adaptive mechanism. No control of the global error is provided!

- The run time of the numerical integration with a method of order p grows like $O(10^{\texttt{DIGITS}/(p+1)})$, when `DIGITS` is increased. Computations with high precision goals are very expensive! High order methods such as the default method *`RKF78`* should be used.



- Presently only explicit single step methods of Runge-Kutta type are implemented. Stiff problems cannot be handled efficiently with these methods.

- For problems of the special type $Y' = f(t,Y)Y$ with a matrix valued function $f(t,Y)$ there is a specialized ("geometric") integration routine `numeric::odesolveGeometric`. Generally, `numeric::odesolve` is faster than the geometric integrator. However, `odesolveGeometric` preserves certain invariants of the sytem more faithfully.

Changes:

- The new option *`RelativeError`*`=tol` allows to set a precision goal independent of the working precision.

`numeric::odesolve2` – numerical solution of an ordinary differential equation

`numeric::odesolve2(f, t0, Y0, ..)` returns a function representing the numerical solution $Y(t)$ of the first order differential equation (dynamical system)

$$\frac{dY}{dt} = f(t,Y)\,, \quad Y(t_0) = Y_0\,, \quad t_0, t \in R\,, \quad Y_0, Y(t) \in C^n\,.$$

Call(s):

- `numeric::odesolve2(f, t0, Y0 <, method> <, `*`RelativeError`*` = tol> <, `*`Stepsize`*` = h> )`

Parameters:

`f` — procedure representing the vector field of the dynamical system
`t0` — numerical real value for the initial time t_0
`Y0` — list or 1-dimensional array of numerical values representing the initial value Y_0.

Options:

`method`	— name of the numerical scheme, see `numeric::odesolve`.
`RelativeError` `= tol`	— forces internal numerical Runge-Kutta steps to use stepsizes with local discretization errors below `tol`. This tolerance must be a numerical real value $\geq 10^{-\texttt{DIGITS}}$.
`Stepsize` `= h`	— switches off the internal error control and uses a Runge-Kutta iteration with constant stepsize `h`. `h` must be a numerical real value.

Return Value: a procedure.

Side Effects: The function returned by `numeric::odesolve2` is sensitive to the environment variable `DIGITS`, which determines the numerical working precision. It uses `option remember`.

Related Functions: `numeric::odesolve`

Details:

- The function generated by

  `Y := numeric::odesolve2(f, t0, Y0` <, options> `)`

  is essentially

  `Y := t -> numeric::odesolve(t0..t, f, Y0` <,options> `).`

- Numerical integration is launched, when `Y` is called with a real numerical argument. The call `Y(t)` returns the solution vector in a format corresponding to the type of the initial condition `Y0` with which `Y` was defined: `Y(t)` either yields a list or a 1-dimensional array.

  If `t` is not a real numerical value, then `Y(t)` returns an unevaluated function call.

- See the help page of `numeric::odesolve` for details on the parameters and the options.

- The options *`Alldata`*=`n` and *`Symbolic`* accepted by `numeric::odesolve` have no effect: `numeric::odesolve2` ignores these options.

- The function `Y` remembers all values it has computed. When calling `Y(t)` it searches its remember table for the time `T` closest to `t` and integrates from `T` to `t` using the previously computed `Y(T)` as initial value. This reduces the costs of a call considerably, if `Y` has to be evaluated many times (e.g., when plotting). However, the result `Y(t)` depends on the history of the MuPAD session! This can lead to unexpected side effects. Cf. example 3. We recommend to call `Y` only with a monotonically increasing (or decreasing) sequence of values `t` starting from `t0`. Further, the function must be re-initialized whenever `DIGITS` is increased. Cf. example 4.

- After the command `setuserinfo(Y,1)` information on the current integration interval is displayed by each call to `Y`.

Example 1. The numerical solution of the initial value problem $y' = t\sin(y)$, $y(0) = 2$ is represented by the following function $Y = [y]$:

```
>> f := (t, Y) -> [t*sin(Y[1])]: t0 := 0: Y0 := [2]:
>> Y := numeric::odesolve2(f, 0, [2])

                          proc Y(t) ... end
```

It starts numerical integration upon call with a numerical argument:

```
>> Y(-2), Y(0), Y(0.1), Y(PI + sqrt(2))

       [2.968232567], [2.0], [2.004541745], [3.141552691]
```

Calling `Y` with a symbolic argument yields an unevaluated call:

```
>> Y(t), Y(t + 5), Y(t^2 - 4)

                                      2
                   Y(t), Y(t + 5), Y(t  - 4)
```

```
>> eval(subs(%, t = PI))

           [3.13235701], [3.141592654], [3.141592611]
```

The numerical solution can be plotted. Note that `Y(t)` returns a list, so we plot the list element `Y(t)[1]`:

```
>> plotfunc2d(Y(t)[1], t = -5..5)
>> delete f, t0, Y0, Y:
```

Example 2. We consider the differential equation $y'' = y^2$ with initial conditions $y(0) = 0$, $y'(0) = 1$. The second order equation is converted to a first order system for $Y = [Y_1, Y_2] = [y, y']$:

$$Y_1' = Y_2\,, \quad Y_2' = Y_1^2\,, \quad Y_1(0) = 0\,, \quad Y_2(0) = 1\,.$$

```
>> f := (t, Y) -> [Y[2], Y[1]^2]: t0 := 0: Y0 := [0, 1]:
>> Y := numeric::odesolve2(f, t0, Y0):
>> Y(1), Y(PI)
```

```
      [1.087473533, 1.362851121], [1274.867468, 37166.5226]
```

```
>> delete f, t0, Y0, Y:
```

Example 3. We demonstrate a pitfall with the remember mechanism built into the functions returned by `numeric::odesolve2`. Consider the initial value problem $y' = t\,\sin(y)$, $y(0) = 2$:

```
>> DIGITS := 5:
   Y := numeric::odesolve2((t, Y) -> [t*sin(Y[1])], 0, [2]):
```

The following result is stored in the remember table of `Y`:

```
>> setuserinfo(Y, 1): Y(8.0)
```

```
 Info: integrating from t0=0 to t=8.0 using Y(t0)=[2]

                              [3.1416]
```

The following value `Y(4.1)` is obtained using the previously computed `Y(8.0)`, integrating backward in time from $t = 8.0$ to $t = 4.1$:

```
>> Y(4.1)
```

```
 Info: integrating from t0=8.0 to t=4.1 using Y(t0)=[3.1416]

                              [4.1634]
```

We erase the remember table (the fifth operand) of `Y`:

```
>> Y := subsop(Y, 5 = NIL):
```

The direct integration from the initial time $t_0 = 0$ to $t = 4.1$ yields a more accurate result than the previous computation:

```
>> Y(4.1)
```

```
 Info: integrating from t0=0 to t=4.1 using Y(t0)=[2]

                              [3.1413]
```

The reason for this phenomenon becomes obvious, when the solution of the ode is computed for various initial values $Y(0) = 2, 3, 4$:

```
>> A := numeric::odesolve2((t, Y) -> [t*sin(Y[1])], 0, [2]):
   B := numeric::odesolve2((t, Y) -> [t*sin(Y[1])], 0, [3]):
   C := numeric::odesolve2((t, Y) -> [t*sin(Y[1])], 0, [4]):
>> plotfunc2d(A(t)[1], B(t)[1], C(t)[1], t = 0..8, Grid = 25)
```

In fact, all solutions with initial values $Y(0)$ in the interval $[0, 2\,\pi]$ approach the same attracting point $Y(\infty) = \pi$. While numerical integration forward in time is a very stable process, it is virtually impossible to recover the correct solution curve integrating backward in time starting at a point close to the attractor.

```
>> setuserinfo(Y, 0): delete DIGITS, Y, A, B, C:
```

Example 4. We consider the system

$$x' = x + y\ ,\ y' = x - y\ ,\ x(0) = 1\ ,\ y(0) = \sqrt{-1}\,:$$

```
>> f := (t, Y) -> [Y[1] + Y[2], Y[1] - Y[2]]:
   Y := numeric::odesolve2(f, 0, [1, I]):
>> DIGITS := 5: Y(1)

             [3.5465 + 1.3683 I, 1.3683 + 0.80989 I]
```

Increasing `DIGITS` does not lead to a more accurate result because of the remember mechanism:

```
>> DIGITS := 15: Y(1)

 [3.54647799893142 + 1.36829578207979 I,

    1.36829578207979 + 0.80988643477182 I]
```

This is the previous value computed with 5 digits, printed with 15 digits. Indeed, only 5 digits are correct. For getting a result that is accurate to full precision one has to erase the remember table via `Y:=subsop(Y,5=NIL)`. Alternatively, one may create a new numerical solution with a fresh (empty) remember table:

```
>> Y := numeric::odesolve2(f, 0, [1, I]): Y(1)

 [3.54648242861716 + 1.36829887200859 I,

    1.36829887200859 + 0.80988468459998 I]

>> delete f, Y, DIGITS:
```

Changes:

- `numeric::odesolve2` is a new function.

`numeric::polyroots` – numerical roots of a univariate polynomial

`numeric::polyroots(p, ..)` returns numerical approximations of all real and complex roots of the univariate polynomial `p`.

Call(s):

- `numeric::polyroots(p <, FixedPrecision> <, SquareFree> <, Factor> )`

Parameters:

`p` — a univariate polynomial expression or a univariate polynomial of domain type `DOM_POLY`.

Options:

`FixedPrecision`	— launches a quick numerical search with fixed precision
`SquareFree`	— a squarefree factorization of `p` is computed before the numerical search starts
`Factor`	— a factorization of `p` is computed before the numerical search starts

Return Value: a list of numerical roots.

Side Effects: The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Related Functions: `RootOf`, `numeric::fsolve`, `numeric::polysysroots`, `numeric::realroot`, `numeric::realroots`, `polylib::realroots`, `solve`

Details:

- The coefficients may be real or complex numbers. Also symbolic coefficients are accepted if they can be converted to floats.
- The trivial polynomial $p = 0$ results in an error message. The empty list is returned for constant polynomials $p \neq 0$.
- Multiple roots are listed according to their multiplicities, i.e., the length of the root list coincides with the degree of p.

- The root list is sorted by `numeric::sort`.
- Up to roundoff effects, the numerical roots should be accurate to `DIGITS` significant digits, unless the option *`FixedPrecision`* is used.
- All floating point entries in `p` are internally approximated by rational numbers: `numeric::polyroots(p)` computes the roots of `numeric::rationalize(p, Minimize)`.
- For polynomial expressions in factored form, the numerical search is applied to each factor separately.
- With `setuserinfo(numeric::polyroots, 2)`, detailed information on the internal search is provided.
- It is recommended to use `polylib::realroots` if `p` is a real polynomial and only real roots are of interest.

Option <*`FixedPrecision`*>:

- This option provides the fastest way to obtain approximations of the roots by a numerical search with a fixed internal precision of 2 `DIGITS` decimal places.
- Note that badly isolated roots or multiple roots will usually not be approximated to `DIGITS` decimals when using this option. The problem of finding such roots is numerically ill-conditioned, i.e., such roots cannot be found to full precision with fixed precision arithmetic. Typically, a q-fold root will be approximated only to about 2`DIGITS`$/q$ decimal places. Cf. example 3.
- Without this option, `numeric::polyroots` internally increases the working precision until all roots are found to `DIGITS` decimal places.

Option <*`SquareFree`*>:

- With this option, a symbolic square free factorization is computed via `polylib::sqrfree(p)`. The numerical root finding algorithm is then applied to each square free factor.
- This option is recommended, when `p` is known to have multiple roots. Such roots force `numeric::polyroots` to increase the working precision internally increasing the costs of the numerical search. A square free factorization reduces the multiplicity of each root to one, speeding up the final numerical search. Cf. example 3.

