



# VisualCalc Pro

VisualCalc Pro is an advanced programmable scientific calculator for Palm OS with complex, array and matrix operations and graphing.

Summary of features:

- Optimized ARM-code speeds calculations and construction of graphs up to 15 times
- Calculations with real and complex numbers
- Work with arrays and matrices
- Flow control: conditionally (if-elif-else) and repeatedly (while-continue-break) execute statements
- Optionally display results in rational fraction format
- Various mathematical functions: trigonometric, logarithmic, complex, statistical, matrix
- Solution of linear algebraic equations
- Calculation of derivatives and integrals
- Unlimited number of user variables
- Date and time arithmetic
- User-defined functions
- Multiline equations with size up to 16 Kbytes
- Saving equations for further work
- Function graphs with support for tracing the graphs
- Cartesian, polar and parametric graphs
- Various array plots: line, dot, bar (vertical and horizontal, stacked)
- 3D surface graphs
- Separate screen for variables and results
- Automatic calculation mode
- Step-by-step integrated debugger
- Ability to interrupt lengthy calculations
- Screen keyboard for input and editing
- Support for comments in the equation text
- Customizable fonts for editing and results
- Palm Standard (160x160), Hi-Res (320x320) and Hi-Res+ (320x480) resolutions supported
- Search for name and text of expression
- Import/export a text of expressions from/to the Memo Pad
- Reading and writing CSV-files
- Exchange of expressions between devices

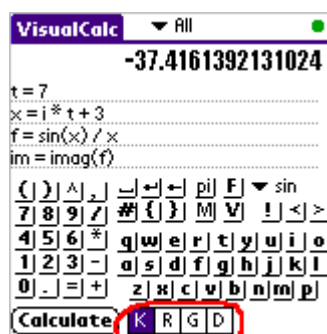
Home page: <http://vc.andrufka.com/>. Please send your comments and bug reports to [visualcalc@gmail.com](mailto:visualcalc@gmail.com)

- [Basic operations](#)
  - [Functions](#)
  - [Complex numbers](#)

- [Constants](#)
- [Variables](#)
- [Arrays](#)
- [Matrices](#)
- [Solution of linear algebraic equations](#)
- [Flow control](#)
- [Derivatives, integrals, interpolation](#)
- [Date-time calculations](#)
- [User-defined functions](#)
- [Working with libraries](#)
- [Results of calculation](#)
- [Graphs](#)
  - [Controlling the graph](#)
  - [Array plot](#)
  - [3D graphs](#)
- [Debugger](#)
- [Expression setup](#)
- [Preferences](#)
- [List of expressions](#)
- [Global find](#)
- [Exchange between applications and devices](#)
  - [Reading and writing CSV-files](#)
- [Built-in functions](#)
  - [Operator precedence](#)
  - [Arithmetic operations](#)
  - [Logic operations](#)
  - [Common functions](#)
  - [Trigonometric and exponential functions](#)
  - [Complex functions](#)
  - [Statistical functions](#)
  - [Matrix functions](#)
  - [Control functions](#)
  - [Date-time functions](#)

## Basic operations

Input and editing of expression is carried out on the screen of the virtual keyboard (button "K" on the switch of screens):



*Switch of screens*

For input, it is possible to use the screen virtual keyboard or graffiti. Calculation of value of expression is made automatically (if the corresponding option in [Preferences](#) is set) or by pressing button "Calculate".

Individual parts of expression should settle down on separate lines or be separated from each other by commas:

```
x = 1  
y = sin(x)
```

or

```
x = 1, y = sin(x)
```

Purpose of individual buttons:

**#** - the beginning of the comment. The comment can be any text up to the end of a line.

**pi** - the [constant](#)  $\pi = 3.14159...$

**F** - input of last used [function](#), which name is displayed to the right of the button.

**M** - calls the editor for input of a [matrix](#).

**V** - the list of user variables.

In the right top corner, there is a small circle. It is the indicator of progress of calculations and correctness of the entered expression. Green color means that expression does not contain syntax errors. Red color means that expression contains error and result is invalid. This indication is especially useful in automatic calculation mode. Tapping this indicator interrupts long calculations.

Standard operations:

[+](#) addition

[-](#) subtraction

[\\*](#) multiplication

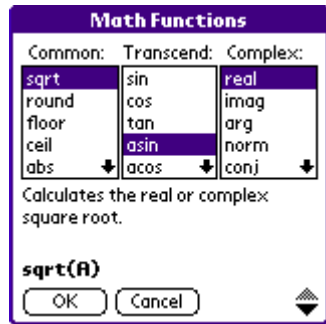
[/](#) division

[^](#) power

[!](#) factorial

## Functions

The name of mathematical function can be entered symbol-by-symbol or you can select it from the list. The list of last used functions is to the right of button "F". The item " more..." opens the window with brief descriptions of the built in functions:



