

Tusanga calculator help pages

www.tusanga.com

Copyright © 1996-2009 by Tusanga. All rights reserved. Updated 10.08.2009

Getting started

To start using the Tusanga Calculator please enter an expression (like $2+3$) in the big form in the middle of the screen - below the 'Enter your mathematical Expressions here' line. When you press the 'Calculate' button the system evaluates your expression. Please try it out now!

Input:

$2 + 3$

Output:

5

The first line ' $2+3$ ' is your input to the calculator. The second line '5' is the resulting output from the calculator.

The user can assign values to variables. Please notice the use of ':=' when assigning values. This is the equal sign the calculator uses for assignments. An ordinary '=' will not do.

Input:

$a := 2$

Output:

$a := 2$

Once again, the first line ' $a := 2$ ' is the user input and the second line is the output from the system. In this case the system responds, that it has assigned the value 2 to the variable a.

' $a := 2$ ' is an example of a statement. An expression is evaluated to another expression ($2+3$ evaluates to 5) where as a statement is executed (assign the value 2 to the variable a).

Let's try another calculation combining an expression and a statement.

Input:

$b := 3 + 4$

Output:

$b := 7$

The right hand side of a variable assignment statement can include known variables. In this example a semicolon separates the different statements and expressions.

Input:

$a := 2 ; b := 7 ; c := a + b$

Output:

$a := 2$

Output:

$b := 7$

Output:

$c := 9$

The right hand side of a variable assignment statement can also include unknown variables.

Input:

$d := 2 + q + 4$

Output:

$d := 6 + q$

Let's list all the defined variables using the command who

Input:

$a := 2 ; b := 7 ; who$

Output:

$a := 2$

Output:

$b := 7$

Output:

Mathe's environment:

Output:

a := 2

Output:

b := 7

Besides the variable assignment statement there are other types of statements. An expression is special type of statement. It just consists of an expression to evaluate.

Input:

2 * 27

Output:

54

The system can also return integer results in the binary, octal or hexadecimal number system. Use `in bin` for binary output. Replace with `oct` and `hex` for octal and hexadecimal output.

Input:

2 * 27 in hex

Output:

0x36

The result of an expression statement is always stored in the variable `ans`. This way the result can be used in following statements.

Input:

2 - 7 ; 2 * ans

Output:

-5

Output:

-10

To separate multiple statements one can use a colon or a semicolon. The colon suppresses output from the preceding statement although the statement is executed normally. If the last or the only statement has neither a colon nor a semicolon, a semicolon is assumed and the result is printed.

Input:

2 ^ 4 ; 3 * 4 : 5 - 6

Output:

16

Output:

-1

Let's define a function on one variable and use it.

Input:

f(x) := 2 * x ^ 2 ; f(3)

Output:

f (x) := (2*(x^2))

Output:

18

Functions on more than one variables can also be defined.

Input:

g(x,y,z) := x + y + 2 * z : g(1, 2, 3)

Output:

9

We can use floating point numbers, numbers in the binary or hexadecimal system or even roman numerals.

Input:

2.5-1.8 ; 0b1101 ; 0h2f ; MCMLXXIV|r

Output:

0.7

Output:

13

Output:

47

Output:

1974

In fact, we can use any number system from 2 to 36. Let's see what 27 is in a number system with base 9. We check the result by also calculating it directly. This time we use the comparison equal sign '==' and not the assignment equal sign ':='.

Input:

```
327|9; ans == (3*9^2+2*9+7)
```

Output:

268

Output:

true

Vectors and matrices are also facilitated for. We'll define a matrix as well as a column vector and multiply them. The two elements of the resulting vector is then assigned to two different scalar variables. Here we chose to enter a row vector { 5, 6 } which we then transpose using the transpose operator '. We could also have defined a column vector directly: { 5; 6 }.

Input:

```
M := { 1, 2 ; 3, 4 } : v := { 5, 6 }' : w := M * v ; {k1, k2} := w'
```

Output:

```
w := {17 ; 39}
```

Output:

```
k1 := 17
```

Output:

```
k2 := 39
```

If v is unknown and w known, we can solve the linear system of equations $M * v = w$ using the `solvem` function. We'll compare the result to the known value of v to make sure the result is correct.

Input:

```
M := { 1, 2 ; 3, 4 } : w := {17 ; 39} : {v, H} := solvem(M, w); v == { 5, 6 }'
```

Output:

```
v := {5 ; 6}
```

Output:

```
H := 0
```

Output:

true

That H equals 0 indicate that we have found an exact solution.

We extend the system to become over-determined. M has become a third row. We execute 'w := M*v'. Calling the function `solvem(M, w)` would give us the original v. Instead we change w a little to make it over-determined and then solve for v. Because the system is over-determined we get a least square solution. This is indicated by H being NaN. Notice how the original v and the solution v2 to the over-determined system differ. Math tries to keep integers and fractions as far as possible. To convert the result to floating points we use the function `float`.

Input:

```
M := { 1, 2 ; 3, 4 ; 2, 3 } : v := { 5, 6 }' : w := M*v; w2:=w+{0,0,1}' ; {v2, H} := solvem(M, w2); v2 <> v; v2-v; float(ans)
```

Output:

w := {17 ; 39 ; 28}

Output:

w := {17 ; 39 ; 29}

Output:

v2 := {(14/3) ; (19/3)}

Output:

H := NaN

Output:

true

Output:

{(-1/3) ; (1/3)}

Output:

{-0.333333333333 ; 0.333333333333}

Expressions

- Basic data types
 - Arithmetic operators
 - Relations
 - Boolean operators
 - Intervals
 - Complex numbers
 - Vectors and matrices
 - Sets
 - Choices
 - Evaluation
 - Constants
-

Expressions

Basic data types

Basic data types include integers, rational numbers, floats, booleans and textstrings.

Example: `123; 4/6; 14.6; 2e-6; true; "Hello world!"`

Numbers are assumed to be decimal when not otherwise indicated. Numbers in other **number systems** are entered as `0b1101` (binary), `0o700` (octal), `0d100` (decimal), `0x2f` (hex) or `0h2f` (hex). Yet other systems can be used: `xxx|yy`, where `xxx` is a number in the number system with basis `yy` ≤ 36 . Roman numerals are entered as `mcmlxxxiv|r`.

Example: `123|8; 123.45|12;`

Textstrings are bounded by double quotes. A backslash is used to indicate a double quote or a backslash in a textstring.

Example: `"The \"best\" task ever"; "This is a single backslash \\ in the middle of a textstring";`

Arithmetic operators

- | | |
|------------------|---|
| <code>+</code> | Addition |
| <code>-</code> | Subtraction |
| <code>*</code> | Multiplication |
| <code>/</code> | Division |
| <code>!</code> | Faculty function |
| <code>mod</code> | Integer modulo, i.e. the rest |
| <code>div</code> | Integer division, the rest is discarded |
| <code>^</code> | Power |
| <code>%</code> | Percent |

Relations

Relations between two expressions are expressed using the following operators:

- < less than
- <= less than or equal
- > greater than
- >= greater than or equal
- == equal
- <> not equal

The result of a relation is an boolean expression.

Boolean operators

Boolean operators are:

- and (&&) Conjunction
- nand Negation of conjunction
- or (||) Disjunction
- nor Negation of disjunction
- xor (^) Exclusive disjunction
- not (~) Negation
- > Conditional / implication
- <- Reverse conditional / implication
- <-> Biconditional / biimplication

For four of the boolean operators the word form and the symbol are interchangeable. (x and y) is the same as (x && y)

For bitvectors the bv... functions are available. Please see the functions page.

Intervals

Intervals are entered as [x..y],]x..y], [x..y[and]x..y[.

Example : [2 .. 6 [

Complex numbers

Complex numbers are constructed using the imaginary unit (i).

Example : 2+3*i

Vectors and matrices

Vectors and matrices are entered using braces { } as delimiters. Use commas to separate elements of a row vector or of a row in a matrix - use semicolons to separate rows in a matrix.

Example : rowvector := { 1, 2, 3 }; columnvector := { 4; 5; 6 }; matrix := { 1, 2; 3, 4 }

Indices to a matrix are surrounded by square bracket []. Indices of first row and column are one - not zero. Indices are integers or integer intervals.

Example : M[2, 3] := 24

Example : M[[1 .. 2], 3] := { 24; 31 }

Operators only working on matrices and vectors include:

- .* Matrix multiplication, element by element
- ./ Matrix division, element by element
- ' Matrix transposing

Example: `a={1,2;3,4}; a.*a; a./a; a.^a; a'`

Sets

Sets are stored and manipulated as vectors.