- For polynomials with real rational coefficients, a square free factorization is always used, i.e., this option does not have any effect for such polynomials. For all other types of coefficients, a square free factorization may be costly and must be requested by this option.
- Multiple roots of `p` can be successfully dealt with by this option. However, for badly separated distinct roots the square free factorization will not improve the performance of the numerical search.

Option <*`Factor`*>:

- With this option, a symbolic factorization of `p` via `factor` is computed. The numerical root finding algorithm is then applied to each factor.
- This option is useful, when `p` can be successfully factorized (e.g., when `p` has multiple roots). The numerical search on the factors is much more efficient than the search on the original polynomial. On the other hand, symbolic factorization of `p` may be costly.

Example 1. Both polynomial expressions as well as `DOM_POLY` objects may be used to specify the polynomial:

```
>> numeric::polyroots(x^3 - 3*x - sqrt(2))

          [-1.414213562, -0.5176380902, 1.931851653]

>> numeric::polyroots(PI*z^4 + I*z + 0.1)

 [- 0.5936837297 - 0.3729248918 I, 0.1003181767 I,

    0.6455316068 I, 0.5936837297 - 0.3729248918 I]

>> numeric::polyroots(poly(x^5 - x^2, [x]))

 [- 0.5 - 0.8660254038 I, - 0.5 + 0.8660254038 I, 0.0, 0.0, 1.0]
```

Example 2. The following polynomial has exact coefficients:

```
>> p := poly((x - 1)*(x - PI)^3, [x]):

>> numeric::polyroots(p)

          [1.0, 3.141592654, 3.141592654, 3.141592654]
```

Note that roundoff errors in the coefficients of `p` have a dramatic effect on multiple roots:

```
>> p := poly((x - 1.0)*(x - float(PI))^3, [x]):

>> numeric::polyroots(p)

 [0.9999999995, 3.140177788 - 0.00244957836 I,

    3.140177788 + 0.00244957836 I, 3.144422386]
```

These are the roots of the rationalized polynomial

```
>> numeric::rationalize(p, Minimize)

       4                  3                 2
 poly(x  - 461286/44249 x  + 176627/4525 x  - 2515405/41498 x +

    1837649/59267, [x])

>> delete p:
```

Example 3. The multiple root `I/3` of the following polynomial can only be computed with restricted precision by fixed precision arithmetic:

```
>> p := poly((x^2 - 6*x +8)*(x - I/3)^5, [x]):

>> numeric::polyroots(p, FixedPrecision)

 [- 0.006109919675 + 0.3322082825 I,

    - 0.002938026857 + 0.3387257645 I,

    - 0.0007618302219 + 0.3271290976 I,

    0.004162114315 + 0.3378408591 I,

    0.005647662459 + 0.330762663 I, 2.0, 4.0]
```

Without the option *`FixedPrecision`,* the working precision is increased internally to compute better approximations:

```
>> setuserinfo(numeric::polyroots, 1): numeric::polyroots(p)

 Info: computing roots of factor poly(x^7 - .. , [x])
 Info: increasing working precision to DIGITS=20
 Info: increasing working precision to DIGITS=40
 Info: increasing working precision to DIGITS=80
 Info: increasing working precision to DIGITS=160
 Info: accepting last approximation

 [0.3333333333 I, 0.3333333333 I, 0.3333333333 I,

    0.3333333333 I, 0.3333333333 I, 2.0, 4.0]
```

A square free factorization reduces the multiplicity of the root `I/3`:

```
>> numeric::polyroots(p, SquareFree)

 Info: starting squarefree factorization
 Info: computing roots of factor poly(3*x - I, [x])
 Info: increasing working precision to DIGITS=20
 Info: computing roots of factor poly(x^2 - 6*x + 8, [x])
 Info: increasing working precision to DIGITS=20

 [0.3333333333 I, 0.3333333333 I, 0.3333333333 I,

    0.3333333333 I, 0.3333333333 I, 2.0, 4.0]

>> setuserinfo(numeric::polyroots, 0): delete p:
```

Example 4. The following polynomial has badly separated roots. `numeric::polyroots` does not manage to separate them properly:

```
>> p := poly(_mult((x - 1 - i/10^9) $ i=0..5), [x]):

>> numeric::polyroots(p)

 [0.9999999987, 1.000000003, 1.000000003, 1.000000003,

    1.000000003, 1.000000003]
```

One can preprocess the polynomial by a symbolic factorization:

```
>> numeric::polyroots(p, Factor)

 [1.0, 1.000000001, 1.000000002, 1.000000003, 1.000000004,

    1.000000005]
```

Alternatively, one can increase the working precision to separate the roots:

```
>> DIGITS := 20: numeric::polyroots(p)

 [1.0, 1.000000001, 1.000000002, 1.000000003, 1.000000004,

    1.000000005]

>> delete p, DIGITS:
```

Background:

- The numerical root finding algorithm implemented by `numeric::polyroots` is Laguerre's method: W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling: *Numerical Recipes in C*, Cambridge University Press, 1988.

Changes:

- `numeric::polyroots` is a new function.

`numeric::polysysroots` – numerical roots of a system of polynomial equations

`numeric::polysysroots(eqs, ..)` returns numerical approximations of all real and complex roots of the polynomial system of equations `eqs`.

Call(s):

- `numeric::polysysroots(eqs)`
- `numeric::polysysroots(eqs, vars)`

Parameters:

`eqs` — a polynomial equation or a list or a set of such equations. Also expressions and polynomials of domain type `DOM_POLY` are accepted wherever an equation is expected. They are interpreted as homogeneous equations.

`vars` — an unknown or a list or set of unknowns. Unknowns may be identifiers or indexed identifiers.

Return Value: a set of lists of equations. The set `{[]}` containing an empty list is returned, if no solutions can be computed.

Side Effects: The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Related Functions: `linsolve`, `numeric::fsolve`, `numeric::linsolve`, `numeric::polyroots`, `numeric::realroot`, `numeric::realroots`, `polylib::realroots`, `solve`

Details:

- The coefficients of the polynomials may contain symbolic parameters.
- If no unknowns are specified by `vars`, then `numeric::indets(eqs)` is used in place of `vars`.

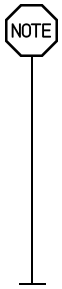
- The solution is a set of lists of solved equations of the form

$$[x_1 = value_1, x_2 = value_2, \dots],$$

  where $x_1, x_2, \dots$ are the unknowns. These simplified equations should be regarded as constraints on the unknowns. E.g., if an unknown x_1, say, does not turn up in the form $x_1 = \dots$ in the solution, then there is no constraint on this unknown and it is an arbitrary parameter. This holds true in general for all unknowns that do not turn up on the left hand side of the solved equations. Cf. example 2.

- The ordering of the unknowns in `vars` determines the ordering of the solved equations. If a *set* `vars` is used, then an internal ordering is used.

- If the solution set of `eqs` is not finite, then `numeric::polysysroots` may return solutions with some of the unknowns remaining as free parameters. In this case the representation of the solution depends on the ordering of the unknowns! Cf. example 3. Further, if higher degree polynomials are involved, then `numeric::polysysroots` may fail to compute roots. Cf. example 5. The same may happen, when `eqs` contains symbolic parameters.



- You may try `numeric::fsolve` to compute a single numerical root, if `numeric::polysysroots` cannot compute all roots of the system. Note, however, that `numeric::fsolve` does not accept symbolic parameters in the equations.

- We recommend to use `numeric::polyroots` to compute all roots of a single univariate polynomial with numerical coefficients.

- `numeric::polysysroots` is a hybrid routine: it calls the symbolic solver

  `solve`(`eqs`, `vars`, `BackSubstitution` = `FALSE`)

  and processes its symbolic result numerically via `solvelib::allvalues`. Cf. example 4.

Example 1. Equations, expressions as well as `DOM_POLY` objects may be used to specify the polynomials:

```
>> numeric::polysysroots(x^2 = PI^2, x)

             {[x = -3.141592654], [x = 3.141592654]}

>> numeric::polysysroots({x^2 + y^2 - 1, x^2 - y^2 = 1/2}, [x, y])

 {[x = -0.8660254038, y = -0.5], [x = -0.8660254038, y = 0.5],

    [x = 0.8660254038, y = -0.5], [x = 0.8660254038, y = 0.5]}
```

```
>> numeric::polysysroots({poly(x^2 + y + 1), y^2 + x = 1}, [x, y])
 {[x = -0.4533976515, y = -1.20556943], [x = 0.0, y = -1.0],

    [x = 0.2266988258 - 1.467711509 I,

    y = 1.102784715 + 0.6654569512 I],

    [x = 0.2266988258 + 1.467711509 I,

    y = 1.102784715 - 0.6654569512 I]}
```

Symbolic parameters are accepted:

```
>> numeric::polysysroots(x^2 + y + exp(z), [x, y])

                                1/2
 {[x = (- 1.0 y - 1.0 exp(z))   ],

                                      1/2
    [x = - 1.0 (- 1.0 y - 1.0 exp(z))   ]}
```

Example 2. The returned solutions may contain some of the unknowns remaining as free parameters:

```
>> numeric::polysysroots({x^2 + y^2 = z}, [x, y, z])

                      2 1/2                          2 1/2
      {[x = (z - 1.0 y )   ], [x = - 1.0 (z - 1.0 y )   ]}
```

Example 3. The ordering of the unknowns determines the representation of the solution, if the solution set is not finite. First, the following equation is solved for `x` leaving `y` as a free parameter:

```
>> numeric::polysysroots({x^3 = y^2}, [x, y])

         2 1/3                                      2 1/3
 {[x = (y )   ], [x = - (0.5 + 0.8660254038 I) (y )   ],

                                    2 1/3
    [x = - (0.5 - 0.8660254038 I) (y )   ]}
```

Reordering the unknowns leads to a representation with `x` as a free parameter:

```
>> numeric::polysysroots({x^3 = y^2}, [y, x])

                      3 1/2                3 1/2
              {[y = (x )   ], [y = - 1.0 (x )   ]}
```

Example 4. The symbolic solver produces a `RootOf` solution of the following system:

```
>> eqs := {y^2 - y = x, x^3 = y^3 + x}:
>> solve(eqs, BackSubstitution = FALSE)

                          2
 {[x = 0, y = 0], [x = y  - y, y =

               3       2            4     5
    RootOf(3 X3  - 2 X3  - X3 - 3 X3  + X3  + 1, X3)]}
```

Internally, `numeric::polysysroots` calls `solve` and processes this result numerically:

```
>> numeric::polysysroots(eqs, [x, y])

 {[x = -0.238328984, y = 0.6080324762], [x = 0.0, y = 0.0],

    [x = 0.8911259621, y = -0.5682349751],

    [x = 2.237302267, y = 2.077118343],

    [x = - 1.445049623 + 0.1279930535 I,

    y = 0.441542078 - 1.094745154 I],

    [x = - 1.445049623 - 0.1279930535 I,

    y = 0.441542078 + 1.094745154 I]}

>> delete eqs:
```

Example 5. The following equation is solved for the first of the specified unknowns:

```
>> eqs := y^5 - PI*y = x: solve(eqs, [x, y])

                              5
                       {[x = y  - y PI]}
```

`numeric::polysysroots` processes this output numerically:

```
>> numeric::polysysroots(eqs, [x, y])

                          5
                 {[x = y  - 3.141592654 y]}
```

The equation is solved for `y` when the unknowns are reordered. However, no simple representation of the solution exists, so a `RootOf` object is returned:

```
>> solve(eqs, [y, x])
```

$$\{[y = \mathtt{RootOf}(x + X5\ PI - X5^{5}, X5)]\}$$

The roots represented by the `RootOf` expression cannot be computed numerically, because the symbolic parameter `x` is involved:

```
>> numeric::polysysroots(eqs, [y, x])
```

$$\{[y = \mathtt{RootOf}(x + 3.141592654\ X6 - 1.0\ X6^{5}, X6)]\}$$

```
>> delete eqs:
```

Changes:

- `numeric::polysysroots` is a new function.

