16.43 NUMERIC: Solving numerical problems

This package implements basic algorithms of numerical analysis. These include:

- solution of algebraic equations by Newton's method

```
num_solve({sin x=cos y, x + y = 1},{x=1,y=2})
```

- solution of ordinary differential equations

```
num_odesolve(df(y,x)=y,y=1,x=(0 .. 1), iterations=5)
```

- bounds of a function over an interval

```
bounds(sin x+x,x=(1 .. 2));
```

- minimizing a function (Fletcher Reeves steepest descent)

```
num_min(sin(x)+x/5, x);
```

- Chebyshev curve fitting

```
chebyshev_fit(sin x/x,x=(1 .. 3),5);
```

- numerical quadrature

```
num_int(sin x,x=(0 .. pi));
```

Author: Herbert Melenk.

The NUMERIC package implements some numerical (approximative) algorithms for REDUCE, based on the REDUCE rounded mode arithmetic. These algorithms are implemented for standard cases. They should not be called for ill-conditioned problems; please use standard mathematical libraries for these.

16.43.1 Syntax

16.43.1.1 Intervals, Starting Points

Intervals are generally coded as lower bound and upper bound connected by the operator `'..'`, usually associated to a variable in an equation. E.g.

```
x= (2.5 .. 3.5)
```

means that the variable x is taken in the range from 2.5 up to 3.5. Note, that the bounds can be algebraic expressions, which, however, must evaluate to numeric results. In cases where an interval is returned as the result, the lower and upper bounds can be extracted by the `PART` operator as the first and second part respectively. A starting point is specified by an equation with a numeric righthand side, e.g.

```
x=3.0
```

If for multivariate applications several coordinates must be specified by intervals or as a starting point, these specifications can be collected in one parameter (which is then a list) or they can be given as separate parameters alternatively. The list form is more appropriate when the parameters are built from other REDUCE calculations in an automatic style, while the flat form is more convenient for direct interactive input.

16.43.1.2 Accuracy Control

The keyword parameters $accuracy = a$ and $iterations = i$, where a and i must be positive integer numbers, control the iterative algorithms: the iteration is continued until the local error is below 10^{-a}; if that is impossible within i steps, the iteration is terminated with an error message. The values reached so far are then returned as the result.

16.43.1.3 tracing

Normally the algorithms produce only a minimum of printed output during their operation. In cases of an unsuccessful or unexpected long operation a trace of the iteration can be printed by setting

```
on trnumeric;
```

16.43.2 Minima

The Fletcher Reeves version of the $steepest\ descent$ algorithms is used to find the minimum of a function of one or more variables. The function must have continuous partial derivatives with respect to all variables. The starting point of the search can be specified; if not, random values are taken instead. The steepest descent algorithms in general find only local minima.

Syntax:

NUM_MIN $(exp, var_1[= val_1][, var_2[= val_2] \ldots]$

$[, accuracy = a][, iterations = i])$

or

NUM_MIN $(exp, \{var_1[= val_1][, var_2[= val_2] \ldots]\}$

$[, accuracy = a][, iterations = i])$

where exp is a function expression,

$var_1, var_2, \ldots$ are the variables in exp and $val_1, val_2, \ldots$ are the (optional) start values.

NUM_MIN tries to find the next local minimum along the descending path starting at the given point. The result is a list with the minimum function value as first element followed by a list of equations, where the variables are equated to the coordinates of the result point.

Examples:

```
num_min(sin(x)+x/5, x);

{4.9489585606,{X=29.643767785}}

num_min(sin(x)+x/5, x=0);

{ - 1.3342267466,{X= - 1.7721582671}}

% Rosenbrock function (well known as hard to minimize).
fktn := 100*(x1**2-x2)**2 + (1-x1)**2;
num_min(fktn, x1=-1.2, x2=1, iterations=200);

{0.00000021870228295,{X1=0.99953284494,X2=0.99906807238}}
```

16.43.3 Roots of Functions/ Solutions of Equations

An adaptively damped Newton iteration is used to find an approximative zero of a function, a function vector or the solution of an equation or an equation system. Equations are internally converted to a difference of lhs and rhs such that the Newton method (=zero detection) can be applied. The expressions must have continuous derivatives for all variables. A starting point for the iteration can be given. If not given, random values are taken instead. If the number of forms is not equal to the number of variables, the Newton method cannot be applied. Then the minimum of the sum of absolute squares is located instead.

With ON COMPLEX solutions with imaginary parts can be found, if either the expression(s) or the starting point contain a nonzero imaginary part.

Syntax:

NUM_SOLVE $(exp_1, var_1[= val_1][, accuracy = a][, iterations = i])$

or

NUM_SOLVE $(\{exp_1, \ldots, exp_n\}, var_1[= val_1], \ldots, var_n[= val_n]$

$[, accuracy = a][, iterations = i])$

or

NUM_SOLVE $(\{exp_1, \ldots, exp_n\}, \{var_1[= val_1], \ldots, var_n[= val_n]\}$

$[, accuracy = a][, iterations = i])$

where $exp_1, \ldots, exp_n$ are function expressions,

$var_1, \ldots, var_n$ are the variables,

$val_1, \ldots, val_n$ are optional start values.