Example: `a={1,2,3,4}; b={3,4,5,6}; intersect(a,b)`

Choices

Choices are made with the expression if-then-else.

Syntax: `if <boolean expression> then <expression> else <expression>`

Example: `if 3>a then [1..2] else [3..4]`

Evaluation

The evaluation of an expression can be delayed using the **delay** operator (`'`).

Example: `2+'a+3'`

When the first character entered is a `'`, the whole input is assumed to be a comment.

Example: `' This is a comment`

The function `eval(exp, level)` evaluates the expression up to a certain level:

- 1: Basic rules of arithmetic, e.g. $2 + 3 = 5$, are applied. Variables and functions are not evaluated
- 2: Variable names are substituted with their definition once. Function call parameters are evaluated
- 3: (Standard) Variables are treated like on level 2, only recursively. Functions are evaluated
- 4: Expressions are evaluated to floating point values where applicable

Function `evalf(exp)` is defined as `eval(exp, 4)`

Constants

A number of **constants** are known to the system:

`i`, `pi`, `e` and `inf`.

`goto top`

Statements

- Variable assignment statement.
 - Multiple variable assignment statement.
 - Function assignment statement.
 - Matrix assignment statement.
 - Plot statement
 - For statement
 - While statement
 - If statement
 - Keyword statement
 - Expression statement
 - Expression statement with numbersystem
 - Procedure definition statement
 - Procedure execution statement
-

Statements

Several types of statements are supported.

Variable assignment statement.

Syntax: `<variable> := <expression>`

Example: `x := 5`

Multiple variable assignment statement.

Syntax: `{ <var1>, ... } := <expression>`

Example: `{x,y} := {5,7}`

Function assignment statement.

Syntax: `<function> (<var1>, ...) := <expression>`

Example: `f(x,y) := x+y`

Matrix assignment statement.

One or more elements in a matrix are assigned a value. If the matrix does not exist, it is created in the environment. If the row and column index selects an element outside the current matrix, the matrix is expanded to include the selected element. Other elements are then assigned the value zero.

Syntax: `<matrix-variable>[row index, column index] := <expression>`

Example: `M[2,3] := 2*q`

Use ... as row and/or column index to include all rows/columns.

Example: `M[2,...] := {1,2,3}`

The row index can be omitted. It is interpreted as follows.

Syntax: `<matrix-variable>[column index] = <matrix-variable>[..., column index]`

Example: `M[2] := 5`

Functions, variables including matrices and procedures share **name-space**. Names are case-sensitive.

The following words are **keywords** and should not be used as names for variables and matrices:

who, restart, version, div, mod, true, false, and, nand, or, nor, not, xor, if, then, else, while, do, end, for, to, step, next, kill, quit, done, exit, stop, i, pi, e, NaN, inf.

Plot statement

The statement plots one dataset.

Syntax: `plot(<plot type>, <plot symbol>, X, Y)`

Syntax: `plot(<plot type>, <plot symbol>, Y)`

Syntax: `plot(<plot type>, <plot symbol>, X, Y, Z)`

Syntax: `plot(<plot type>, <plot symbol>, X, Y, a, b, c)`

The plot types `xy`, `polar`, `pie`, `bar`, `col`, `par` and `fun` are available. Available plot symbols are `dot`, `star`, `plus`, `cross`, `ring` and `line`. The plot type, the plot symbol and the X vector may be omitted. Then the following default values are used: Plot type `xy`, plot symbol `line` and X vector `{1, 2, . . . , n}`. The plot types `bar` and `col` can be used with one or two data series - with two the first data series is the lower values and the second the upper values. Plot type `pie` works only with one data series. For plot type `xy` the Y data series is plotted as a function of the X data series. For plot type `polar` the radia are given in X and the angles in Y. For plot type `fun` the first argument after an eventual `plotsymbol` is the function-expression to plot. The next argument indicates the variable and interval for the plot. For plot type `par` the arguments must contain two function expressions followed by an argument with the independent variable set equal to the selected interval. Every plot statement can take zero, one, two or three additional string arguments. The additional arguments `a`, `b` and `c` are used as 1. axis label, 2.axis label and plot title.

Example: `plot(xy, star, {1, 2, 3, 4}, {20, 3, 17, -4})`

Example: `plot(polar, line, {1, 2, 3, 4}, {20, 3, 17, -4})`

Example: `plot(bar, {20, 3, 17, -4})`

Example :

`plot(xy, line, {1, 2, 3, 4}, {20, 3, 17, -4}, "Time", "Earnings", "Results")`

Example: `plot(fun, k^2-5*k+6, k=[-5..5])`

Example: `plot(par, cross, sin(t), cos(t), t=[0..2*pi])`

For statement

The statement list is repeated a number of times.

Syntax: `for <variable> := <expression> to <expression> do <list of statements> next`

Syntax: `for <variable> := <expression> to <expression> step <expression> do <list of statements> next`

Example: `for a:=1 to 5 step 2 do a^2; a^3 next`

While statement

The statement repeats a statement list until a condition is true. The condition must be an expression, which can be evaluated to true or false.

Syntax: `while <boolean expression> do <list of statements> end`

Example: `a:=1: while a<4 do a^2; b:=a: a:=b+1: end`

If statement

The if chooses between two statement lists depending upon a condition. The last statement list can be left out. The condition must be an expression, which can be evaluated to true or false.

Syntax: `if <boolean expression> then <list of statements> else <list of statements> end`

Syntax: `if <boolean expression> then <list of statements> end`

Example: `if a>5 then b:=3 else c:=2 end`

Keyword statement

A keyword statement consists of one word only.

`who`: Lists the environment, i.e. user-defined variables and functions.

`restart`: Clears the environment.

`version`: Prints system information.

Expression statement

The result is stored in variable `<ans>`

Syntax: `<expression>`

Example: `1+5`

Expression statement with numbersystem

The result is stored in variable <ans>

Syntax: <expression> in bin|oct|dec|hex|roman

Syntax: <expression> in base n

Example: 1+5 in bin

Procedure definition statement

The list of statements are stored in the environment under the procedure name. The variables are used to initialize the environment when the procedure is executed.

Syntax: proc <procedure name>(<variable list>) := (local | global)
<statement list> end

Example: proc g(a,b) := local a; b; a*b end

Procedure execution statement

The list of statements stored under the procedure name are executed. The values of the expressions are assigned to the variables listed as the procedure was defined. These values can be known to only the function (local) or be remembered when the procedure has executed (global). This must be set when the procedure is defined.

Syntax: do <procedure name>(<expression list>)

Example: do g(1,2)

Use **semicolon** or **colon** to separate multiple statements. Colon suppresses printing output, while semicolon allows for output to be printed. The last or only statement does not need a semicolon or a colon. When one is missing, a semicolon is assumed.

Example: 1+2; 2+3: 3+4

goto top

Functions

- Functions on scalars
abs sign floor ceil trunc round float frac sqrt roots exp ln log2 log10 sin cos
tan csc sec cot
 - Functions on intervals
lower upper
 - Functions on complex numbers
conj real imag abs angle
 - Functions on sets
makeset union intersect difference symdifference ismember
 - Elementary matrix and vector functions
I ones zeros rand nrows ncolumns rot90 fliplr fliptb lowertrian uppertrian diag
det submatrix dot cross all exists
 - Functions on bit vectors
bvmake bvtoint bvtosint bvzero bvsign bvpart bvsplit bvconcat bvneg bvadd
bvsub bvmul bvnot bvand bv NAND bvor bvnor bvxor bvxnor
 - Solving functions
solve solveb solvei solvef solvem inv pinv forwardsub backwardsub pluq pldvq
 - Symbolic and numeric computation
diff diffi int intf expand factor
 - Data analysis functions
cumsum cumprod moments sum prod mean variance variances stddev stddevs
sort quartiles max min median geomean harmean
 - Data manipulation functions
op nops subs subsop map
 - Other functions
gcd lcm space eval evalf tostring primefactors
-

Functions on scalars

Function : **abs**

Name : Absolute value

Syntax : `abs(x)`

Description : The abs function returns the absolute value of a number.

Example : `abs(-2) == 2`

Function : **sign**

Syntax : `sign(x)`

Description : The sign function returns minus one if the value of x is less than zero, zero if the value is zero and one if the value is greater than zero.

Example : `sign(-10) == -1; sign(0) == 0; sign(200) == 1`

Function : **floor**

Syntax : `floor(x)`

Description : The floor function rounds the value towards minus infinity. It returns an integer.