`numeric::quadrature` – numerical integration

`numeric::quadrature(f(x), x=a..b, ..)` computes a numerical approximation of $\int_a^b f(x)\,dx$.

Call(s):

- `numeric::quadrature(f(x), x=a..b, <, method=n> <, Adaptive=v> <, MaxCalls=m> )`

Parameters:

`f(x)` — expression in x
`x` — identifier or indexed identifier
`a,b` — real or complex numerical values or $\pm$`infinity`

Options:

`method=n` — `method` is the name of the underlying quadrature scheme, either *`GaussLegendre`*, *`GaussTschebyscheff`*, or *`NewtonCotes`*. Each quadrature rule can be used with an arbitray number of nodes specified by the positive integer n.

`Adaptive``=v` — `v` may be `TRUE` or `FALSE`. With `Adaptive=FALSE` the internal error control is switched off.

`MaxCalls``=m` — `m` must be a (large) positive integer or `infinity`. It is the maximal number of evaluations of the integrand, before `numeric::quadrature` gives up.

Return Value: A floating point number is returned, unless non-numerical symbolic objects in the integrand `f(x)` or in the boundaries `a,b` prevent numerical evaluation. In this case `numeric::quadrature` returns itself unevaluated. If numerical problems occur, then `FAIL` is returned.

Side Effects: The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Related Functions: `int`, `numeric::gldata`, `numeric::gtdata`, `numeric::int`, `numeric::ncdata`

Details:

- `numeric::quadrature` returns itself unevaluated, if the integrand `f(x)` contains symbolic objects apart from the integration variable `x` that cannot be converted to numerical values via `float`. Symbolic objects such as `PI` or `sqrt(2)` etc. are accepted.
- The integrand `f(x)` should be integrable in the Riemann sense. In particular, `f(x)` should be bounded on the integration interval `x=a..b`. Certain types of mild singularities are handled, but numerical convergence is not guaranteed and will be slow in most cases. Also discontinuities and singularities of (higher) derivatives of `f(x)` slow down numerical convergence. For integrands with irregular points it is recommended to split the integration into several parts, using subintervals on which the integrand is smooth. Cf. example 4.
- Integrands given by user-defined procedures can be handled. Cf. Examples 4 and 5.
- `numeric::quadrature` returns itself unevaluated, if the boundaries `a,b` contain symbolic objects that cannot be converted to numerical values via `float`. Symbolic objects such as `PI` or `sqrt(2)` etc. as well as `infinity` and `-infinity` are accepted.
- For infinite ranges the user should make sure that the integral exists! If `f(x)` does not decay as fast as $O(|x|^{-2})$ at infinity, then convergence may be slow.

- Boundaries $a > b$ are accepted. Note:

$$\texttt{quadrature(f(x), x = a..b)} = -\texttt{quadrature(f(x), x = b..a)}.$$

- For complex values of `a,b` the integration is to be understood as a contour integral along a straight line from `a` to `b`. Complex boundaries must not involve `infinity`.

- Multi-dimensional integration such as

  `numeric::quadrature(numeric::quadrature(f(x,y), y=A(x)..B(x)), x=a..b)`

  is possible. Cf. examples 3 and 5.

- Internally an adaptive mechanism based on a fixed quadrature rule specified by `method=n` is used. It tries to keep the relative quadrature error of the result below $10^{-\texttt{DIGITS}}$. The last digit(s) of the result may be incorrect due to roundoff effects.

- The routine `numeric::quadrature` is purely numerical: no symbolic check for singularities etc. is carried out.

Option `<method = n>`:

- Usually there is no need to use this option to change the default method *`GaussLegendre`*=`n` with `n=20,40,80` or `160`, depending on the precision goal determined by the environment variable `DIGITS`. Due to the corresponding high quadrature orders 40, 80, 160 or 320, respectively, the default settings are suitable for high precision computations.

- With *`GaussLegendre`*=`n` an adaptive version of Gauss-Legendre quadrature with n nodes is used.

  For `DIGITS`≤ 200 the weights and abscissae of Gaussian quadrature with $n = 20$, 40, 80 and 160 nodes are available and the integration starts immediately.

  For `DIGITS`>200 the routine `numeric::gldata` is called to compute the Gaussian data before the actual integration starts. This will be noted by some delay in the first call of `numeric::quadrature`.

  For `DIGITS`$\gg 200$ it is recommended not to use the default setting but to use *`GaussLegendre`*=`n` with sufficiently high `n` (approximately `DIGITS`) instead.

- With `GaussTschebyscheff=n` non-adaptive Gauss-Tschebyscheff quadrature with n nodes is used. This method may only be used in conjunction with `Adaptive=FALSE`.

- With `NewtonCotes=n` an adaptive version of Newton-Cotes quadrature with n nodes is used. Note that these quadrature rules become numerically unstable for large n ($\gg 10$).

- Following alternative names for the methods are accepted:

  `GaussLegendre`, *`Gauss`*, *`GL`*,

  `GaussTschebyscheff`, *`GT`*, *`GaussChebyshev`*, *`GC`*,

  `NewtonCotes`, *`NC`*.

Option <`Adaptive= v`>:

- The default setting is `Adaptive=TRUE`. An adaptive quadrature scheme with automatic control of the quadrature error is used.
- The call `numeric::quadrature(f(x), x=a..b, method=n, Adaptive=FALSE)` returns the quadrature sum $(b-a)\ \sum_{i=1}^{n} B_i\, f(a+C_i\,(b-a))$ approximating $\int_a^b f(x)\ dx$ without any control of the quadrature error. The weights B_i and abscissae C_i are determined by the quadrature rule given by `method=n`. For the methods *`GaussLegendre`*, *`GaussTschebyscheff`* and *`NewtonCotes`* these data are available via `numeric::gldata`, `numeric::gtdata` and `numeric::ncdata`, respectively.
- `Adaptive=FALSE` may only be used in conjunction with `method=n`.
- Usually there is no need to switch off the internal adaptive quadrature via `Adaptive=FALSE`. This option is meant to give easy access to standard non-adaptive quadrature rules of Gauss-Legendre, Gauss-Tschebyscheff and Newton-Cotes type, respectively. The user may want to build his own adaptive quadrature based on these non-adaptive rules. Cf. example 6.

Option <*`MaxCalls`*`= m`>:

- `numeric::quadrature` gives up after `m` evaluations of the integrand. When called interactively, `numeric::quadrature` returns the approximation it has computed so far and issues a warning. Cf. example 7. When called from inside a procedure, `numeric::quadrature` returns `FAIL`.
- The default value is `m=MAXDEPTH*n`. The environment variable `MAXDEPTH` (default value 500) represents the maximal recursive depth of a function. `n` is the number of nodes of the basic quadrature rule specified by the optional argument `method=n`. If no such method is specified by the user, then Gauss-Legendre quadrature is used with `n`$=20$ for `DIGITS`≤ 10, `n`$=40$ for $10<$`DIGITS`≤ 50, `n`$=80$ for $50<$`DIGITS`≤ 100, `n`$=160$ for $100<$`DIGITS`. Thus, for `DIGITS=10`, the default setting is *`MaxCalls`*$=10\,000$.
- The default value of `m` ensures that the *`MaxCalls`* limit is reached before `numeric::quadrature` reaches its maximal internal recursive depth. Specifying *`MaxCalls`*`=infinity` removes this restriction and `numeric::quadrature` computes until it obtains an approximation with about `DIGITS` correct digits or until it runs into an internal error. The typical error that may occur is the evaluation of the integrand at a singularity. There also is the

danger of `numeric::quadrature` reaching its maximal internal recursive depth. When called interactively, `numeric::quadrature` returns the approximation it has computed so far and issues a warning. Cf. example 8. When called from inside a procedure, `numeric::quadrature` returns `FAIL`.

Example 1. We demonstrate the standard calls for adaptive numerical integration:

```
>> numeric::quadrature(exp(x^2), x=-1..1)

                              2.925303492

>> numeric::quadrature(max(1/10, cos(PI*x)), x=-2..0.0123)

                             0.7521024709

>> numeric::quadrature(exp(-x^2), x=-2..infinity)

                              1.768308316
```

The precision goal is set by `DIGITS`:

```
>> DIGITS := 50:
>> numeric::quadrature(besselJ(0, x), x=0..PI)

       1.3475263146739901712314731279612149642205400522774
```

Note that due to the internal adaptive mechanism the choice of different methods should not have any significant effect on the quadrature result:

```
>> DIGITS := 10:
>> numeric::quadrature(sin(x)/x, x=-1..10, GaussLegendre=5),
   numeric::quadrature(sin(x)/x, x=-1..10, GaussLegendre=160),
   numeric::quadrature(sin(x)/x, x=-1..10, NewtonCotes=8)

              2.604430665, 2.604430665, 2.604430665
```

Example 2. The user should make sure that the integrand is well defined and sufficiently regular. The following fails, because Newton-Cotes quadrature tries to evaluate the integrand at the boundaries:

```
>> numeric::quadrature(sin(x)/x, x=0..1, NewtonCotes=8)

 Error: Division by zero [_power];
 during evaluation of 'Quadsum'
```

One may cure this problem be assigning a value to `f(0)`. The integrand is passed to the integrator as `hold(f)` to prevent premature evaluation of `f(x)` to `sin(x)/x`. Internally, `numeric::quadrature` replaces `x` by numerical values and then evaluates the integrand. Note that one has to define `f(0.0):= 1` rather than `f(0):= 1`:

```
>> f := x -> sin(x)/x: f(0.0) := 1:
>> numeric::quadrature(hold(f)(x), x=0..1, NewtonCotes=8)
```

```
                              0.9460830704
```

The default method (Gauss-Legendre quadrature) does not evaluate the integrand at the end points:

```
>> numeric::quadrature(sin(x)/x, x=0..1)
```

```
                              0.9460830704
```

Nevertheless, problems may still arise for improper integrals:

```
>> numeric::quadrature(ln(1 - cos(x)), x=0..PI)
```

```
 Error: singularity [ln]
```

In this example the integrand is evaluated close to 0. Due to numerical cancellation $1 - \cos(x)$ may yield `0.0` for non-zero x. A numerically stable representation is $1 - \cos(x) = 2\sin(x/2)$^2:

```
>> numeric::quadrature(ln(2*sin(x/2)^2), x=0..PI)
```

```
                              -2.17758609
```

Note that convergence is rather slow, because the integrand is not bounded.

```
>> delete f:
```

Example 3. We demonstrate multi-dimensional quadrature:

```
>> Q := numeric::quadrature: Q(Q(exp(x*y), x=0..y), y=0..1)
```

```
                              0.6589510757
```

Also more complex types of nested calls are possible. We use numerically defined functions

```
>> b := y -> Q(exp(-t^2-t*y), t=y..infinity):
```

and

```
>> f := y -> cos(y^2) +  Q(exp(x*y), x=0..b(y)):
```

for the following quadrature:

```
>> Q(f(y), y=0..1)
```

```
                                1.261578952
```

Multi dimensional quadrature is computationally expensive. Note that a call to `numeric::quadrature` evaluates the integrand at least $3n$ times, where n is the number of nodes of the internal quadrature rule (by default, $n = 20$ for `DIGITS`≤ 10). The following triple quadrature would call the the exp function no less than $(3 \times 20)^3 = 216\,000$ times!

```
>> Q(Q(Q(exp(x*y*z), x=0..y+z), y=0..z), z=0..1)
```