NUM_SOLVE tries to find a zero/solution of the expression(s). Result is a list of equations, where the variables are equated to the coordinates of the result point.

The Jacobian matrix is stored as a side effect in the shared variable JACOBIAN.

Example:

```
num_solve({sin x=cos y, x + y = 1},{x=1,y=2});

{X= - 1.8561957251,Y=2.856195584}

jacobian;

[COS(X)  SIN(Y)]
[               ]
[  1       1    ]
```

16.43.4 Integrals

For the numerical evaluation of univariate integrals over a finite interval the following strategy is used:

1. If the function has an antiderivative in close form which is bounded in the integration interval, this is used.

2. Otherwise a Chebyshev approximation is computed, starting with order 20, eventually up to order 80. If that is recognized as sufficiently convergent it is used for computing the integral by directly integrating the coefficient sequence.

3. If none of these methods is successful, an adaptive multilevel quadrature algorithm is used.

For multivariate integrals only the adaptive quadrature is used. This algorithm tolerates isolated singularities. The value $iterations$ here limits the number of local interval intersection levels. $Accuracy$ is a measure for the relative total discretization error (comparison of order 1 and order 2 approximations).

Syntax:

NUM_INT $(exp, var_1 = (l_1..u_1)[, var_2 = (l_2..u_2)\ldots]$

$[, accuracy = a][, iterations = i])$

where exp is the function to be integrated,

$var_1, var_2, \ldots$ are the integration variables,

$l_1, l_2, \ldots$ are the lower bounds,

$u_1, u_2, \ldots$ are the upper bounds.

Result is the value of the integral.

Example:

```
num_int(sin x,x=(0 .. pi));
```

```
2.0000010334
```

16.43.5 Ordinary Differential Equations

A Runge-Kutta method of order 3 finds an approximate graph for the solution of a ordinary differential equation real initial value problem.

Syntax:

NUM_ODESOLVE $(exp$,$depvar = dv$,$indepvar$=$(from..to)$

$[, accuracy = a][, iterations = i])$

where

exp is the differential expression/equation,

$depvar$ is an identifier representing the dependent variable (function to be found),

$indepvar$ is an identifier representing the independent variable,

exp is an equation (or an expression implicitly set to zero) which contains the first derivative of $depvar$ wrt $indepvar$,

$from$ is the starting point of integration,

to is the endpoint of integration (allowed to be below $from$),

dv is the initial value of $depvar$ in the point $indepvar = from$.

The ODE exp is converted into an explicit form, which then is used for a Runge Kutta iteration over the given range. The number of steps is controlled by the value of i (default: 20). If the steps are too coarse to reach the desired accuracy in the neighborhood of the starting point, the number is increased automatically.

Result is a list of pairs, each representing a point of the approximate solution of the ODE problem.

Example:

```
num_odesolve(df(y,x)=y,y=1,x=(0 .. 1), iterations=5);

{{0.0,1.0},{0.2,1.2214},{0.4,1.49181796},{0.6,1.8221064563},

 {0.8,2.2255208258},{1.0,2.7182511366}}
```

Remarks:

– If in exp the differential is not isolated on the lefthand side, please ensure that the dependent variable is explicitly declared using a `DEPEND` statement, e.g.

```
depend y,x;
```

otherwise the formal derivative will be computed to zero by REDUCE.

– The REDUCE package SOLVE is used to convert the form into an explicit ODE. If that process fails or has no unique result, the evaluation is stopped with an error message.

16.43.6 Bounds of a Function

Upper and lower bounds of a real valued function over an interval or a rectangular multivariate domain are computed by the operator BOUNDS. The algorithmic basis is the computation with inequalities: starting from the interval(s) of the variables, the bounds are propagated in the expression using the rules for inequality computation. Some knowledge about the behavior of special functions like ABS, SIN, COS, EXP, LOG, fractional exponentials etc. is integrated and can be evaluated if the operator BOUNDS is called with rounded mode on (otherwise only algebraic evaluation rules are available).

If BOUNDS finds a singularity within an interval, the evaluation is stopped with an error message indicating the problem part of the expression.

Syntax:

BOUNDS $(exp, var_1 = (l_1..u_1)[, var_2 = (l_2..u_2)\ldots])$

BOUNDS $(exp, \{var_1 = (l_1..u_1)[, var_2 = (l_2..u_2)\ldots]\})$

where exp is the function to be investigated,

$var_1, var_2, \ldots$ are the variables of exp,

$l_1, l_2, \ldots$ and $u_1, u_2, \ldots$ specify the area (intervals).

$BOUNDS$ computes upper and lower bounds for the expression in the given area. An interval is returned.