Example : `floor(2.7) == 2`

Example : `floor(-2.7) == -3`

Function : **ceil**

Name : Ceiling

Syntax : `ceil(x)`

Description : The ceil function rounds the value towards plus infinity. It returns an integer.

Example : `ceil(2.7) == 3`

Example : `ceil(-2.7) == -2`

Function : **trunc**

Name : Truncate

Syntax : `trunc(x)`

Description : The trunc function rounds the value towards zero. It returns the integer part of the value.

Example : `trunc(2.7) == 2`

Example : `trunc(-2.7) == -2`

Function : **round**

Syntax : `round(x)`

Description : The round function round the value to the nearest integer.

Example : `round(2.7) == 3`

Example : `round(2.3) == 2`

Function : **float**

Syntax : `float(x)`

Description : The float function converts integers to floating point values.

Example : `float(2) == 2.0`

Function : **frac**

Name : Fraction

Syntax : `frac(x)`

Description : The frac function ignores the integer part of the value and returns the fraction part.

Example : `frac(2.4) == 0.4`

Function : **sqrt**

Name : Square root

Syntax : `sqrt(x)`

Description : The function returns the square root of the value

Example : `sqrt(4) == 2`

Function : **roots**

Syntax : `roots(x,y)`

Description : The roots function returns the y'th root of x.

Example : `roots(8,3) == 2`

Function : **exp**

Name : Exponential

Syntax : `exp(x)`

Description : The function returns the exponential value, that is the e^x .

Example : `exp(0) == 1`

Function : **ln**

Name : Natural logarithm

Syntax : `ln(x)`

Description : The function returns the natural logarithm of the value.

Example : `ln(e) == 1`

Function : **log2**

Name : Logarithm base 2

Syntax : `log2(x)`

Description : The function returns the logarithm base 2 of the value.

Example : `log2(4) == 2.0`

Function : **log10**

Name : Logarithm base 10

Syntax : `log10(x)`

Description : The function returns the logarithm base 10 of the value.

Example : `log10(1000) == 3.0`

Function : **sin**

Name : Sine

Syntax : `sin(x)`

Description : The sine function of an angle returns the ratio of the length of the opposite side to the length of the hypotenuse. The angle is assumed to be in radians.

Example : `sin(pi/2) == 1.0`

Function : **cos**

Name : Cosine

Syntax : `cos(x)`

Description : The cosine function of an angle returns the ratio of the length of the adjacent side to the length of the hypotenuse. The angle is assumed to be in radians.

Example : `cos(pi) == -1.0`

Function : **tan**

Name : Tangens

Syntax : `tan(x)`

Description : The tangens function of an angle returns the ratio of the opposite side to the length of the adjacent side. The angle is assumed to be in radians.

Example : `tan(pi) == 0`

Function : **csc**

Name : Cosecant

Syntax : `csc(x)`

Description : The cosecant function of an angle returns the ratio of the length of the hypotenuse to the length of the opposite side. The angle is assumed to be in radians. The functions is the reciprocal of sinus.

Example : `csc(pi/2) == 1.0`

Function : **sec**

Name : Secant

Syntax : `sec(x)`

Description : The secant function of an angle returns the ratio of the length of the hypotenuse to the length of the adjacent side. The angle is assumed to be in radians. The functions is the reciprocal of cosinus.

Example : `sec(pi) == -1.0`

Function : **cot**

Name : Cotangens

Syntax : `cot(x)`

Description : The cotangens function of an angle returns the ratio of the length of the adjacent side to the opposite side. The angle is assumed to be in radians.

Example : `cot(pi) == Inf`

goto top

Functions on intervals

Function : **lower**

Name : Lower boundary

Syntax : `lower([l..u])`

Description : The lower function returns the lower boundary l of an interval.

Example : `lower ([2..8]) == 2`
Function : **upper**
Name : Upper boundary
Syntax : `upper([l..u])`
Description : The upper function returns the upper boundary u of an interval.
Example : `upper ([2..8]) == 8`
goto top

Functions on complex numbers

Function : **conj**
Name : Conjugate
Syntax : `conj(r+j*i)`
Description : The conjugate function of a complex number returns a complex number with same real part and with the imaginary part multiplied with minus one.
Example : `conj(2*i) == -2*i`

Function : **real**
Name : Real part
Syntax : `real(r+j*i)`
Description : The real function returns the real part of a complex number.
Example : `real(2+3*i) == 2`

Function : **imag**
Name : Imaginary part
Syntax : `imag(r+j*i)`
Description : The imag function returns the imaginary part of a complex number
Example : `imag(2+3*i) == 3`

Function : **abs**
Name : Absolute value
Syntax : `abs(r+j*i)`
Description : The abs function returns the absolute value of a complex number, that is distance in the complex plane of the complex number from zero.
Example : `abs(3+4*i) == 5`

Function : **angle**
Syntax : `angle(r+j*i)`
Description : The angle function of a complex number returns the angle in the complex plane between the vector, which goes from zero to the complex number, and the vector from zero to 1.
Example : `angle(2+2*i) - pi/4 == 0`
goto top

Functions on sets

Function : **makeset**
Syntax : `makeset(V)`
Description : The makeset function takes a matrix as argument, eliminates double elements in the matrix and returns the remaining elements as a row vector.
Example : `makeset({a,b,c,d,c}) == {a,b,c,d}`
Function : **union**
Syntax : `union(V1,V2)`

Description : The union function takes two sets in form of matrices and returns the union in form of a matrix. The union includes all elements in either of the two given sets.

Example : $\text{union}(\{1, 2, 3\}, \{3, 4, 5\}) == \{1, 2, 3, 4, 5\}$

Function : **intersect**

Name : Intersection

Syntax : $\text{intersect}(V1, V2)$

Description : The intersection function takes two sets in form of matrices and returns the intersection in form of a matrix. The intersection consists of all elements that occur in both of the two given sets.

Example : $\text{intersect}(\{1, 2, 3\}, \{3, 4, 5\}) == 3$

Function : **difference**

Syntax : $\text{difference}(V1, V2)$

Description : The difference function takes two sets in form of matrices and returns the difference in form of a matrix. The difference consists of all elements that occur in the first set except those that also occur in the latter set.

Example : $\text{difference}(\{1, 2, 3\}, \{3, 4, 5\}) == \{1, 2\}$

Function : **symdifference**

Name : Symmetric difference

Syntax : $\text{symdifference}(V1, V2)$

Description : The symmetric difference function takes two sets in form of matrices and returns the symmetric difference in form of a matrix. The symmetric difference consists of all elements that occur in the first or the latter set except for those that occur in both sets.

Example : $\text{symdifference}(\{1, 2, 3\}, \{3, 4, 5\}) == \{1, 2, 4, 5\}$

Function : **ismember**

Syntax : $\text{ismember}(V, x)$

Description : The ismember function evaluates to true if the element x is a member of the set V.

Example : $\text{ismember}(\{1, 2, 3, 4, 5\}, 12) == \text{false}$

goto top

Elementary matrix and vector functions

Function : **I**

Name : Identity matrix

Syntax : $I(n)$

Syntax : $I(r, c)$

Description : The identity function evaluates to the identity matrix with side length n, or with row count r and column count c.

Example : $I(3) == \{1, 0, 0; 0, 1, 0; 0, 0, 1\}$

Example : $I(3, 2) == \{1, 0; 0, 1; 0, 0\}$

Function : **ones**

Syntax : $\text{ones}(n)$

Syntax : $\text{ones}(r, c)$

Description : The ones function evaluates to a matrix of ones with side length n, or with row count r and column count c.

Example : $\text{ones}(3) == \{1, 1, 1; 1, 1, 1; 1, 1, 1\}$

Example : $\text{ones}(3, 2) == \{1, 1; 1, 1; 1, 1\}$

Function : **zeros**

Syntax : $\text{zeros}(n)$

Syntax : $\text{zeros}(r, c)$

Description : The zeros function evaluates to a matrix of zeros with side length n, or with row count r

and column count c.

Example : `zeros(3) == {0,0,0;0,0,0;0,0,0}`

Example : `zeros(3,2) == {0,0;0,0;0,0}`

Function : **rand**

Name : Random number(s)

Syntax : `rand([1..u])`

Syntax : `rand(n)`

Syntax : `rand(n,[1..u])`

Syntax : `rand(r,c)`

Syntax : `rand(r,c,[1..u])`

Description : The random number function returns one or more random numbers. If a side length n or row count r and column count c is given, a matrix with those attributes is returned. Else a scalar is returned. If an interval is specified, at random numbers lie in that interval. If not, they lie in the interval from zero to one.

Example : `rand(4,2,[2..10])`

Function : **nrows**

Name : Number of rows

Syntax : `nrows(M)`

Description : The `nrows` function of a matrix evaluates to the number of rows in that matrix.