For low precision goals low order quadrature rules suffice. In the following we reduce the computational costs by using Gauss-Legendre quadrature with 5 nodes. We use the shorthand notation `GL` to specify the `GaussLegendre` method:

```
>> DIGITS := 4:
>> Q(Q(Q(exp(x*y*z), x=0..y+z, GL=5), y=0..z, GL=5), z=0..1, GL=5)
```

```
                                    0.665
```

```
>> delete Q, b, f, DIGITS:
```

Example 4. We demonstrate how integrands given by user-defined procedures should be handled. The following integrand

```
>> f := proc(x) begin
         if x<1 then sin(x^2) else cos(x^5) end_if
       end_proc:
```

cannot be called with a symbolic argument:

```
>> f(x)
```

```
 Error: Can't evaluate to boolean [_less];
 during evaluation of 'f'
```

Consequently, one must use `hold` to prevent premature evaluation of `f(x)`:

```
>> numeric::quadrature(hold(f)(x), x=-1..PI/2)
```

```
                                0.5354101317
```

Note that the above integrand is discontinuous at $x = 1$, causing slow convergence of the numerical quadrature. It is much more efficient to split the integral into two subquadratures with smooth integrands:

```
>> numeric::quadrature(sin(x^2), x=-1..1) +
   numeric::quadrature(cos(x^5), x=1..PI/2)
```

```
                          0.5354101318
```

See example 5 for multi-dimensional quadrature of user-defined procedures.

```
>> delete f:
```

Example 5. The following integrand

```
>> f := proc(x, y) begin
          if x<y then x-y else (x-y) + (x-y)^5 end_if
        end_proc:
```

can only be called with numerical arguments and must be delayed twice for 2-dimensional quadrature:

```
>> Q := numeric::quadrature:
>> Q(Q(hold(hold(f))(x, y), x=0..1), y=0..1)
```

```
                          0.0238095238
```

Note that delaying the integrand via `hold` will not work for triple or higher-dimensional quadrature! The user can handle this by making sure that the integrand can also be evaluated for symbolic arguments:

```
>> f := proc(x, y, z)
        begin
          if not testtype([args()], Type::ListOf(Type::Numeric))
             then return(procname(args()))
          end_if;
          if x^2 + y^2 + z^2 <= 1
             then return(1)
             else return(0)
          end_if
        end_proc:
```

Note that this function is not continuous, implying slow convergence of the numerical quadrature. For this reason we use a low precision goal of only 2 digits and reduce the costs by using a low order quadrature rule:

```
>> DIGITS := 2:
>> Q(Q(Q(f(x, y, z), x=0..1, GL=5), y=0..1, GL=5), z=0..1, GL=5)
```

```
                               0.53
```

```
>> delete f, Q, DIGITS:
```

Example 6. The following example uses non-adaptive Gauss-Tschebyscheff quadrature with an increasing number of nodes. The results converge quickly to the exact value:

```
>> a := exp(x)/sqrt(1 - x^2), x=-1..1:
>> numeric::quadrature(a, Adaptive=FALSE, GT=n) $ n=3..7

 3.97732196, 3.977462635, 3.977463259, 3.977463261, 3.977463261

>> delete a:
```

Example 7. The improper integral $\int_0^1 x^{-9/10}\,dx = 10$ exists. Numerical convergence, however, is rather slow because of the singularity at $x = 0$:

```
>> numeric::quadrature(x^(-9/10), x=0..1)

 Warning: Precision goal not achieved after 10000 function call\
 s!
 Increase MaxCalls and try again for a more accurate result. [n\
 umeric::quadrature]

                              9.998221196
```

We remove the limit for the number of function calls and let `numeric::quadrature` grind along until a result is found. The time for the computation grows accordingly, the last digit is incorrect due to roundoff effects:

```
>> numeric::quadrature(x^(-9/10), x=0..1, MaxCalls=infinity)

                              9.999999993
```

Example 8. The following integral does not exist in the Riemann sense, because the integrand is not bounded:

```
>> numeric::quadrature(1/x, x=-1..1)

 Warning: Precision goal not achieved after 10000 function call\
 s!
 Increase MaxCalls and try again for a more accurate result. [n\
 umeric::quadrature]

                              93.14572971
```

We increase *`MaxCalls`*. There is no convergence of the numerical algorithm, because the integral does not exist. Consequently, some internal problem must arise: `numeric::quadrature` reaches its maximal recursive depth:

```
>> numeric::quadrature(1/x, x=-1..1, MaxCalls=infinity)

 Warning: Precision goal not achieved after MAXDEPTH=500 recurs\
 ive calls!
 There may be a singularity of 1/x close to x=1.466369455e-
149.
 Increase MAXDEPTH and try again for a more accurate result! [a\
 daptiveQuad]

                              343.1078544
```

Changes:

- Handling of infinite ranges was added. Quadrature now stops after m function calls.
- The option *MaxCalls*=m was introduced to set the maximal number of calls.

---

numeric::rationalize – approximate a floating point number by a rational number

numeric::rationalize(object, ..) replaces all floating point numbers in object by rational numbers.

Call(s):

- numeric::rationalize(object <, mode> <, digits>)

Parameters:

object — an arbitrary MuPAD object

Options:

mode — either *Exact*, or *Minimize*, or *Restore*. This controls the strategy for approximating floating point numbers by rational numbers.

digits — a positive integer (the number of decimal digits) not bigger than the environment variable DIGITS. It determines the precision of the rational approximation.

Return Value: If the argument is an object of some kernel domain, then it is returned with all floating point operands replaced by rational numbers. An object of some library domain is returned unchanged.

Overloadable by: `object`

Side Effects: The function is sensitive to the environment variable `DIGITS`.

Details:

- An object of a library domain, characterized by

  $$\texttt{domtype}(\texttt{extop}(\texttt{object}, 0)) = \texttt{DOM\_DOMAIN},$$

  is returned unchanged. For all other objects `numeric::rationalize` is applied recursively to all operands. Objects of library domains can be rationalized, if the domain has an appropriate `map` method. Cf. example 5.

Option <`digits`>:

- A floating point number f is approximated by a rational number r satisfying $|f - r| < \epsilon\,|f|$.

  With the options ***`Exact`*** and ***`Minimize`*** the guaranteed precision is $\epsilon = 10^{-\texttt{digits}}$. With ***`Restore`*** the guaranteed precision is only $\epsilon = 10^{-\texttt{digits}/2}$.

- The default precision is `digits=DIGITS`.

- The user defined precision must not be larger then the internal floating point precision set by `DIGITS`: an error occurs for `digits>DIGITS`.

Option <*`Exact`***>:**

- Any real floating point number $f \neq 0.0$ has a unique representation

  $$f = \text{sign}(f) \times \mathit{mantissa} \times 10^{\mathit{exponent}}$$

  with integer exponent and $1.0 \leq \mathit{mantissa} < 10.0$. With the option ***`Exact`*** the float mantissa is replaced by the rational approximation

  $$\frac{\texttt{round}(\mathit{mantissa} \times 10^{\texttt{digits}})}{10^{\texttt{digits}}}\,.$$

  This guarantees a relative precision of `digits` significant decimals of the rational approximation.

- This is the default strategy, so there is no real need to pass this option as a parameter to `numeric::rationalize`.

Option <`Minimize`>:

- This strategy tries to minimize the complexity of the rational approximation, i.e., numerators and denominators are to be small.
- The guaranteed precision of the rational approximation is `digits`.
- Cf. example 3.

Option <`Restore`>:

- This strategy tries to restore rational numbers obtained after *elementary* arithmetical operations applied to floating point numbers. E.g., for rational `r` the float division `f = 1/float(r)` introduces additional round-off, which the `Restore` algorithm tries to eliminate: `numeric::rationalize(f, Restore) = 1/r`. This strategy, however, is purely heuristic and will not succeed, when significant round-off is caused by arithmetical float operations!
- The guaranteed precision of the rational approximation is only `digits/2`!

- Cf. example 4.

Example 1. `numeric::rationalize` is applied to each operand of a composite object:

```
>> numeric::rationalize(0.2*a+b^(0.7*I))

                            a    7/10 I
                            - + b
                            5

>> numeric::rationalize([{poly(0.2*x, [x]), sin(7.2*PI) + 1.0*I},
                         exp(3 + ln(2.0*x))])

 -- {                       / 36 PI \     }                    -
-
 |  { poly(1/5 x, [x]), sin| ----- | + I }, exp(ln(2 x) + 3)  |
 -- {                       \   5   /     }                    -
-
```

Example 2. We demonstrate the default strategy *Exact*:

```
>> numeric::rationalize(12.3 + 0.5*I),
   numeric::rationalize(0.33333),
   numeric::rationalize(1/3.0)

     123/10 + 1/2 I, 33333/100000, 33333333333/100000000000

>> numeric::rationalize(10^12/13.0),
   numeric::rationalize(10^(-12)/13.0)

       76923076923, 76923076923/10000000000000000000000000
```

We reduce the precision of the approximation to 5 digits:

```
>> numeric::rationalize(10^12/13.0, 5),
   numeric::rationalize(10^(-12)/13.0, 5)

            76923100000, 769231/10000000000000000000
```

Example 3. We demonstrate the strategy *Minimize* for minimizing the complexity of the resulting rational number:

```
>> numeric::rationalize(1/13.0, 5),
   numeric::rationalize(1/13.0, Minimize, 5),
   numeric::rationalize(0.333331, 5),
   numeric::rationalize(0.333331, Minimize, 5),
   numeric::rationalize(14.285, 5),
   numeric::rationalize(14.2857, Minimize, 5),
   numeric::rationalize(1234.1/56789.2),
   numeric::rationalize(1234.1/56789.2, Minimize)

 769231/10000000, 1/13, 333331/1000000, 1/3, 2857/200, 100/7,

    21731244673/1000000000000, 12341/567892
```

We compute rational approximations of π with various precisions:

```
>> numeric::rationalize(float(PI), Minimize, i) $ i = 1..10

 3, 22/7, 22/7, 355/113, 355/113, 355/113, 355/113,

    102573/32650, 104348/33215, 208341/66317
```

Example 4. We demonstrate the strategy *Restore* for restoring rational numbers after elementary float operations. In many cases also the *Minimize* strategy restores:

```
>> numeric::rationalize(1/7.3, Exact),
   numeric::rationalize(1/7.3, Minimize),
   numeric::rationalize(1/7.3, Restore)

            13698630137/100000000000, 10/73, 10/73
```

However, using *Restore* improves the chances of recovering from round-off effects:

```
>> numeric::rationalize(10^12/13.0, Minimize),
   numeric::rationalize(10^12/13.0, Restore)

                 76923076923, 1000000000000/13

>> numeric::rationalize(123.456/12.34567, Minimize),
   numeric::rationalize(123.456/12.34567, Restore)

                529097/52910, 12345600/1234567
```

In some cases *Restore* manages to recover from round-off error propagation in composite arithmetical operations:

```
>> x := 125/12.34567: y := 123/12.34567: z := (x^2 - y^2)/(x + y)

                          0.1620001183

>> numeric::rationalize(z, Minimize),
   numeric::rationalize(z, Restore)

                 35612/219827, 200000/1234567
```

The result with *Restore* corresponds to exact arithmetic:

```
>> rx := numeric::rationalize(x, Restore):
>> ry := numeric::rationalize(y, Restore):
>> (rx^2 - ry^2)/(rx + ry)

                          200000/1234567
```

Note that an approximation with *Restore* may have a reduced precision of only `digits/2`:

```
>> x := 1.0 + 1/10^6:
>> numeric::rationalize(x, Exact),
   numeric::rationalize(x, Restore)

                      1000001/1000000, 1

>> delete x, y, z, rx, ry:
```

Example 5. The floats inside objects of library domains are not rationalized directly. However, for most domains the corresponding `map` method can forward `numeric::rationalize` to the operands:

```
>> Dom::Multiset(0.2, 0.2, 1/5, 0.3)

                {[0.3, 1], [0.2, 2], [1/5, 1]}

>> numeric::rationalize(%), map(%, numeric::rationalize, Restore)

      {[0.3, 1], [0.2, 2], [1/5, 1]}, {[1/5, 3], [3/10, 1]}
```

Background:

- Continued fraction (CF) expansion is used with the options *`Minimize`* and *`Restore`*.
- With *`Minimize`* the first CF approximation satisfying the precision criterion is returned.
- The *`Restore`* algorithm stops, when large coefficients of the CF expansion are found.