[sin\(x\)](#) - sine

[cos\(x\)](#) - cosine

[tan\(x\)](#) - tangent

[asin\(x\)](#) - arcsine

[acos\(x\)](#) - arccosine

[atan\(x\)](#) - arctangent

[atan\(y, x\)](#) - arctangent of y / x

[sinh\(x\)](#) - hyperbolic sine

[cosh\(x\)](#) - hyperbolic cosine

[tanh\(x\)](#) - hyperbolic tangent

[asinh\(x\)](#) - inverse hyperbolic sine

[acosh\(x\)](#) - inverse hyperbolic cosine

[atanh\(x\)](#) - inverse hyperbolic tangent

[sinc\(x\)](#) - sinc,  $\sin(x) / x$

[exp\(x\)](#) - exponential e to the x

[ln\(x\)](#) - natural logarithm

[lg\(x\)](#) - base 10 logarithm

[sqrt\(x\)](#) - square root

[round\(x\)](#) - round to nearest integer

[floor\(x\)](#) - round towards minus infinity

[ceil\(x\)](#) - round towards infinity

[abs\(x\)](#) - absolute value (module)

[mod\(x, y\)](#) - remainder of division x on y

[hypot\(x, y\)](#) - hypotenuse of a right triangle

[deg\(x\)](#) - convert radians to degrees

[rad\(x\)](#) - convert degrees to radians

[gcd\(x, y\)](#) - greatest common divisor

[lcm\(x, y\)](#) - least common multiple

[rat\(x\)](#) - rational fraction approximation

[rnd\(\)](#) - random numbers uniformly distributed in the interval (0, 1)

[rndn\(\)](#) - normally distributed random numbers

[seed\(n\)](#) - initialization of rnd()

[seedn\(n\)](#) - initialization of rndn()

[fzero\(x, f\(x\), a, b\)](#) - find a zero of a function f(x) lying between a and b

[fmin\(x, f\(x\), a, b\)](#) - find the local minimum of a function f(x) within the interval (a, b)

## Complex numbers

Complex numbers are entered in the form of

```
3 + 9i
or
3 + 9 * i
```

And the first variant is more preferable, since allows excluding one complex multiplication.

As a symbol of imaginary unit, it is possible to use either "i" or "j" (it is set in [expression setup](#)).

Operations with complex numbers:

[real\(z\)](#) - real part of complex number  
[imag\(z\)](#) - imaginary part of complex number  
[abs\(z\)](#) - absolute value (module)  
[arg\(z\)](#) - argument of complex number  
[norm\(z\)](#) - norm of complex number (a square of the module)  
[conj\(z\)](#) - complex conjugate  
[polar\(x, y\)](#) - complex number with the module x and argument y  
[sign\(z\)](#) - signum function

## Constants

The built in constants:

**pi** - ratio of a circle's circumference to its diameter, 3.14159...  
**e** - the base of the natural logarithm, 2.718...  
**i** or **j** - imaginary unit, a square root from (-1).

## Variables

The name of a variable should begin with the letter and can contain letters, digits and the underscore character "\_". Names are case sensitive, so that **t** and **T** are distinct variables.

The name of a variable can coincide with a name of any built in function if this function is not used in the given expression.

You can change the value of a variable, however it is impossible to change type of a variable:

```
z = 2 + 10i # complex variable
a = {1, 2, 3} # one-dimensional array
z = 5 # correct
z = a # error: assigning an array value to a complex variable
a = {1, 2} # correct
a = ident(3) # error: assigning a matrix value to an array variable
```

## Arrays

The array is the sequence concluded in braces. Arrays can be multidimensional.  
Examples of arrays:

- one-dimensional array of 3 elements:

```
a1 = {1, 2, 3}
```

- complex two-dimensional array of 3 elements - arrays of 3, 2 and 4 elements correspondingly:

```
a2 = {{1, 2, 3}, {4, 5}, {6 - 3i, 9, -2, 0}}
```

- array of two matrices
- `m = matrix({{2, 3, 7, 6}, {3, 5, 9, 9}, {4, 7, 6, 2}})`
- `n = matrix({{2, 4, 7}, {5, 3, 7}, {9, 1, 4}})`  
`am = {m, n}`

Function [zero](#) creates an array of zeros

```
zero(4) # array of 4 zeros  
zero(4i) # complex array of 4 elements (0 + 0i)
```

[seq](#) fills an array with the elements forming arithmetic sequence:

```
seq(2, 8) # {2, 3, 4, 5, 6, 7, 8}  
seq(4, -1.5, -2) # {4, 2.5, 1, -0.5, -2}
```

[rep](#) creates an array with a given number of identical elements:

```
rep(1, 5) # {1, 1, 1, 1, 1}  
v = {1, 2, 3}  
rep(v, 3) # {{1, 2, 3}, {1, 2, 3}, {1, 2, 3}}
```

Each element of an array can be accessed by its index. Numbering of elements begins with 1 or 0 (it is set in [Expression setup](#)). For example:

- 2-nd element of an array

```
a1(2)
```

- assignment to an element of a two-dimensional array

```
a2(1, 2) = 3.7
```

Most of functions work not only with numbers, but also with arrays. Thus function is calculated for each element of an array.