Example : `nrows({1,2,3;4,5,6}) == 2`

Function : **ncolumns**

Name : Number of columns

Syntax : `ncolumns(M)`

Description : The `ncolumns` function of a matrix evaluates to the number of columns in that matrix.

Example : `ncolumns({1,2,3;4,5,6}) == 3`

Function : **rot90**

Name : Rotate 90 degrees clockwise

Syntax : `rot90(M)`

Description : The `rot90` function of a matrix evaluates to the matrix rotated 90 degrees clockwise.

Example : `rot90({1,2;3,4}) == {3,1;4,2}`

Function : **fliplr**

Name : Flip left-right

Syntax : `fliplr(M)`

Description : The `fliplr` function of a matrix evaluates to the matrix flipped over a vertical axis.

Example : `fliplr({1,2;3,4}) == {2,1;4,3}`

Function : **fliptb**

Name : Flip top-bottom

Syntax : `fliptb(M)`

Description : The `fliptb` function of a matrix evaluates to the matrix flipped over a horizontal axis.

Example : `fliptb({1,2;3,4}) == {3,4;1,2}`

Function : **lowertrian**

Name : Lower triangular matrix

Syntax : `lowertrian(M)`

Description : The lower triangular function of a matrix with dimension (r, c) evaluates to the matrix with all elements, where $c > r$, set to zero and all elements, where $r = c$, set to one.

Example : `lowertrian({1,2,3;4,5,6;7,8,9}) == {1,0,0;4,1,0;7,8,1}`

Function : **uppertrian**

Name : Upper triangular matrix

Syntax : `uppertrian(M)`

Description : The upper triangular function of a matrix with dimension (r, c) evaluates to the matrix

with all elements, where $r > c$, set to zero.

Example : `uppertrian({1,2,3;4,5,6;7,8,9}) == {1,2,3;0,5,6;0,0,9}`

Function : **diag**

Name : Diagonal

Syntax : `diag(V)`

Syntax : `diag(M)`

Description : The diagonal function of a vector evaluates to a matrix with the vector a diagonal and all other elements zero. The diagonal function of a matrix evaluates to a vector consisting of the diagonal elements of the matrix.

Example : `diag({1,2,3,4}) == {1,0,0,0;0,2,0,0;0,0,3,0;0,0,0,4}`

Example : `diag({1,2,3,4;5,6,7,8}) == {1,6}`

Function : **det**

Name : determinant

Syntax : `det(M)`

Description : The det function of a matrix evaluates to the determinate of the matrix

Example : `det({a11,a12;a21,a22}) == a11 * a22 - a21 * a12`

Function : **submatrix**

Syntax : `submatrix(M,r,c)`

Description : The submatrix function evaluates to the matrix M with row r and column c deleted

Example : `submat({1,2;3,4} , 2, 1) == 4`

Function : **dot**

Name : scalar (dot) product

Syntax : `dot(V1,V2)`

Description : The dot function evaluates to the scalar produkt of the two vectors. Complex vectors are not supported.

Example : `dot({1,0},{0,2}) == 0`

Function : **cross**

Name : vector product

Syntax : `cross(V1,V2)`

Description : The cross function of two three element vectors evaluates to the vector product of the two vectors

Example : `cross({1,0,0},{0,1,0}) == {0,0,1}`

Function : **all**

Syntax : `all(M)`

Description : The all function of a matrix evaluates to true if all elements in the matrix evaluates to true

Example : `all({false, true, true}) == false`

Function : **exists**

Syntax : `exists(M)`

Description : The exists function of a matrix evaluates to true if any element in the matrix evaluates to true

Example : `exists({false, true, true}) == true`

goto top

Functions on bit vectors

Function : **bvmake**

Name : make bit vector

Syntax : `bvmake(n, x)`

Description : The bvmake functions converts the integer x to a n-bit bit vector.

Example : `bvmake(6,13) == {0, 0, 1, 1, 0, 1}`

Example : `bvmake(4,-3) == {1, 1, 0, 1}`

Function : **bvtoint**

Name : bit vector to integer

Syntax : `bvtoint(V)`

Description : The bvtoint function converts a bit vector V to an unsigned integer.

Example : `bvtoint({1, 0, 1}) == 5`

Function : **bvtosint**

Name : bit vector to signed integer

Syntax : `bvtosint(V)`

Description : The bvtoint function converts a bit vector V to a signed integer.

Example : `bvtosint({1, 1, 0, 1}) == -3`

Function : **bvzero**

Name : bit vector resize (zero bits)

Syntax : `bvzero(n, V)`

Description : The bvzero function extends or chops a bit vector V to length n padding with zero bits when necessary.

Example : `bvzero(5, {0, 0, 1}) == {0, 0, 0, 0, 1}`

Example : `bvzero(5, {1, 0, 1}) == {0, 0, 1, 0, 1}`

Example : `bvzero(2, {1, 0, 1}) == {0, 1}`

Function : **bvsign**

Name : bit vector resize (sign bits)

Syntax : `bvsign(n, V)`

Description : The bvsign function extends or chops a bit vector V to length n padding with the sign bit when necessary.

Example : `bvsign(5, {0, 0, 1}) == {0, 0, 0, 0, 1}`

Example : `bvsign(5, {1, 0, 1}) == {1, 1, 1, 0, 1}`

Example : `bvsign(2, {1, 0, 1}) == {0, 1}`

Function : **bvpart**

Name : Extract part of bit vector

Syntax : `bvpart(V,n2,n1)`

Description : The bvpart functions returns bits n1 to n2 of the bit vector V. $1 \leq n1 \leq n2 \leq \text{ncolumns}(V)$. Please note, that the higher bits come before lower bit and that the bits are numbered from one and upwards.

Example : `bvpart({0, 1, 1, 0, 1}, 4, 1) == {1, 1, 0, 1}`

Function : **bvsplit**

Name : split bit vector in halves

Syntax : `bvsplit(V)`

Syntax : `bvsplit(V,n1,n2)`

Description : The bvsplit functions with one argument splits the given bit vector V into halves. If the number of bits is odd, the second bit vector returned has one more bit than the first one.

Example : `bvsplit({0, 0, 1, 1, 0, 1}) == {{0, 0, 1}, {1, 0, 1}}`

Example : `bvsplit({0, 1, 1, 0, 1}) == {{0, 1}, {1, 0, 1}}`

Function : **bvconcat**

Name : bit vector concatenation

Syntax : `bvconcat(V1,V2)`

Description : The bvconcat functions concatenates two bit vectors.

Example : `bvconcat({0, 1, 0},{1, 0, 0}) == {0, 1, 0, 1, 0, 0}`

Example : `bvconcat({0, 1, 0, 0},{1, 0}) == {0, 1, 0, 0, 1, 0}`

Function : `bvneg`

Name : negate signed integer

Syntax : `bvneg(V)`

Description : The `bvneg` function negates a bit vector `V`.

Example : `bvneg({1, 1, 0, 1}) == {0, 0, 1, 1}`

Function : `bvadd`

Name : bit vector addition

Syntax : `bvadd(V1, V2)`

Syntax : `bvadd(n, V1, V2)`

Description : The `bvadd` functions adds two bit vectors. The result is `n` long or as long as the longest of the two bit vectors. The vectors are when needed extended with 0 bits, i.e. as unsigned bit vectors.

Example : `bvadd({0, 1, 0}, {1, 0, 0}) == {1, 1, 0}`

Example : `bvadd(6, {0, 1, 0}, {1, 0, 1, 0, 0}) == {0, 1, 0, 1, 1, 0}`

Function : `bvsub`

Name : bit vector subtraction

Syntax : `bvsub(V1, V2)`

Syntax : `bvsub(n, V1, V2)`

Description : The `bvsub` functions subtracts two bit vectors. The result is `n` long or as long as the longest of the two bit vectors. The vectors are when needed extended with 0 bits, i.e. as unsigned bit vectors.

Example : `bvsub({1, 0, 0}, {0, 1, 0}) == {0, 1, 0}`

Example : `bvsub(6, {0, 1, 0}, {1, 0, 1, 0, 0}) == {1, 0, 1, 1, 1, 0}`

Function : `bvmul`

Name : bit vector multiplication

Syntax : `bvmul(V1, V2)`

Syntax : `bvmul(n, V1, V2)`

Description : The `bvmul` functions multiplies two bit vectors. The result is `n` long or as long as the concatenation of the two bit vectors. The vectors are when needed extended with 0 bits, i.e. as unsigned bit vectors.