Changes:

- `numeric::rationalize` used to be `sharelib::rational`.
- Option *`Approx`* is now called *`Minimize`*. The option *`Restore`* was introduced.

`numeric::realroot` – numerical search for a real root of a real univariate function

`numeric::realroot(f(x), x=a..b, ..)` computes a numerical real root of $f(x)$ in the interval $[a, b]$.

Call(s):

- `numeric::realroot(f(x), x = a..b <, SearchLevel = s>)`

Parameters:

`f(x)` — an arithmetical expression in one unknown `x`. Alternatively, an equation `f1(x)=f2(x)` equivalent to the expression `f1(x)-f2(x)`.

`x` — an identifier or an indexed identifier

`a, b` — finite real numerical values

Options:

SearchLevel = s — s is a small non-zero integer. It controls the internal refinement of the search.

Return Value: a single numerical real root of domain type DOM_FLOAT. If no solution is found, then FAIL is returned.

Side Effects: The function is sensitive to the environment variable DIGITS, which determines the numerical working precision.

Related Functions: numeric::fsolve, numeric::polyroots, numeric::realroots, polylib::realroots, solve

---

Details:

- The expression f(x) must not contain symbolic objects other than the indeterminate x that cannot be converted to numerical values via float. Symbolic objects such as PI or sqrt(2) etc. are accepted. The same holds true for the boundaries a,b of the search interval.
- The function must produce real values. If float(f(x)) does not yield real floating point numbers for all real floating point numbers x from the interval [a,b], then internal problems may occur. Cf. example 5.
- numeric::realroot never tries to evaluate f(x) outside the search interval. Consequently, singularities outside the interval do not cause any problems. In many cases also singularities inside the interval do not affect the numerical search. However, numeric::realroot is not guaranteed to work in such a case. An error may occur, if the internal search accidentally hits a singularity. Cf. example 5.
- Up to round-off effects numerical roots r with $|r| \geq 10^{-\mathtt{DIGITS}}$ are computed to a relative precision of DIGITS significant decimal places. Roots of smaller absolute size are computed to an absolute precision of $10^{-2\,\mathtt{DIGITS}}$. These precision goals are not achieved, if significant round-off occurs in the numerical evaluation of $f(x)$.
- If f takes opposite signs at the endpoints a,b of the search interval and does not have zero-crossing singularities, then numeric::realroot is bound to find a root in the interval [a,b].
- User defined functions can be handled. Cf. example 2.
- numeric::realroot approximates a point where f(x) changes its sign. This is a root only if the function f is continuous. Cf. example 3.

- setuserinfo(numeric::realroot,1) provides information on the internal search.

- Note that `numeric::realroots` may be used to isolate *all* real roots. However, this function is much slower than `numeric::realroot`, if `f` is not a polynomial.
- For univariate polynomials we recommend to use `polylib::realroots` rather than `numeric::realroot`.

Option <*SearchLevel*= s>:

- The nonnegative integer s controls the internal refinement of the search. The default value is $s = 1$. Increasing s increases the chance of finding roots that are difficult to detect numerically. Cf. example 6.
- Note that increasing s by 1 may quadruple the time before `FAIL` is returned, if no real root is found. For this reason we recommend to restrict s to small values ($s \leq 5$, say).

Example 1. The following functions assume different signs at the boundaries, so the searches are bound to succeed:

```
>> numeric::realroot(x^3 - exp(3), x = -PI..10)

                          2.718281829

>> numeric::realroot(exp(-x[1]) = x[1], x[1] = 0..1)

                         0.5671432904
```

Example 2. The following function cannot be evaluated for non-numerical `x`. So one has to delay evaluation via `hold`:

```
>> f := proc(x) begin
         if x<0 then 1 - x else exp(x) - 10*x end_if
        end_proc:
>> numeric::realroot(hold(f)(x), x = -10..10)

                         0.1118325592

>> delete f:
```

Example 3. `numeric::realroot` approximates a point, where $f(x)$ changes its sign. For the following function this happens at the discontinuity $x = 1$:

```
>> f := proc(x) begin if x<1 then -1 else x end_if end_proc:
>> numeric::realroot(hold(f)(x), x = 0..3)

                                1.0

>> delete f:
```

Example 4. The following function does not have a real root. Consequently, `numeric::realroot` fails:

```
>> numeric::realroot(x^2 + 1, x = -2..2)

                               FAIL
```

The following function does not have a real root in the search interval:

```
>> numeric::realroot(x^2 - 1, x =  2..3)

                               FAIL
```

Example 5. The following function is complex valued for $x^2 < 3.5$. An error occurs, when the internal search hits such a point:

```
>> numeric::realroot(ln(x^2 - 3.5), x = -2..3)

 Error: Complex arguments are not allowed in comparisons;
 during evaluation of 'numeric::BrentFindRoot'
```

The singularity at $x = 2$ does not cause any problem in the following call:

```
>> numeric::realroot((x-1)/(x-2), x = -10..PI)

                                1.0
```

However, the singularity may be hit accidentally in the internal search:

```
>> numeric::realroot((x-1)/(x-2), x = -10..14)

 Error: Division by zero [_power];
 during evaluation of 'f'
```

Example 6. The following function has a root close to 1.0, which is difficult to detect. With the default search level $s = 1$ this root is not found:

```
>> f := 2 - exp(-100*(x - 2)^2) - 2*exp(-1000*(x - 1)^2):
>> numeric::realroot(f, x = 0..5)

                                FAIL
```

The root is detected with an increased search level:

```
>> numeric::realroot(f, x = 0..5, SearchLevel = 3)

                                 1.0

>> delete f:
```

Background:

- A mixture of bisectioning, secant steps and quadratic interpolation is used by `numeric::realroot`.

Changes:

- `numeric::realroot` is a new function.

`numeric::realroots` – isolate intervals containing real roots of an expression

`numeric::realroots(f(x), ..)` returns a list of intervals in which real roots of `f(x)` may exist.

Call(s):

- `numeric::realroots(f(x), x = a..b <, eps> <, Merge>)`

Parameters:

`f(x)` — an expression in one indeterminate `x`. Alternatively, an equation `f1(x)=f2(x)` equivalent to `f(x)=f1(x)-f2(x)`.

`x` — an identifier or an indexed identifier

`a, b` — finite real numbers or numerical expressions satisfying $a < b$

Options:

`eps` — a (small) positive real numerical value defining the precision goal

`Merge` — makes `numeric::realroots` merge adjacent intervals to larger intervals

Return Value: a list $[[a_1, b_1], [a_2, b_2], \ldots]$ of distinct floating point intervals $[a_i, b_i] \subset [a, b]$ which may contain roots of $f(x)$. The empty list is returned, if no root exists in the search interval.

Side Effects: The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

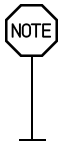
Related Functions: `Dom::Interval`, `numeric::fsolve`, `numeric::polyroots`, `numeric::realroot`, `polylib::realroots`, `solve`

Details:

- All intervals returned by `numeric::realroots` have length $b_i - a_i < eps$ with a default value $eps = 0.01$. The absolute precision eps of the root isolation may be redefined using the optional parameter `eps`.
- The intervals returned by `numeric::realroots` may contain roots. However, this is not conclusive: some intervals may contain no root. Cf. example 5.

- The complement $[a, b] \setminus \cup_i [a_i, b_i]$ of the subintervals $[a_i, b_i]$ returned by `numeric::realroots` is guaranteed not to contain any real roots. In particular, from the return value [ ] one may positively conclude that no root exists in the search interval $[a, b]$. Cf. example 2.
- Symbolic parameters in $f(x)$ are not allowed: `float(f(x))` must evaluate to a floating point number for all x from the interval $[a, b]$. Also `float(a)` and `float(b)` must produce floating point numbers.
- $f(x)$ must not produce non-real values for real input x from the interval $[a, b]$. Note that this may happen, if $f(x)$ involves non-integer powers (such as sqare roots) or logarithms.

- The expression `f(x)` must be suitable for interval arithmetic. In particular, MuPAD must be able to evaluate `f(Dom::Interval([a,b]))`. Note that not all MuPAD functions support this kind of arithmetic.



- For non-polynomial expressions $f(x)$ this routine is rather slow. It is fast for polynomial expressions.

Option <eps>:

- The default value is `eps`$= 0.01$. User defined precision goals must satisfy `eps`$\geq 10^{-\texttt{DIGITS}}$.

- For non-polynomial expressions computations with small precision goals may need some time!

Option <*Merge*>:

- By default all isolating intervals $[a_i, b_i]$ satisfy $b_i - a_i < eps$ where eps is the precision goal. However, adjacent intervals $[a_i, b_i], [a_{i+1}, b_{i+1}]$ with $b_i = a_{i+1}$ may be produced. This option combines such intervals to larger intervals $[a_i, b_{i+1}]$. Cf. examples 3 and 4.

Example 1. The following expression has integer zeros. The solutions in the specified interval are approximated to the default precision 0.01:

```
>> numeric::realroots(sin(PI*x), x = -2..sqrt(2))

 [[-2.0, -1.99], [-1.005969517, -0.9993206625],

    [-0.001992470411, 0.004656384203],

    [0.9953357217, 1.001984576]]
```

The following equation is solved with an absolute precision of 4 digits:

```
>> numeric::realroots(x*sin(x) = exp(-x), x = -1..1, 0.001)

                  [[0.7265625, 0.7275390625]]
```

Example 2. The following expression does not have a real root:

```
>> numeric::realroots(exp(x) + x^2, x = -100..100)

                                []
```

Example 3. We demonstrate the option *Merge*. If interval arithmetic can not isolate roots to the desired precision *eps* (default 0.01), then adjacent intervals are returned, each of length smaller than *eps*. This happens in the following example:

```
>> numeric::realroots(ln(x^2 -2*x + 2) = 0, x = -10..10)
```

```
 [[0.869140625, 0.87890625], [0.87890625, 0.888671875],

    [0.888671875, 0.8984375], ...,

    [1.123046875, 1.1328125], [1.1328125, 1.142578125]]
```

With the option *Merge* these intervals are combined to a single larger interval:

```
>> numeric::realroots(ln(x^2 -2*x + 2) = 0, x = -10..10, Merge)

                  [[0.869140625, 1.142578125]]
```

Example 4. The following expression has infinitely many solutions $x = 1/n$ with $n = 1, 2, \ldots$ in the search interval $[0, 1]$:

```
>> numeric::realroots(sin(PI/x), x = 0..1, 0.1)

 [[0.0, 0.05625], [0.05625, 0.1125], [0.1125, 0.16875],

    [0.16875, 0.225], [0.225, 0.28125], [0.28125, 0.3375],

    [0.45, 0.50625], [0.9, 1.0]]
```