Example:

```
bounds(sin x,x=(1 .. 2));

{-1,1}

on rounded;
bounds(sin x,x=(1 .. 2));

0.84147098481 .. 1

bounds(x**2+x,x=(-0.5 .. 0.5));

 - 0.25 .. 0.75
```

16.43.7 Chebyshev Curve Fitting

The operator family $Chebyshev\_\ldots$ implements approximation and evaluation of functions by the Chebyshev method. Let $T_n^{(a,b)}(x)$ be the Chebyshev polynomial of order n transformed to the interval (a, b). Then a function $f(x)$ can be approximated in (a, b) by a series

$f(x) \approx \sum_{i=0}^{N} c_i T_i^{(a,b)}(x)$

The operator $Chebyshev\_fit$ computes this approximation and returns a list, which has as first element the sum expressed as a polynomial and as second element the sequence of Chebyshev coefficients c_i. $Chebyshev\_df$ and $Chebyshev\_int$ transform a Chebyshev coefficient list into the coefficients of the corresponding derivative or integral respectively. For evaluating a Chebyshev approximation at a given point in the basic interval the operator $Chebyshev\_eval$ can be used. Note that $Chebyshev\_eval$ is based on a recurrence relation which is in general more stable than a direct evaluation of the complete polynomial.

CHEBYSHEV_FIT $(fcn, var = (lo..hi), n)$

CHEBYSHEV_EVAL $(coeffs, var = (lo..hi), var = pt)$

CHEBYSHEV_DF $(coeffs, var = (lo..hi))$

CHEBYSHEV_INT $(coeffs, var = (lo..hi))$

where fcn is an algebraic expression (the function to be fitted), var is the variable of fcn, lo and hi are numerical real values which describe an interval ($lo < hi$), n is the approximation order,an integer > 0, set to 20 if missing, pt is a numerical value in the interval and $coeffs$ is a series of Chebyshev coefficients, computed by one of $CHEBYSHEV\_COEFF$, $\_DF$ or $\_INT$.

Example:

```
on rounded;

w:=chebyshev_fit(sin x/x,x=(1 .. 3),5);

                3           2
w := {0.03824*x  - 0.2398*x  + 0.06514*x + 0.9778,

      {0.8991,-0.4066,-0.005198,0.009464,-0.00009511}}

chebyshev_eval(second w, x=(1 .. 3), x=2.1);
```

```
0.4111
```

16.43.8 General Curve Fitting

The operator NUM_FIT finds for a set of points the linear combination of a given set of functions (function basis) which approximates the points best under the objective of the least squares criterion (minimum of the sum of the squares of the deviation). The solution is found as zero of the gradient vector of the sum of squared errors.

Syntax:

NUM_FIT $(vals, basis, var = pts)$

where $vals$ is a list of numeric values,

var is a variable used for the approximation,

pts is a list of coordinate values which correspond to var,

$basis$ is a set of functions varying in var which is used for the approximation.

The result is a list containing as first element the function which approximates the given values, and as second element a list of coefficients which were used to build this function from the basis.

Example:

```
 % approximate a set of factorials by a polynomial
pts:=for i:=1 step 1 until 5 collect i$
vals:=for i:=1 step 1 until 5 collect
        for j:=1:i product j$

num_fit(vals,{1,x,x**2},x=pts);

                2
{14.571428571*X  - 61.428571429*X + 54.6,{54.6,

     - 61.428571429,14.571428571}}

num_fit(vals,{1,x,x**2,x**3,x**4},x=pts);
```

```
                4                   3
{2.2083333234*X  - 20.249999879*X

                   2
  + 67.791666154*X  - 93.749999133*X

  + 44.999999525,

 {44.999999525, - 93.749999133,67.791666154,

   - 20.249999879,2.2083333234}}
```

16.43.9 Function Bases

The following procedures compute sets of functions e.g. to be used for approximation. All procedures have two parameters, the expression to be used as $variable$ (an identifier in most cases) and the order of the desired system. The functions are not scaled to a specific interval, but the $variable$ can be accompanied by a scale factor and/or a translation in order to map the generic interval of orthogonality to another (e.g. $(x - 1/2) * 2pi$). The result is a function list with ascending order, such that the first element is the function of order zero and (for the polynomial systems) the function of order n is the $n + 1$-th element.

```
monomial_base(x,n)        {1,x,...,x**n}
trigonometric_base(x,n)   {1,sin x,cos x,sin(2x),cos(2x)...}
Bernstein_base(x,n)       Bernstein polynomials
Legendre_base(x,n)        Legendre polynomials
Laguerre_base(x,n)        Laguerre polynomials
Hermite_base(x,n)         Hermite polynomials
Chebyshev_base_T(x,n)     Chebyshev polynomials first kind
Chebyshev_base_U(x,n)     Chebyshev polynomials second kind
```

Example:

```
Bernstein_base(x,5);

        5       4       3       2
{ - X  + 5*X  - 10*X  + 10*X  - 5*X + 1,
```

```
       4       3       2
5*X*(X  - 4*X  + 6*X  - 4*X + 1),

    2       3       2
10*X *( - X  + 3*X  - 3*X + 1),

    3   2
10*X *(X  - 2*X + 1),

   4
5*X *( - X + 1),

 5
X }
```