Example : `bvmul({0, 1, 0}, {1, 0, 0}) == {0, 0, 1, 0, 0, 0}`

Example : `bvmul(8, {0, 1, 0}, {1, 0, 0}) == {0, 0, 0, 0, 1, 0, 0, 0}`

Function : `bvnot`

Name : bit vector negation

Syntax : `bvnot(V)`

Syntax : `bvnot(n, V)`

Description : The `bvnot` functions generates the bit vector equivalent to the 'not' operator. The result is `n` long or as long as the bit vector argument. The argument bit vector is when needed extended with 0 bits, i.e. as unsigned bit vectors.

Example : `bvnot({0, 1, 0}) == {1, 0, 1}`

Example : `bvnot(5, {0, 1, 0}) == {1, 1, 1, 0, 1}`

Function : `bvand`

Name : bit vector and

Syntax : `bvand(V1, V2)`

Syntax : `bvand(n, V1, V2)`

Description : The `bvand` functions applies the boolean operator 'and' to the two bit vectors. The result is `n` long or as long as the longest of the two bit vectors. The vectors are when needed extended with 0 bits, i.e. as unsigned bit vectors.

Example : `bvand({1, 1, 0, 0}, {1, 0, 1, 0}) == {1, 0, 0, 0}`

Example : `bvand(6, {1, 1, 0}, {1, 0, 1, 0, 0}) == {0, 0, 0, 1, 0, 0}`

Function : `bvrand`

Name : bit vector nand

Syntax : `bvnand(V1,V2)`

Syntax : `bvnand(n,V1,V2)`

Description : The `bvnand` functions applies the boolean operator 'nand' to the two bit vectors. The result is n long or as long as the longest of the two bit vectors. The vectors are when needed extended with 0 bits, i.e. as unsigned bit vectors.

Example : `bvnand({1, 1, 0, 0},{1, 0, 1, 0}) == {0, 1, 1, 1}`

Example : `bvnand(6,{1, 1, 0},{1, 0, 1, 0, 0}) == {1, 1, 1, 0, 1, 1}`

Function : **bvor**

Name : bit vector or

Syntax : `bvor(V1,V2)`

Syntax : `bvor(n,V1,V2)`

Description : The `bvor` functions applies the boolean operator 'or' to the two bit vectors. The result is n long or as long as the longest of the two bit vectors. The vectors are when needed extended with 0 bits, i.e. as unsigned bit vectors.

Example : `bvor({1, 1, 0, 0},{1, 0, 1, 0}) == {1, 1, 1, 0}`

Example : `bvor(6,{0, 1, 0},{1, 0, 1, 0, 0}) == {0, 1, 0, 1, 1, 0}`

Function : **bvnor**

Name : bit vector nor

Syntax : `bvnor(V1,V2)`

Syntax : `bvnor(n,V1,V2)`

Description : The `bvnor` functions applies the boolean operator 'nor' to the two bit vectors. The result is n long or as long as the longest of the two bit vectors. The vectors are when needed extended with 0 bits, i.e. as unsigned bit vectors.

Example : `bvnor({1, 1, 0, 0},{1, 0, 1, 0}) == {0, 0, 0, 1}`

Example : `bvnor(6,{0, 1, 0},{1, 0, 1, 0, 0}) == {1, 0, 1, 0, 0, 1}`

Function : **bvxor**

Name : bit vector exclusive or

Syntax : `bvxor(V1,V2)`

Syntax : `bvxor(n,V1,V2)`

Description : The `bvxor` functions applies the boolean operator 'xor' to the two bit vectors. The result is n long or as long as the longest of the two bit vectors. The vectors are when needed extended with 0 bits, i.e. as unsigned bit vectors.

Example : `bvxor({1, 1, 0, 0},{1, 0, 1, 0}) == {0, 1, 1, 0}`

Example : `bvxor(6,{0, 1, 0},{1, 0, 1, 0, 0}) == {0, 1, 0, 1, 1, 0}`

Function : **bvxnor**

Name : bit vector exclusive nor

Syntax : `bvxnor(V1,V2)`

Syntax : `bvxnor(n,V1,V2)`

Description : The `bvxnor` functions applies the boolean operator 'xnor' to the two bit vectors. The result is n long or as long as the longest of the two bit vectors. The vectors are when needed extended with 0 bits, i.e. as unsigned bit vectors.

Example : `bvxnor({1, 1, 0, 0},{1, 0, 1, 0}) == {1, 0, 0, 1}`

Example : `bvxnor(6,{0, 1, 0},{1, 0, 1, 0, 0}) == {1, 0, 1, 0, 0, 1}`

goto top

Solving functions

Function : **solve**

Name : Symbolic solver

Syntax : `solve(x, v)`

Syntax : `solve(x == y, v)`

Description : The solve function finds feasible solutions to the equations $x==0$ or $x==y$ for the variable v .

Example : `solve(2*x==12,x) == 6`

Function : **solveb**

Name : Boolean solver

Syntax : `solveb(b, v)`

Syntax : `solveb(b, {v1,v2,...})`

Syntax : `solveb({b1,b2,...},v)`

Syntax : `solveb({b1,b2,...},{v1,v2,...})`

Description : The solveb function finds feasible solutions to the Boolean variables $v,v1,v2,\dots$ using the requirements stated as Boolean expressions $b,b1,b2,\dots$

Example : `solveb({b1 or b2, not b1},{b1,b2}) == {false, true}`

Function : **solvei**

Name : Integer solver

Syntax : `solvei(b, v==[l..u])`

Syntax : `solvei(b, {v1==[l1..u1],v2==[l2..u2],...})`

Syntax : `solvei({b1,b2,...},v)`

Syntax : `solvei({b1,b2,...},{v1==[l1..u1],v2==[l2..u2],...})`

Syntax : `solvei(b, v==[l..u], x)`

Syntax : `solvei(b, {v1==[l1..u1],v2==[l2..u2],...}, x)`

Syntax : `solvei({b1,b2,...},v, x)`

Syntax : `solvei({b1,b2,...},{v1==[l1..u1],v2==[l2..u2],...}, x)`

Description : Solvei finds integer solutions to the Boolean expressions $b,b1,b2$ for the variables $v,v1,v2$ in the given intervals. If an optional expression x is given, the solution(s) where x has it's maximum is returned. A matrix is returned with each row being the values of the variables for one solution. If no solutions are found, NaN is returned.

Example : `solvei({x+y < 3, y+z > 2},{x==[0..5], y==[0..10], z==[2..7]})`

Example : `solvei({x+y < 3, y+z > 2},{x==[0..5], y==[0..10], z==[2..7]}, 2*x+3*z)`

Function : **solvef**

Name : Numeric solver

Syntax : `solvef(x, v==[l..u])`

Description : The solvef function tries to find a feasible solution to the equation $x==0$ in the interval $[l..u]$.

Example : `solvef(x^2-8*x+12,x==[0..5])`

Function : **solvem**

Name : Matrix solver

Syntax : `solvem(M1, M3)`

Description : Solvem finds the solution $M2$ to the equation $M1*M2=M3$. If $M1$ is regular, $\{M2,0\}$ is returned. If the system is over-determined the least square solution $M2$ is found and $\{M2, NaN\}$ is returned. If the system is under-determined the minimum norm solution as well as a generating matrix H are found and returned: $\{M2,H\}$

Example : `solvem({1,2;3,4},{13, 29})`)`

Example : `A:={1,2,3,4;4,5,6,7}; b:={1,7}`; {x0,H}:=solvem(A,b);`

$A \cdot x_0 = b$; $A \cdot (x_0 + H \cdot \{2, 3\}') = b$

Function : **inv**

Name : Matrix inversion

Syntax : `inv(M)`

Description : Inv finds the solution X to the equation $M \cdot X = I$, where I ist the identity matrix, if M is square and regular. If M is not square and regular, the function returns NaN. In this case the function pinv calculates a pseudoinverse.

Example : `inv({1, 2; 3, 4})`

Function : **pinv**

Name : Matrix pseudo-inversion

Syntax : `pinv(M)`

Description : Pinv finds the solution X to the equation $M \cdot X = I$, where I ist the identity matrix. The solution can be the inverse or a pseudoinverse.

Example : `pinv({1, 2, 3; 4, 5, 6})`

Function : **forwardsub**

Name : Forward substitution

Syntax : `forwardsub(M1, M2)`

Description : Forwardsub finds the solution X to the equation $M1 \cdot X = M2$ by performing forward substitution.

Example : `forwardsub({1, 0; 3, 1} , {2, 9})`

Function : **backwardsub**

Name : Backward substitution

Syntax : `backwardsub(M1, M2)`

Description : Backwardsub finds the solution X to the equation $M1 \cdot X = M2$ by performing backward substitution.