The first of the following intervals contains infinitely many roots:

```
>> numeric::realroots(sin(PI/x), x = 0..1, 0.1, Merge)

          [[0.0, 0.3375], [0.45, 0.50625], [0.9, 1.0]]
```

Example 5. The following equation has no root close to 0. However, interval arithmetic does not produce realistic values of $\sin(\pi x)/x$ for small intervals containing 0, so an isolating interval around 0 is returned:

```
>> numeric::realroots(sin(PI*x)/x = 0, x = -1..1.2)

 [[-1.0, -0.99], [-0.0062109375, 0.00234375],

    [0.9946875, 1.003242188]]
```

A similar phenomenon occurs with x^x ($= e^{x \ln(x)}$) in a neighbourhood of $x = 0$. An isolating interval around 0 is returned, although no solution exists there:

```
>> numeric::realroots(x^x*cos(PI*x) = tan(x), x = 0..1)

            [[0.0, 0.0078125], [0.328125, 0.3359375]]
```

This cannot be cured by increasing the precision goal:

```
>> numeric::realroots(x^x*cos(PI*x) = tan(x), x = 0..1,
                      10^(-DIGITS))

     [[0.0, 5.820766091e-11], [0.3334737903, 0.3334737903]]
```

Background:

- Let X be a subset of the real numbers. Interval arithmetic produces a set $f(X)$ such that the set of image values $\{f(x);\, x \in X\}$ is contained in $f(X)$. The MuPAD domain `Dom::Interval` facilitates this kind of arithmetic. The routine `numeric::realroots` computes `F:=f(Dom::Interval([a.i,b.i]))` for various subintervals $[a_i, b_i]$ of $[a, b]$. If F does not contain zero, then this subinterval is eliminated from the search interval. Otherwise the subinterval is returned as a candidate for containing zeros of $f(x)$. However, one cannot conclude that F does indeed contain at least one zero, since F is larger than the true image set $\{f(x);\, x \in [a_i, b_i]\}$.
- For polynomials $f(x)$ the routine `polylib::realroots` is called. Its results are intersected with the search interval $[a, b]$. No interval arithmetic is used.

Changes:

- `numeric::realroots` used to be `numeric::fsolve`.
- The search interval is now specified by `x=a..b`. Merging of intervals used to be the default. It now must be requested by the option *`Merge`*.
- The option `eps` for controlling the precision goal was introduced. The new option *`Merge`* controls merging of the return intervals.

`numeric::singularvalues` – numerical singular values of a matrix

`numeric::singularvalues(A)` returns numerical singular values of the matrix `A`.

Call(s):

- `numeric::singularvalues(A)`

Parameters:

`A` — a numerical matrix of domain type `DOM_ARRAY` or of category `Cat::Matrix`.

Return Value: an ordered list of real floating point values

Side Effects: The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Related Functions: `linalg::eigenvalues`, `linalg::eigenvectors`, `numeric::eigenvalues`, `numeric::eigenvectors`, `numeric::singularvectors`, `numeric::spectralradius`

Details:

- The singular values of an $m \times n$ matrix A are the $p = \min(m, n)$ real non-negative square roots of the eigenvalues of $A^H A$ (for $p = n$) or of AA^H (for $p = m$). The Hermitean transpose A^H is the complex conjugate of the transpose of A.
- `numeric::singularvalues` returns a list of real singular values $[d_1, \ldots, d_p]$ sorted by `numeric::sort`, i.e., $0.0 \leq d_1 \leq \cdots \leq d_p$.
- All entries of `A` must be numerical. Numerical expressions such as `exp(PI)`, `sqrt(2)` etc. are accepted and converted to floats. Non-numerical symbolic entries lead to an error.
- `Cat::Matrix` objects, i.e., matrices `A` of a matrix domain such as `Dom::Matrix(..)` or `Dom::SquareMatrix(..)`, are internally converted to arrays over expressions via `A::dom::expr(A)`.
- Singular values are approximated with an *absolute* precision of $10^{-DIGITS}\, r$, where r is the spectral radius of `A` (i.e., r is the absolute value of the largest eigenvalue). Consequently, large singular values should be computed correctly to `DIGITS` decimal places. The numerical approximations of the small singular values are less accurate.

- Singular values may also be computed via

  `map(numeric::eigenvalues(`AA^H`), sqrt);`

  or

  `map(numeric::eigenvalues(`$A^H A$`), sqrt);`

  respectively. The use of `numeric::singularvalues` avoids the costs of the matrix multiplication. Further, the eigenvalue routine requires about twice as many `DIGITS` to compute small singular values with the same precision as `numeric::singularvalues`. Cf. example 2.

Example 1. The singular values of A and A^H coincide:

```
>> A := array(1..3, 1..2, [[1, 2*I], [2, 3],[3, PI]]):
>> numeric::singularvalues(A)

                  [1.503668692, 5.882906158]
```

The Hermitean transpose $B = A^H$:

```
>> B := array(1..2, 1..3,[[1, 2, 3], [-2*I, 3, PI]]):
>> numeric::singularvalues(B)

                  [1.503668692, 5.882906158]

>> delete A, B:
```

Example 2. We use `numeric::eigenvalues` to compute singular values:

```
>> M := Dom::Matrix():
>> A := M([[1, 2*I], [PI, 312689/49766*I], [2, 4*I]]):
```

The Hermitean transpose $B = A^H$ can be computed by the methods `conjugate` and `transpose` of the matrix domain:

```
>> B := M::conjugate(M::transpose(A)):
```

Note that $A^H A$ is positive semi-definite and cannot have negative eigenvalues. However, computing small eigenvalues is numerically ill-conditioned and a small negative value occurs due to round-off:

```
>> numeric::eigenvalues(B*A)

                [-8.67361738e-19, 74.34802201]
```

Consequently, an illegal imaginary singular value is computed:

```
>> map(%, sqrt)

            [0.0000000009313225746 I, 8.622529908]
```

We have to increase `DIGITS` in order to compute this value more accurately:

```
>> DIGITS := 20: map(numeric::eigenvalues(B*A), sqrt)

    [0.00000000001511543358541514159 2, 8.62252990752593714 93]
```

With `numeric::singularvalues` the standard precision suffices:

```
>> DIGITS := 10: numeric::singularvalues(A)

                [1.511542232e-11, 8.622529908]

>> delete M, A, B:
```

Background:

- The function implements standard numerical algorithms from the Handbook of Automatic Computation by Wilkinson and Reinsch.

Changes:

- Internal conversion of `Cat::Matrix` objects now uses the method `"expr"` of the matrix domain.
- The singular values are now sorted by `numeric::sort` from small values to large values.

`numeric::singularvectors` – numerical singular value decomposition of a matrix

`numeric::singularvectors(A, ..)` returns numerical singular values and singular vectors of the matrix `A`.

Call(s):

- `numeric::singularvectors(A <, NoLeftVectors> <, NoRightVectors> <, NoErrors>)`

Parameters:

`A` — a numerical matrix of domain type `DOM_ARRAY` or of category `Cat::Matrix`

Options:

`NoLeftVectors`	—	suppresses the computation of left singular vectors
`NoRightVectors`	—	suppresses the computation of right singular vectors
`NoErrors`	—	suppresses the computation of error estimates

Return Value: a list `[U,d,V,resU,resV]`. `U` is a unitary square float matrix of domain type `DOM_ARRAY`, whose columns are left singular vectors. `d` is a list of singular float values. `V` is a unitary square float matrix of domain type `DOM_ARRAY`, whose columns are right singular vectors. The lists of float residues `resU` and `resV` provide error estimates for the numerical data.

Side Effects: The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Related Functions: `linalg::eigenvalues`, `linalg::eigenvectors`, `numeric::eigenvalues`, `numeric::eigenvectors`, `numeric::singularvalues`, `numeric::spectralradius`

Details:

- All entries of `A` must be numerical. Numerical expressions such as `exp(PI)`, `sqrt(2)` etc. are accepted and converted to floats. Non-numerical symbolic entries lead to an error.

- `Cat::Matrix` objects, i.e., matrices `A` of a matrix domain such as `Dom::Matrix(..)` or `Dom::SquareMatrix(..)` are internally converted to arrays over expressions via `A::dom::expr(A)`.

- The list `[U,d,V,resU,resV]` returned by `numeric::singularvectors` corresponds to the singular data of an $m \times n$ matrix A as described below.

- Let V^H denote the Hermitean transpose of the matrix V, i.e., the complex conjugate of the transpose. The singular value decomposition of an $m \times n$ matrix A is a factorization $A = UDV^H$.

- D is a sparse $m \times n$ matrix with real nonnegative "diagonal" entries $D_{ii} = d_i$, $i = 1, \ldots, p$, $p = \min(m, n)$:

$$D = \begin{pmatrix} d_1 & & 0 \\ & \ddots & \\ & & d_p \\ & & \\ 0 & & 0 \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} d_1 & & & 0 \\ & \ddots & & \\ 0 & & d_p & 0 \end{pmatrix} .$$

  The list `d` $= [d_1, \ldots, d_p]$ returned by `numeric::singularvectors` are the "singular values" of A. They are sorted by `numeric::sort`, i.e., $0.0 \le d_1 \le \cdots \le d_p$.

- U is a unitary $m \times m$ matrix. Its i-th column is an eigenvector of AA^H associated with the eigenvalue d_i^2 ($d_i = 0$ for $i > p$). These are the "left singular vectors" of A. They are returned by `numeric::singularvectors` as an array of floating point numbers.

- V is a unitary $n \times n$ matrix. Its i-th column is an eigenvector of $A^H A$ associated with the eigenvalue d_i^2 ($d_i = 0$ for $i > p$). These are the "right singular vectors" of A. They are returned by `numeric::singularvectors` as an array of floating point numbers. It is normalized such that its diagonal entries are real and nonnegative.

- `resU=[resU[1],..,resU[m]]` is a list of float residues associated with the left singular vectors:

$$\texttt{resU[i]} = \langle A^H u_i, A^H u_i \rangle - d_i^2 \, , \quad i = 1, \ldots, m.$$

Here u_i is the (normalized) i-th column of `U`, $\langle .,. \rangle$ is the usual complex Euclidean scalar product and $d_i = 0$ for $p < i \leq m$.

- `resV=[resV[1],..,resV[n]]` is a list of float residues associated with the right singular vectors:

$$\texttt{resV[i]} = \langle Av_i, Av_i \rangle - d_i^2 \ , \quad i = 1, \ldots, m.$$

  Here v_i is the (normalized) i-th column of `V`, $d_i = 0$ for $p < i \leq n$.

- The residues `resU,resV` vanish for exact singular data U, d, V. Their size indicate the quality of the numerical data `U,d,V`.

- Singular values are approximated with an *absolute* precision of $10^{-DIGITS}\, r$, where r is the spectral radius of `A` (i.e., r is the absolute value of the largest eigenvalue). Consequently, large singular values should be computed correctly to `DIGITS` decimal places. The numerical approximations of the small singular values are less accurate.

- The singular values computed by `numeric::singularvectors` are identical to those computed by `numeric::singularvalues`.

- Singular data may also be computed via

$$[\texttt{d2},\ \texttt{U},\ \texttt{errU}] := \texttt{numeric::eigenvectors}(AA^H);$$

  or

$$[\texttt{d2},\ \texttt{V},\ \texttt{errV}] := \texttt{numeric::eigenvectors}(A^HA);$$

  respectively. The list `d2` is related to the singular values by

$$\texttt{d2} = [0, \ldots, 0, d_1^2, d_2^2, \ldots, d_p^2] \ .$$

  The use of `numeric::singularvectors` avoids the costs of the matrix multiplication. Further, the eigenvector routine requires about twice as many `DIGITS` to compute the data associated with small singular values with the same precision as `numeric::singularvectors`. Also note that the normalization of `U` and `V` may be different.

Option <*NoLeftVectors*>:

- If only right singular vectors are required, then this option may be used to suppress the computation of `U` and the corresponding residues `resU`. The return values for these data are `NIL`.

Option <*NoRightVectors*>:

- If only left singular vectors are required, then this option may be used to suppress the computation of `V` and the corresponding residues `resV`. The return values for these data are `NIL`.

Option <*NoErrors*>:

- If no error estimates are required, then this option may be used to suppress the computation of the residues `resU` and `resV`. The return values for these data are `NIL`.

Example 1. Numerical expressions are converted to floats:

```
>> DIGITS := 5:
>> A :=array(1..3, 1..2, [[1, PI], [2, 3], [3, exp(sqrt(2))]]):
>> [U, d, V, resU, resV] := numeric::singularvectors(A):
```

The singular data are:

```
>> U, d, V
```

```
 +-                               -+
 |  -0.88078, 0.45729,  0.12293   |
 |                                 |
 |   0.14947, 0.51483, -0.84417   |, [0.89905, 6.9986],
 |                                 |
 |   0.44932, 0.72515,  0.5218    |
 +-                               -+

    +-                     -+
    |   0.85215,  0.5233   |
    |                      |
    |  -0.52331, 0.85215   |
    +-                     -+
```

The residues indicate that these results are not severely affected by round-off within the working precision of 5 digits:

```
>> resU, resV
```

```
   [2.0954e-9, 2.9802e-8, 1.7347e-18], [3.4925e-9, 7.4506e-
8]
```

```
>> delete DIGITS, A, U, d, V, resU, resV:
```

Example 2. We demonstrate how to reconstruct a matrix from its singular data:

```
>> DIGITS := 3: A := array(1..2, 1..3, [[1.0, I, PI], [2, 3, I]]):
>> [U, d, V, resU, resV] := numeric::singularvectors(A, NoErrors):
```

For convenience, the matrix domain `Dom::Matrix()` is used to process the matrices:

```
>> M := Dom::Matrix(): U := M(U)
```

```
            +-                                        -+
            |  - 0.789 + 0.336 I, 0.511 + 0.0665 I  |
            |                                         |
            |   0.487 - 0.168 I,    0.84 + 0.17 I   |
            +-                                        -+
```

A "diagonal" matrix is built from the singular values:

```
>> DD := M(2, 3, d, Diagonal)
```

```
                       +-              -+
                       |  3.27,  0,  0  |
                       |                |
                       |    0,  3.9, 0  |
                       +-              -+
```

```
>> V := M(V)
```

```
   +-                                                    -
+
   |       0.0568,      0.562 + 0.104 I, - 0.681 - 0.454 I  |
   |                                                        |
   |   0.55 + 0.0871 I,      0.663,       0.454 + 0.208 I   |
   |                                                        |
   |  - 0.81 + 0.174 I, 0.455 - 0.162 I,       0.283        |
   +-                                                    -
+
```

We use the methods `conjugate` and `transpose` of the matrix domain to compute the Hermitean transpose of `V` and reconstruct `A` up to numerical round-off:

```
>> VH := M::conjugate(M::transpose(V)):
>> U*DD*VH
```

```
   +-                                                     -
+
   |  1.0 + 2.62e-10 I,       1.0 I,      3.14 + 4.66e-10 I  |
   |                                                         |
   |  2.0 + 5.02e-10 I, 3.0 - 1.16e-10 I, - 3.73e-9 + 1.0 I  |
   +-                                                     -
+
```

```
>> delete DIGITS, A, U, d, V, resU, resV, M, DD, VH:
```

Changes:

- Conversion of `Cat::Matrix` objects now uses the method `"expr"` of the matrix domain.
- The singular values are now sorted by `numeric::sort` from small values to large values. Matrices of singular vectors are now returned as arrays.

`numeric::sort` – sort a numerical list

`numeric::sort(list)` sorts the elements in `list`.

Call(s):

- `numeric::sort(list)`

Parameters:

`list` — a list of numbers or numerical expressions

Return Value: a sorted list

Side Effects: The function is sensitive to the environment variable `DIGITS`.

Related Functions: `sort`

Details:

- The elements of the list are converted to floating point numbers via `float`. Elements that cannot be converted lead to an error.
- The floating point numbers are sorted from small real part to large real part. Elements with the same real part are sorted from small absolute value to large absolute value. In case of a tie (i.e., two elements form a complex conjugate pair) the element with positive imaginary part comes first.
- This function is used to sort the return values of `numeric::eigenvalues`, `numeric::eigenvectors`, `numeric::polyroots`, `numeric::singularvalues` and `numeric::singularvectors`.

Example 1.

```
>> numeric::sort([1, 2.0, I, -3, -I, PI, sqrt(2)])

   [-3.0, 1.0 I, - 1.0 I, 1.0, 1.414213562, 2.0, 3.141592654]
```

In the following example the sorting criterion does not seem to be satisfied:

```
>> x := sin(PI/3):
>> L := numeric::sort([x, sin(float(PI/3)) - I, x + I])

   [0.8660254038 - 1.0 I, 0.8660254038, 0.8660254038 + 1.0 I]
```

This is explained by the fact that the floating point numbers internally have a more accurate representation than shown on the screen. The real part of the first element is indeed a little bit smaller than the other real parts:

```
>> DIGITS := 20: L

 [0.86602540378443864668 - 1.0 I, 0.86602540378443864673,

    0.86602540378443864673 + 1.0 I]

>> delete x, L, DIGITS:
```

Changes:

- `numeric::sort` is a new function.

`numeric::solve` – numerical solution of equations (the float attribute of solve)

`numeric::solve` computes numerical solutions of equations.

Call(s):

- `numeric::solve(eqs, <, vars> <, Options>)`
  `float(hold(solve)(eqs, <, vars> <, Options>))`
  `float(freeze(solve)(eqs, <, vars> <, Options>))`

Parameters:

`eqs` — an equation, a list of equations, or a set of equations. Also arithmetical expressions are accepted and interpreted as homogeneous equations.

Options:

`vars`	— an unknown, a list of unknowns or a set of unknowns. Unknowns may be identifiers or indexed identifiers. Also equations of the form `x=a` or `x=a..b` are accepted wherever an unknown `x` is expected. This way starting points and search ranges are specified for the numerical search. They must be numerical, infinite search ranges are accepted.
`Options`	— a combination of *`Multiple`*, *`FixedPrecision`*, *`SquareFree`*, *`Factor`*, *`RestrictedSearch`*, or *`Random`*
`Multiple`	— only to be used if `eqs` is a polynomial equation or a system of polynomial equations. With this option, information on the multiplicity of degenerate polynomial roots is returned.
`FixedPrecision`	— only to be used if `eqs` is a single univariate polynomial. It launches a quick numerical search with fixed internal precision.
`SquareFree`	— only to be used if `eqs` is a single univariate polynomial. Symbolic square free factorization is applied, before the numerical search starts.
`Factor`	— only to be used if `eqs` is a single univariate polynomial. Symbolic factorization is applied, before the numerical search starts.
`RestrictedSearch`	— only to be used for non-polynomial equations. The numerical search is restricted to the search ranges specified in `vars`.
`Random`	— only to be used for non-polynomial equations. With this option, several calls to `numeric::solve` may lead to different solutions of the equation(s).

Return Value: a set of numerical solutions. With the option *`Multiple`*, a set of domain type `Dom::Multiset` is returned.

Side Effects: The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Related Functions: `linsolve`, `numeric::fsolve`, `numeric::linsolve`, `numeric::polyroots`, `numeric::polysysroots`, `numeric::realroot`, `numeric::realroots`, `polylib::realroots`, `solve`

Details:

- The call `numeric::solve(arguments)` is equivalent to calling the float attribute of `solve` by `float(hold(solve)(arguments))` or alternatively by `float(freeze(solve)(arguments))`
- `numeric::solve` is a simple interface function unifying the functionality of the numerical solvers `numeric::fsolve`, `numeric::polyroots` and `numeric::polysysroots`. The return format of these routines is changed to make it consistent with the return values of the symbolic solver `solve`.
- You may call the specialized numerical solvers directly. However, note the return types specific to each of these solvers.
- `numeric::solve` classifies the equations as follows:
  a) If `eqs` is a single univariate polynomial equation, then it is directly passed to `numeric::polyroots`. Cf. example 2. The roots are returned as a set or as a `Dom::Multiset`, if *`Multiple`* is used.
  b) If `eqs` is a multivariate polynomial equation or a list or set containing such an equation, then the equations and the appropriate optional arguments are passed to `numeric::polysysroots`. Cf. example 3. The roots are returned as a set or as a `Dom::Multiset`, if *`Multiple`* is used.
  c) If `eqs` is a non-polynomial equation or a set or list containing such an equation, then then the equations and the appropriate optional arguments are passed to the numerical solver `numeric::fsolve`. Note that for non-polynomial equations there must not be more equations than unknowns!

     Using *`Multiple`* for non-polynomial equations leads to an error! A single numerical root is returned. Cf. example 4.
- For convenience, also polynomials of domain type `DOM_POLY` are accepted, wherever an equation is expected.
- In contrast to the symbolic solver `solve`, the numerical solver does not react to properties of identifiers set via `assume`.

Option `<vars>`:

- If the user does not specify indeterminates to be solved for, then the indeterminates are internally chosen by `numeric::indets(eqs)`.
- Starting points such as `x=a` or search ranges such as `x=a..b` specified in `vars` are ignored, if `eqs` is a polynomial equation or a system of polynomial equations.

Option <*Multiple*>:

- This option may only be used when solving polynomial equations! It changes the return type from `DOM_SET` to `Dom::Multiset`.

Option <*FixedPrecision*>:

- This option only has an effect if `eqs` is a single univariate polynomial equation. It is passed to `numeric::polyroots`, which uses a numerical search with fixed internal precision. This is fast, but degenerate roots may be returned with a restricted precision. See the help page of `numeric::polyroots` for details.

Option <*SquareFree*>:

- This option only has an effect if `eqs` is a single univariate polynomial equation. It is passed to `numeric::polyroots`, which preprocesses the polynomial by a symbolic square free factorization. See the help page of `numeric::polyroots` for details.

Option <*Factor*>:

- This option only has an effect if `eqs` is a single univariate polynomial equation. It is passed to `numeric::polyroots`, which preprocesses the polynomial by a symbolic factorization. See the help page of `numeric::polyroots` for details.

Option <*RestrictedSearch*>:

- This option only has an effect if `eqs` contains a non-polynomial equation. It is passed to `numeric::fsolve`, which restricts the search to the search range specified in `vars`. See the help page of `numeric::fsolve` for details.

- This option should only be used in conjunction with search ranges.

Option <*Random*>:

- This option only has an effect if `eqs` contains a non-polynomial equation. It is passed to `numeric::fsolve` which switches to a random search strategy. See the help page of `numeric::fsolve` for details.