Example : `backwardsub({1, 3; 0, 1} , {9, 2})`

Function : **pluq**

Name : P-L-U-Q matrix factorization

Syntax : `pluq(M)`

Description : Pluq returns a rowvector containing then rank followed by the P, L, U and Q matrices. If the matrix is not fullrank, the factorization may not be successfull.

Example : `pluq({1, 2; 3, 4})`

Function : **pldvq**

Name : P-L-D-V-Q matrix factorization

Syntax : `pldvq(M)`

Description : Pldvq returns a rowvector containing the rank followed by the P, L, D, V and Q matrices. If the matrix is not fullrank, the factorization may not be successfull.

Example : `pldvq({1, 2; 3, 4})`

goto top

Symbolic and numeric computation

Function : **diff**

Name : Differentiation

Syntax : `diff(x, v)`

Syntax : `diff(x, v==y)`

Description : The function diff finds the derivative of the expression x with respect to the variable v.

Example : `diff(2*x^2-12*x+2, x)`

Example : `diff(2*x^2-12*x+2, x==1)`

Function : **diff**

Name : Numerical differentiation

Syntax : `diff(x, v==y)`

Description : The function `diff` finds the derivative of the expression `x` with respect to the variable `v` at `v==y` numerically.

Example : `diff(2*x^2-12*x+2, x==1)`

Function : **int**

Name : integration

Syntax : `int(x, v)`

Syntax : `int(x, v=[1..u])`

Description : The function `int` finds the indefinite integral of the expression `x` with respect to the variable `v`. If an interval is given the definite integral is found.

Example : `int(2*x^2-12*x+2, x)`

Example : `int(2*x^2-12*x+2, x=[1..3])`

Function : **intf**

Name : numerical integration

Syntax : `intf(x, v=[1..u])`

Description : The function `intf` finds the definite integral of the expression `x` with respect to the variable `v` in the given interval.

Example : `intf(2*x^2-12*x+2, x=[1..3])`

Function : **expand**

Syntax : `expand(x)`

Description : The `expand` function expands the expression `x` so that multiplications and powers are expanded.

Example : `expand((x-2)*(x-4))`

Function : **factor**

Syntax : `factor(x)`

Description : The `factor` function collects common factor of the expression `x`.

Example : `factor(a*x*b+c*x*d)`

goto top

Data analysis functions

Function : **cumsum**

Name : Cummulated sum

Syntax : `cumsum(M)`

Description : The `cumsum` function of a matrix evaluates to a row vector containing the cummulated sum of the matrix elements taken row by row

Example : `cumsum({1,2,3,4;5,6,7,8}) == {1,3,6,10,15,21,28,36}`

Function : **cumprod**

Name : Cummulated product

Syntax : `cumprod(M)`

Description : The `cumprod` function of a matrix evaluates to a row vector containing the cummulated product of the matrix elements taken row by row

Example : `cumprod({1,2,3,4;5,6,7,8}) == {1,2,6,24,120,720,5040,40320}`

Function : **moments**

Syntax : `moments(M)`

Description : The `moments` function of a matrix evaluates to a row vector containing the number of elements, sum, mean, variance, skewness and curtosis of the matrix elements.

Example : `moments({1.0,2,3,4;5,6,7,8})`

Function : **sum**

Syntax : `(M)`

Description : The sum function of a matrix evaluates to the sum of the matrix elements.

Example : `sum({1,2,3,4;5,6,7,8}) == 36`

Function : **prod**

Name : Product

Syntax : `prod(M)`

Description : The prod function of a matrix evaluates to the product of the matrix elements.

Example : `prod({1,2,3,4;5,6,7,8}) == 40320`

Function : **mean**

Syntax : `mean(M)`

Description : The mean function of a matrix evaluates to the mean of the matrix elements.

Example : `mean({1,2,3,4;5,6,7,8}) == 9/2`

Function : **variance**

Name : Variance of population

Syntax : `var(M)`

Description : The variance function of a matrix evaluates to the variance of the matrix elements.

Example : `variance({1,2,3,4;5,6,7,8}) == 21/4`

Function : **variances**

Name : Variance of sample

Syntax : `var(M)`

Description : The variances function of a matrix evaluates to the variance of the matrix elements.

Example : `variances({1,2,3,4;5,6,7,8}) == 6`

Function : **stddev**

Name : Standard deviation of population

Syntax : `std(M)`

Description : The stddev function of a matrix evaluates to the standard deviation of the matrix elements.

Example : `stddev({1,2,3,4;5,6,7,8}) == sqrt(21/4)`

Function : **stddevs**

Name : Standard deviation of sample

Syntax : `std(M)`

Description : The stddevs function of a matrix evaluates to the standard deviation of the matrix elements.

Example : `stddevs({1,2,3,4;5,6,7,8}) == sqrt(6)`

Function : **sort**

Syntax : `sort(M)`

Description : The sort function of a matrix evaluates to a row vector containing the sorted matrix elements

Example : `sort({5,6,7,8; 1,2,3,4}) == {1,2,3,4,5,6,7,8}`

Function : **quartiles**

Syntax : `quartiles(M)`

Description : The quartiles function of a matrix evaluates to a row vector containing the minimum, first quartile, the mean, the third quartile and the maximum of the matrix elements.

Example : `quartiles({1.0,2,3,4;5,6,7,8}) == {1.0, 5/2, 9/2, 13/2, 8}`

Function : **max**

Name : Maximum value

Syntax : `max(M)`

Syntax : `max(x, y, ...)`

Description : The max function of a matrix evaluates to the maximum of the elements.

Example : $\max(\{1, 2, 3, 4; 5, 6, 7, 8\}) == 8$

Example : $\max(1, 2, 3, 4, x) == \max(4, x)$

Function : **min**

Name : Minimal value

Syntax : $\min(M)$

Syntax : $\min(x, y, \dots)$

Description : The min function of a matrix evaluates to the minimum of the elements.

Example : $\min(\{1, 2, 3, 4; 5, 6, 7, 8\}) == 1$

Example : $\min(1, 2, 3, 4, x) == \min(1, x)$

Function : **median**

Name : Median value

Syntax : $\text{median}(M)$

Description : The median function of a matrix evaluates to the the number separating the higher half of the matrix elements from the lower half.

Example : $\text{median}(\{1, 3, 5, 7, 9\}) == 5$

Function : **geomean**

Name : Geometric mean

Syntax : $\text{geomean}(M)$

Description : The geomean function of a matrix evaluates to the geometric mean of the matrix elements.

Example : $\text{geomean}(\{1.0, 2, 3, 4; 5, 6, 7, 8\})$

Function : **harmean**

Name : Harmonic mean

Syntax : $\text{harmean}(M)$

Description : The harmean function of a matrix evaluates to the harmonic mean of the matrix elements.

Example : $\text{harmean}(\{1.0, 2, 3, 4; 5, 6, 7, 8\})$

goto top

Data manipulation functions

Function : **op**

Syntax : $\text{op}(x, n)$

Description : The op function evaluates to the n`th subexpression of the expression x.

Example : $\text{op}(a+b, 2) == b$

Function : **nops**

Syntax : $\text{nops}(x)$

Description : The nops function evaluates to the number of subexpressions in the expression x.

Example : $\text{nops}(a+b) == 2$

Function : **subs**

Syntax : $\text{subs}(x1, x2, x3)$

Description : The subs function evaluates to the expression x1 with subexpressions x2 exchanged for subexpression x3

Example : $\text{subs}(a+b+c, b, k) == a+k+c$

Function : **subsop**

Syntax : $\text{subsop}(x1, n, x2)$

Description : The subsop function evaluates to the expression x1 with n`th subexpressions has been exchanged for subexpression x2

Example : `subsop(a+b+c,2,k) == a+k+c`

Function : **map**

Syntax : `map(M,x,v)`

Description : The map function evaluates to a matrix of the same dimensions as M, where for each element the expression x has been evaluated after assigning the corresponding element of M to the variable v.

Example : `map({1,2,3,4,5},x^2,x) == {1,4,9,16,25}`

goto top

Other functions

Function : **gcd**

Name : Greatest common denominator

Syntax : `gcd(x,y)`

Description : The gcd functions returns the largest integer, that is a denominator of x as well as y.

Example : `gcd(30,36) == 6`

Function : **lcm**

Name : Least common multiple

Syntax : `lcm(x,y)`

Description : The lcm function returns the smallest integers, that has x and y as denominator.

Example : `lcm(30,36) == 180`

Function : **space**

Syntax : `space(x, v==[l..u],n)`

Syntax : `space([l..u],n)`

Description : The function space evaluates the expression x for n different evenly distributed values of the variable v in the given interval [l..u] and returns the evaluated values of the x as a row vector. The expression x and the variable v may be omitted. In that case a linear distribution of the values are assumed.

Example : `space(x^2,x==[0..3],4) == {0,1,4,9}`

Example : `space([0..5],6) == {0,1,2,3,4,5}`

Function : **eval**

Name : Evaluate

Syntax : `eval(x,n)`

Description : The evaluate function evaluates the expression x, i.e. the first argument, depending on the integer n, i.e. the second argument. Setting n = 1 makes the system apply basic rules of arithmetic, n = 2 enables variable substitution as well, n = 3 turns function evaluation on and n = 4 makes certain expression evaluate to a floating point value.

Example : `eval(pi,4)`

Function : **evalf**

Name : Evaluate to floating point value

Syntax : `evalf(x)`

Description : The evalf function is defined as `evalf(x) := eval(x,4)`. Please see description for function eval.

Example : `evalf(pi)`

Function : **tostring**

Name : Convert expression to string

Syntax : `tostring(x)`

Description : Converts an expression to a string.

Example : `tostring(2*pi*x)`

Function : **primefactors**

Syntax : `primefactors(n)`

Description : The `primefactors` function returns a vector of the primefactors of n.

Example : `primefactors(12345) == {1 , 3 , 5 , 823}`

[goto top](#)

Examples

- Example 1: Data types
 - Example 2: Programming
 - Example 3: Optimization
 - Example 4: Numerics
 - Example 5: Set theory
 - Example 6: Logic & Boolean Algebra
 - Example 7: Calculus and symbolic computation
 - Example 8: Data analysis and statistics
 - Example 9: Graphics and visualization
 - Example 10: Linear Algebra
-

Example 1: Data types

Keywords: Ratios, floating point values and if-then-else expressions

Assume a FIFO queue with Poisson arrival process and one server. Following parameters are given: Offered traffic (A), average service time (s) and form factor (f). Using the Pollaczek-Khintchine formula the average waiting time for all customers (W1) and the average waiting time for customers with a positive waiting time (W2) are calculated.

Input:

```
temp:=f/(2*(1-A)): W1:=if (A<=1) then A*s*temp else -1; W2:=if
(A<=1) then s*temp else -1;
```

Output:

```
-: W1 := (if (A<=1) then (A*s*f/(2*(1-A)))else -1)
```

Output:

```
-: W2 := (if (A<=1) then (s*f/(2*(1-A)))else -1)
```

Assigning values to the input variables let us calculate W1 and W2.

Input:

```
temp:=f/(2*(1-A)): W1:=if (A<=1) then A*s*temp else -1: W2:=if
(A<=1) then s*temp else -1: A:=7/10: s:=1: f:=4 /* Typical form
factor for phone services */: W1; W2;
```

Output:

```
-: (14/3)
```

Output:

```
-: (20/3)
```

Notice how the answers are given as ratios. The system tries to keep ratios as long as they are not involved in a real expression. Using a real number in calculations yields real numbers in the answers.

Input:

```
temp:=f/(2*(1-A)): W1:=if (A<=1) then A*s*temp else -1: W2:=if
(A<=1) then s*temp else -1: A:=7/10: s:=1: f:=4 /* Typical form
factor for phone services */: W1: W2: 1.0*W1; 1.0*W2;
```

Output:

```
-: 4.666666666667
```

Output:

```
-: 6.666666666667
```

```
goto top
```

Example 2: Programming

Keywords: Functions, procedures, loops, for-next statement, while statement, intervals and if-then-else statement

Functions of type $\sin(t^n)$, $t \in [0..2\pi]$ are to be plotted for different integer values of n . First the time vector t is constructed as 150 values spread evenly in the interval. The function `space(<expression>, <variable> == <interval>, <n>)` evaluates the expression for n different evenly distributed values of the variable in the given interval and returns the evaluated values of the expression as a row vector.

Input:

```
t := space(x, x == [0..4.0*pi], 150);
```

Output:

```
-: t := {0.0 , 0.0490873852123 , 0.0981747704247 ...}
```

The functions of type $\sin(t^n)$ are stored in the environment as a function of t to variables.

Input:

```
f(x, n) := sin(x^n);
```

Output:

```
-: f(x, n) := sin(x^n);
```

A procedure utilizing the above defined function is created to output the value of n before plotting the function $\sin(t^n)$.

Input:

```
proc plotproc(x, n) := local y:=f(x, n); plot(line, x, y) end;
```

Output:

```
-: proc plotproc(x, n) := local y:=f(x, n); plot(line, x, y) end;
```

Using a for statement the procedure `plotproc` is called for a number of different integer values of n

Input:

```
for counter :=1 to 3 do counter; do plotproc(t, counter) next;
```

The for-next and the while statements provide for loops. Here a vector - or matrix with only one row - is constructed with every other element a square and the rest negativ numbers

Input:

```
for a:=1 to 9 step 2 do M[a]:=a^2: next; M
```

Input:

```
a:=2: while a<9 do M[a]:=-a: a:=a+2: end; M
```

Another way of constructing the matrix M is to use the if-then-else statement to separate between then two halves.

Input:

```
for a:=1 to 9 do if (a mod 2) == 0 then M[a]:=-a: else M[a]:=a^2:
```

```
end: next; M
```

```
goto top
```

Example 3: Optimization

Keywords: Vectors, matrices, loops, for-next, while, if-then-else

Mr. Smith would like to open a booth at the upcoming city fair. He has to find out which products to sell. From a store he can buy boxes of apples (A), bananas (B), oranges (O), nuts (N) and pears (P). He is sure, that selling both apples and pears will not do (not (A and P)), that he should only sell oranges with nuts and vice versa ($O \leftrightarrow N$), that he should at least sell one of the more solid fruits - nuts excluded (A or B or O or P), that he needs to sell apples or pears (A or P) and that he should either sell bananas or oranges, but not both of them (B xor O). Mr. Smith finds the set of possible solution to his

criteria.

Input:

```
solveb({not(A and P), O <-> N, A or B or O or P, A or P, B xor  
O}, {A,B,O,N,P});
```

Output:

```
-: {false , false , true , true , true ;  
false , true , false , false , true ;  
true , false , true , true , false ;  
true , true , false , false , false}
```

Each row contains a valid solution to the variables. The management at the fair requires all sellers to carry at least three different products. Having to choose between the first and the third option, he select the third based on his personal preferences. He will sell apples, oranges and nuts, but no bananas and pears.

Input:

```
X:={A,O,N}:
```

At the store he has to pay 4 for boxes of apples and 5 for boxes of oranges or nuts.

Input:

```
P:={4,5,5}:
```

A box of apples weighs 8, a box of oranges 5 and a box of butts 3.

Input:

```
W:={8,5,3}:
```

Knowing that he can only load 10 boxes in his car, he can buy from zero to 10 of each type of fruit, that he in total can pay the store up to 70 and that his car can carry boxes with a total weight of 40, he finds a number of possible solutions to his assignment problem. He uses the solvei function to find the valid solutions and then the nrows function to find the number of valid solutions.

Input:

```
result := solvei({X*P' <= 70, X*W' <= 40}, {A == [0..10], O==[0..10],  
N==[0..10]}): nrows(result);
```

Output:

```
-: 153
```

Having a large amount of solutions, Mr Smith wants to find the solution where he can make the biggest profit. He knows he can earn 3 for each box of apples, 4 for each box of oranges and 7 for each box of nuts. He uses the solvei function again - this time with a criteria to maximize.

Input:

```
E := {3,4,7}:
```

```
criteria := X*E':
```

```
result1 := solvei({X*P' <= 70, X*W' <= 40}, {A == [0..10],  
O==[0..10], N==[0..10]}, criteria);
```

Output:

```
-: result1 := {0,2,10}
```

Now he only gets only solution: He should buy two boxes of oranges and 10 of nuts but leave the apples alone. This is not consistent with his first decision to sell apples, oranges and nuts. He decides to add requirements to make sure he sells all three products.

Input:

```
result2 := solvei({X*P' <= 70, X*W' <= 40, A>0, O>0, N>0}, {A ==  
[0..10], O==[0..10], N==[0..10]}, criteria);
```

Output:

```
-: result2 := {1,1,9}
```

Once again he gets one solution: He should buy one box of apples, one boxes of oranges and nine boxes of nuts. His estimated profit is then

Input:

```
result2*E` ;
```

Output:

```
-: 70
```

Had he stayed with only oranges and nuts his profit would have been

Input:

```
result1*E` ;
```

Output:

```
-: 78
```

```
goto top
```

Example 4: Numerics

Keywords: Curve fitting, nonlinear regression, differentiation, integration, rootfinding

In an experiment the dependent values in vector y have been measured for the independent values in vector x , i.e. $y = f(x)$.