Example 1. The following three calls are equivalent:

```
>> eqs := {x^2 = sin(y), y^2 = cos(x)}:
>> numeric::solve(eqs, {x, y}),
   float(hold(solve)(eqs, {x, y})),
   float(freeze(solve)(eqs, {x,y}))

 {[y = 0.8116062152, x = 0.8517004887]},

    {[y = 0.8116062152, x = 0.8517004887]},

    {[y = 0.8116062152, x = 0.8517004887]}

>> delete eqs:
```

Example 2. We demonstrate the root search for univariate polynomials:

```
>> numeric::solve(x^6 - PI*x^2 = sin(3), x)

 {-1.339589767, 1.339589767, - 1.322706295 I, - 0.2120113223 I,

    0.2120113223 I, 1.322706295 I}
```

Polynomials of type `DOM_POLY` can be used as input:

```
>> numeric::solve(poly((x - 1/3)^3, [x]), x)

                          {0.3333333333}
```

With *`Multiple`*, a `Dom::Multiset` is returned, indicating the multiplicity of the root:

```
>> numeric::solve(x^3 - x^2 + x/3 -1/27, x, Multiple)

                        {[0.3333333333, 3]}
```

Example 3. We demonstrate the root search for polynomial systems. Note that the symbolic solver `solve` is involved. Symbolic parameters are accepted:

```
>> numeric::solve({x^2 + y^2 = 1, x^2 - y^2 = exp(z)}, {x, y})

                                          1/2
 {[y = - 0.7071067812 (1.0 - 1.0 exp(z))   ,

                                       1/2
    x = - 0.7071067812 (exp(z) + 1.0)   ],

    ...
                                          1/2
    [y = 0.7071067812 (1.0 - 1.0 exp(z))   ,

                                      1/2
    x = 0.7071067812 (exp(z) + 1.0)   ]}
```

Example 4. We demonstrate the root search for non-polynomial equations.

```
>> eq := exp(-x) - 10*x^2:
>> numeric::solve(eq, x)

                          {0.2755302947}
```

Since `numeric::solve` just calls the root finder `numeric::fsolve`, one may also use this routine directly. Note the different output format:

```
>> numeric::fsolve(eq, x)

                        [x = 0.2755302947]
```

The input syntax of `numeric::solve` and `numeric::fsolve` are identical, i.e., starting points, search ranges and options may be used. E.g., another solution of the previous equation is found by a restricted search over the interval $[-1, 0]$:

```
>> numeric::solve(eq, x = -1..0, RestrictedSearch)

                          {-0.3829657727}
```

The following search for a solution in the entire 2-dimensional plane fails:

```
>> eqs := [exp(x) = 2*y^2, sin(y) = y*x^3]:
>> numeric::solve(eqs, [x, y])

                               {[]}
```

Assisted by starting points for the internal search a solution is found:

```
>> numeric::solve(eqs, [x = 1, y = 1.5])

              {[x = 0.9290711315, y = 1.125201325]}
```

Another solution with negative y is found with an appropriate search range:

```
>> numeric::solve(eqs, [x = 1, y = -infinity..0])

             {[x = 0.9290711314, y = -1.125201325]}

>> delete eq, eqs:
```

Changes:

- `numeric::solve` is a new function.

`numeric::spectralradius` – the spectral radius of a matrix

`numeric::spectralradius(A, ..)` returns the eigenvalue of the matrix `A` that has the largest absolute value.

Call(s):

- `numeric::spectralradius(A, x0, n)`

Parameters:

- `A` — an $m \times m$ matrix of domain type `DOM_ARRAY` or of category `Cat::Matrix`
- `x0` — a starting vector: a 1-dimensional array or a list of length m
- `n` — the maximal number of iterations: a positive integer

Return Value: A list `[lambda, x, residue]` is returned. `lambda` is a floating point approximation of the the eigenvalue of largest absolute value. The 1-dimensional array `x` is a numerical eigenvector corresponding to `lambda`. `residue` is a floating point number indicating the numerical quality of `lambda` and `x`.

Side Effects: The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Related Functions: `linalg::eigenvalues`, `linalg::eigenvectors`, `numeric::eigenvalues`, `numeric::eigenvectors`, `numeric::singularvalues`, `numeric::singularvectors`

Details:

- The spectral radius of a matrix with eigenvalues λ_i is $\max|\lambda_i|$.
- The return value `lambda` is an approximation of the corresponding eigenvalue: `abs(lambda)` is the spectral radius.
- The return value `x` is the corresponding normalized eigenvector: $\|x\|_2 = 1$.
- The return value `residue` $= \|$`A x` $-$ `lambda x`$\|_2$ provides an error estimate for the eigenvalue. For Hermitean matrices this is a rigorous upper bound for the error $|$`lambda` $- \lambda_{exact}|$, where λ_{exact} is the exact eigenvalue.
- `numeric::spectralradius` implements the power method to compute the eigenvalue and the associated eigenvector defining the spectral radius: the vector iteration $x_i = (A^i x_0)/\|A^i x_0\|_2$ "converges" towards the eigenspace associated with the spectral radius. The starting vector x_0 is provided by the second argument of `numeric::spectralradius`.
- The iteration does not converge (converges slowly), if the spectral radius is generated by several distinct eigenvalues with the same (similar) absolute value.

- Internally, the iteration stops, when the approximation of the eigenvalue becomes stationary within the relative precision given by `DIGITS`. If this does not happen within n iterations, then a warning is issued and the present values are returned. Cf. example 2.

Example 1. We define a starting vector as a 1-dimensional array and allow a maximum of 1000 internal iterations:

```
>> A := array(1..2, 1..2, [[1, 2], [5, -10]]):
   x0 := array(1..2, [1, 1]):
   numeric::spectralradius(A, x0, 1000)

 --               +-                             -+
 |  -10.84429006, | 0.166500972, -0.9860412904 |,
 --               +-                             -+

                    --
    1.041382012e-11  |
                    --
```

Next we use a list to specify a starting vector:

```
>> A := array(1..2, 1..2, [[I, 3], [3, I]]):
   numeric::spectralradius(A, [1, 1], 1000)
```

```
     --              +-                              -+      -
-
     |  3.0 + 1.0 I, | 0.7071067812, 0.7071067812 |, 0.0  |
     --              +-                              -+      -
-
```

```
>> delete A, x0:
```

Example 2. The following matrix has two distinct eigenvalues 1 and -1 of the same absolute value. The power method must fail.

```
>> A := array(1..2, 1..2, [[1, 0], [0, -1]]):
```

We allow a maximum of 1000 internal steps. The call results in a warning. The large residue also indicates that the power method did not converge:

```
>> numeric::spectralradius(A, [1, 1], 1000)
```

```
 Warning: no convergence of vector iteration [numeric::spectral\
 radius]
```

```
         --      +-                               -+      --
         |  0.0, | 0.7071067812, -0.7071067812 |, 1.0  |
         --      +-                               -+      --
```

```
>> delete A:
```

Changes:

- `numeric::spectralradius` used to be `numeric::vonMises`.

`numeric::sum` – numerical approximation of sums (the float attribute of sum)

`numeric::sum(f[i], i=a..b)` computes a numerical approximation of $\sum_{i=a}^{b} f_i$.

`numeric::sum(f(x), x = RootOf(p(X), X))` computes a numerical approximation of $\sum_{x \in \texttt{RootOf(p(X),X)}} f(x)$.

Call(s):

- `numeric::sum(f[i], i = a..b))`
  `float(hold(sum)(f[i], i = a..b))`
  `float(freeze(sum)(f[i], i = a..b))`
- `numeric::sum(f(x), x = RootOf(p(X), X)))`
  `float(hold(sum)(f(x), x = RootOf(p(X), X)))`
  `float(freeze(sum)(f(x), x = RootOf(p(X), X)))`

Parameters:

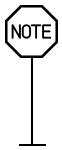
`f[i]` — an arithmetical expression in i
`i` — the summation index: an identifier or an indexed identifier
`a, b` — integers or $\pm$`infinity` satisfying $a \leq b$
`f(x)` — an arithmetical expression in x
`x` — the summation variable: an identifier or an indexed identifier
`p(X)` — a univariate polynomial in X
`X` — the indeterminate of p: an identifier or an indexed identifier

Return Value: a floating point number.

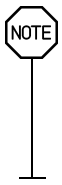
Related Functions: `_plus`, `int`, `numeric::quadrature`, `sum`

Details:

- The call `numeric::sum(..)` is equivalent to calling the float attribute of `sum` via `float(hold(sum)(..))` or `float(freeze(sum)(..))`.
- The summation variable i (respectively, x) must be the only symbolic parameter in f_i (respectively, $f(x)$), otherwise an error occurs! Numerical expressions such as `exp(PI)`, `sqrt(2)` etc. are accepted and converted to floating point numbers.
- For infinite sums, the expression f_i with integer i must have an extension $f(x)$ to all real x in the interval $[a, b]$, i.e., $f(i) = f_i$. Internally, the integral $\int_a^b f(x)\,dx$ is computed numerically and used in the approximation process.



- For finite sums, `numeric::sum` just returns `_plus(float(f[i]$i=a..b)`. Note that numerical cancellation may occur! If `f[i]` does not contain floating point numbers, then cancellation can be avoided summing the symbolic terms by `_plus(f[i]$i=a..b)`. Cf. example 3.



- Convergence is fast, if f_i decays rapidly for $|i| \to \infty$.

- Convergence may be slow for alternating sums containing expressions such as $(-1)^i$. Such sums are also often subject to cancellation problems!

- The call `numeric::sum(f(x), x = RootOf(p(X), X))` computes numerical approximations of all roots of p, substitutes these values into $f(x)$ and adds up the results. Cf. example 4. This process may be subject to cancellation problems!

Example 1. We demonstrate some equivalent calls for numerical summation:

```
>> numeric::sum(1/i!, i = 0..infinity),
   float(hold(sum)(1/i!, i = 0..infinity)),
   float(freeze(sum)(1/i!, i = 0..infinity))

            2.718281829, 2.718281829, 2.718281829
```

MuPAD's symbolic summation does not find a simple representation of the following sum:

```
>> sum(1/i!/(i+1)!, i = 0..infinity)

              /          1                              \
           sum| ------------------, i = 0..infinity |
              \ fact(i) fact(i + 1)                    /
```

The following float evaluation calls `numeric::sum`:

```
>> float(%)

                          1.590636855
```

The exact value of the following sum is $\pi \coth(\pi)$:

```
>> numeric::sum(1/(1+i^2), i = -infinity..infinity) =
   float(PI*coth(PI))

                  3.153348095 = 3.153348095
```

Example 2. The following sum cannot be evaluated numerically because of the symbolic parameter `x`:

```
>> numeric::sum(1/(x+i^2), i = -infinity..infinity)

 Error: first argument may only contain i as symbolic parameter\
  [numeric::sum]
```

Example 3. We demonstrate numerical cancellation when summing the Taylor series for exp(−20):

```
>> exp(-20.0) <> numeric::sum((-20)^i/i!, i = 0..100)

          0.000000002061153622 <> 0.000000002068277833
```

Also the infinite sum suffers from cancellation:

```
>> exp(-20.0) <> numeric::sum((-20)^i/i!, i = 0..infinity)

          0.000000002061153622 <> 0.000000002068334676
```

Cancellation can be avoided using a finite sum with exact terms:

```
>> exp(-20.0) = float(_plus((-20)^i/i! $ i = 0..100))

           0.000000002061153622 = 0.000000002061153622
```

Example 4. The following call computes the numerical roots of the polynomial in the `RootOf` expression and sums over all the roots:

```
>> numeric::sum(exp(x)/x, x = RootOf(X^10 - X - PI, X))

                                9.681693381
```

Background:

- `numeric::sum` makes use of the Euler-MacLaurin formula

$$\sum_{i=a}^{b} f(i) = \frac{f(a)+f(b)}{2} + \int_a^b f(x)\,dx$$

$$+ \sum_{m=1}^{M} \frac{B_{2m}}{(2\,m)!}\left(f^{(2m-1)}(b) - f^{(2m-1)}(a)\right) + \cdots$$

  involving the Bernoulli numbers B_{2m}.

Changes:

- `numeric::sum` used to be `funcattr(sum, "float")`.
- Lower bounds $a =$ `-infinity` are now accepted.
- Summation over `RootOf` expressions was introduced.