Input:

```
x:={2.0,2.2,2.6,2.7,2.8,2.9,3.1,3.2,3.3,3.6}` ;
```

```
y:={1.6,2.0,1.8,2.8,2.1,2.0,2.6,2.2,2.6,3.0}` ;
```

A nonlinear relationship between the values is assumed. In this case the model $y = a + b*x + c*x^2$ is chosen. The coefficients a , b and c are unknown. The vector f defines this relationship using the variable q .

Input:

```
f:={1, q, q^2} ;
```

The coefficients a , b and c are calculated using least-square regression.

Input:

```
for temp:=nrows(x) to 1 step -1 do
```

```
M[temp,...]:=map(x[temp,...],f,q): next; temp:='temp': M:
```

```
{coeff,H}:=solvem(M,y): /* H is NaN for an over-determined system */
```

```
{a,b,c}:=coeff` ;
```

Output:

```
-: a := 0.971497202483
```

```
-: b := 0.24799530993
```

```
-: c := 0.0717281648134
```

Now one can plot the measurements, the regression fit and the residual.

Input:

```
Y:=M*coeff: plot(x,y); /* Measurements */ plot(line,x,Y); /* Fitted
```

```
curve */ plot(x,y-Y); /* Residual */
```

Using the coefficients (a, b, c) and the model (f) the fitted polynomial p as a function of the independent variable q is constructed.

Input:

```
p:=sum(f.*coeff`); plot(p,q=[-10..10]);
```

The derivative of p is found and evaluated for $q=2.5$.

Input:

```
dp:=diff(p,q); subs(dp,q,2.1);
```

The derivative of p in $q=2.5$ can also be found directly without first explicitly differentiating the polynomial.

Input:

```
diff(p,q=2.1);
```

The indefinite and one definite integral of the polynomial p is found.

Input:

```
int(p,q); int(p,q==[2.0..3.0]);
```

Roots in p are determined.

Input:

```
solve(p,q)
```

```
goto top
```

Example 5: Set theory

Keywords: Union, intersection, difference, member, discrete mathematics

Given a number of elements in a vector or matrix one can construct the equivalent set, where identical elements are left out

Input:

```
A:={1,2,2,3,x,y}; A:=makeset(A);
```

The union as well as the intersection of two sets are found using the builtin functions. The difference of two sets are the elements of the first set that do not occur in the second set.

Input:

```
B:={3,4,5,y,z}; union(A,B); difference(A,B);
```

We can also find the elements that belong in one of the set, but not in both.

Input:

```
symdifference(A,B);
```

Let's see if x is a member of the two sets A and B .

Input:

```
ismember(A,x); ismember(B,x);
```

We can also find out if a set is a subset of another set using the `ismember` function in addition to the `all` function.

Input:

```
C:={3,x,y}; all(ismember(A,C)); all(ismember(B,C));
```

```
goto top
```

Example 6: Logic & Boolean Algebra

Keywords: Logic, booleans, relations, discrete mathematics

Given a number of boolean expressions one can try to simplify them using the default simplification capabilities.

Input:

```
x and (x and y); (x or not x) and y;
```

To find the solutions to more advanced boolean expressions one can use the `solveb` function. A number of expressions, which should evaluate to `true` and a list of variables are given as function arguments.

Input:

```
solveb({x and (x and y), (x or not x) and y},{x,y})
```

Bit vectors are constructed using the `bvmake` function. We construct two 6-bit bit vectors for the integers 13 and 5.

Input:

```
x:=bvmake(6,13); y:=bvmake(6,5);
```

Boolean operators for bit vectors are applied.

Input:

```
a:=bvand(x,y); b:=bvxor(x,y);  
Some arithmetic functions for bit vectors can also be used.
```

Input:

```
m:=bvmul(x,y); {hi,lo}:=bvsplit(m); bvtoint(hi); bvtoint(lo);  
goto top
```

Example 7: Calculus and symbolic computation

Keywords: Derivative, integral, equation solving, expand, factor

Given a function f one can find the derivative and the integral

Input:

```
f:=2*x^3-sin(x); df:=diff(f,x); F:=int(f,x);
```

Let's see if the results are correct.

Input:

```
int(df,x); f == ans; diff(F,x); f == ans;
```

Solving an equation is equal to find the roots of the equivalent function.

Input:

```
f:=2*(1/(a*ln(x)^2))^3; solve(f==12,x);
```

We plot the function to see the .

Input:

```
a:=0.5; plot(f,x==[2..20]);
```

The extrema is found by setting the derivative equal to zero. Numerical differentiation is used to check the result.

Input:

```
solve(diff(f,x),x); diff(f,x==ans);
```

An expression being a sum of many subexpressions containing common factors can be rewritten to better see the structure.

Input:

```
f:=x*a*y*b+c*x*d*y; factor(f);
```

Likewise an expression containing powers and multiplication can be expanded.

Input:

```
f:=2*(x+3)^3-(5*x-2)^4; expand(f);
```

goto top

Example 8: Data analysis and statistics

Keywords: Discriptive statistics, summary statistics, central tendency

Given a data set in a matrix, a number of statistics can be calculated.

Input:

```
X:={1,2,3,4;5,6,7,8;9,10,11,12}; s:=sum(X); p:=prod(X); m:=mean(X);  
v:=variance(X); sd:=stddev(X);
```

Beside the arithmetic mean, the geometric and the harmonic mean can be found.

Input:

```
g:=geomean(X); h:=harmean(X);
```

Finding the minimum, first quartile, mean, third quartile and maximum is achieved through the quartiles function.

Input:

```
{min, q1, av, q2, max} := quartiles(X);
```

A distribution of the data can be plotted.

Input:

```
X2:=sort(X); plot('col',X2);
```

goto top

Example 9: Graphics and visualization

Keywords: Graphics, visualization, plots, diagrams

The visualization functionality is centered around the plot statement. A function of one variable can be plotted directly

Input:

```
f:=2*x^2-5*x+3; plot(f,x==[-5..10]);
```

A parametric plot is constructed using two function expressions. Please note, that the plot type (par) and the plot symbol (cross) can be left out.

Input:

```
plot(par, cross, sin(t), cos(t), t==[0..2*pi]);
```

A data set can be presented in a bar plot, a column plot or in pie chart.

Input:

```
X:={1,4,2,7,3,2,6,4}; plot(bar,X); plot(col,X); plot(pie,X);
```

Assuming the data set represents the dependent variables corresponding to an independent variable 1,2,...,n we plot using the default xy plot type and the dot plot symbol.

Input:

```
plot(xy,dot,X);
```

In case we had both the independent and the dependent values we still use the xy plot type. This time the ring plot symbol is used.

Input:

```
X:={1,3,5,6,7,8}; Y:={5,3,2,7,6,4}; plot(xy,ring,X,Y);
```

goto top

Example 10: Linear Algebra

Keywords: Vectors, matrices, linear algebra, system of equations, change of basis

Given three vectors a1, a2 and a3 in an ordinary three-dimensional basis e.

Input:

```
a1 := {1,2,-1}; a2:= {0,3,-1}; a3:={-4,11,-2};
```

If the three vectors are a basis for the space, the matrix eMa consisting of the vectors as columns must be regular. Us use the inv functions to find the inverse matrix aMe. If however the function returns NaN, the matrix eMa is not regular.

Input:

```
eMa[... ,1] := a1`; eMa[... ,2] := a2`; eMa[... ,3] := a3`; aMe :=  
inv(eMa);
```

Having a regular matrix, we also have another basis for the space. We can transform the coordinates of the vectors eX1 and eX2 from basis e to basis a.

Input:

```
eX1 := {1,5,-2}`; eX2:= {-1,3,-1}`;
```

Input:

```
aX1 := aMe * eX1; aX2 := aMe * eX2;
```

The inverse transformation results in the old coordinates in basis e . We check to make sure.

Input:

$e_{X1} == e_{Ma} * a_{X1}; e_{X2} == e_{Ma} * a_{X2};$

Source: Eising, Jens: Lineær Algebra, Matematisk Institut, Danmarks Tekniske Højskole 1983, p. 154

[goto top](#)

Please note, that the Tusanga calculator only stores values between calculations when logged in.

Therefore all calculations based on each other must be entered at the same time if used outside of a session.

Try it out online

www.tusanga.com