The functions working with an array as a whole:

[zero\(n\)](#) - create an array of n zeros

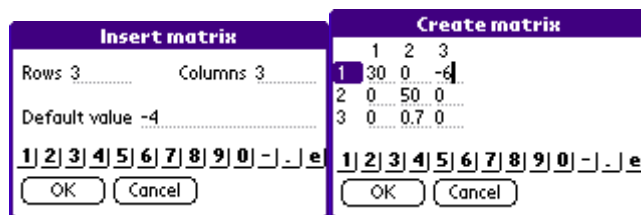
[seq\(from, to\)](#), [seq\(from, step, to\)](#) - arithmetic sequence

[rep\(v, n\)](#) - replicate a value multiple times

[min\(a\)](#) - smallest element  
[max\(a\)](#) - largest element  
[find\(a\)](#) - find indices of nonzero elements  
[sum\(a\)](#) - sum of elements  
[prod\(a\)](#) - product of elements  
[sort\(a\)](#) - sort array elements in ascending order  
[rev\(a\)](#) - inverse order of elements  
[mean\(a\)](#) - average value of elements  
[var\(a\)](#) - variance (the square of the standard deviation), biased estimate  
[sdev\(a\)](#) - standard deviation, biased estimate  
[med\(a\)](#) - median  
[corr\(a, b\)](#) - correlation of two arrays  
[range\(a, from, size\)](#) - returns size elements of an array starting with from  
[size\(a\)](#) - size of an array  
[join\(a, b\)](#) - join two arrays  
[merge\(a\)](#) - merge elements of the nested arrays

## Matrices

To insert a matrix into expression it is possible by means of the editor called by button "M" on the screen keyboard. It is necessary to specify the dimensions of a matrix and default value of elements:



The matrix can be converted from a two-dimensional array by means of function [matrix](#)

```
a = {{2, 4, 7}, {5, 3, 7}, {9, 1, 4}}
m = matrix(a) # matrix 3x3
```

or from an one-dimensional array by means of function [vector](#)

```
a = {2, 4, 7, -8}
v = vector(a) # matrix 4x1
```

Each element of a matrix can be accessed by its row and column indexes. Numbering begins with 1 or 0 (it is set in [Expression setup](#)). For example:

- 3-rd row, 2-nd column

```
m(3, 2)
```

- assignment to an element of a matrix

```
v(1, 1) = 9
```

Matrix functions:

[matrix\(a\)](#) - convert a two-dimensional array to a matrix  
[vector\(a\)](#) - convert an array to a matrix with one column  
[ident\(n\)](#) - n-by-n identity matrix  
[diag\(a\)](#) - diagonal matrix  
[zero\(m, n\)](#) - m-by-n zero matrix  
[rep\(v, m, n\)](#) - m-by-n matrix filled with v's value.  
[inv\(m\)](#) - matrix inverse  
[det\(m\)](#) - matrix determinant  
[trans\(m\)](#) - matrix transpose  
[solve\(A, B\)](#) - solution of set of linear equations  $Ax = B$   
[tr\(m\)](#) - sum of diagonal elements  
[eig\(m\)](#) - eigenvalues and eigenvectors of a symmetric matrix  
[block\(m, row, col, n\\_rows, n\\_cols\)](#) - return the block of matrix elements  
[array\(m\)](#) - transform a matrix into a two-dimensional array  
[size\(m\)](#) - size of a matrix  
[row\(m, n\)](#) - row of a matrix  
[col\(m, n\)](#) - column of a matrix  
[joinh\(m1, m2\)](#) - join two matrices horizontally  
[joinv\(m1, m2\)](#) - join two matrices vertically

## Solution of linear algebraic equations

A set of linear algebraic equations looks like this:

$$\begin{aligned}a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \dots + a_{1N}x_N &= b_1 \\a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \dots + a_{2N}x_N &= b_2 \\a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \dots + a_{3N}x_N &= b_3 \\&\vdots \\a_{M1}x_1 + a_{M2}x_2 + a_{M3}x_3 + \dots + a_{MN}x_N &= b_M\end{aligned}$$

M - number of the equations,  $x_1 \dots x_N$  - N unknowns,  $a_{11} \dots a_{MN}$ ,  $b_1 \dots b_M$  - known numbers.

If the number of unknowns is equal to number of the equations such system can be solved by means of function [solve](#):

```
2 x1 + 3 x2 + 7 x3 = -1
3 x1 + 5 x2 + 9 x3 = 2
4 x1 + 7 x2 + 6 x3 = 1
A = matrix({{2, 3, 7}, {3, 5, 9}, {4, 7, 6}})
b = vector({-1, 2, 1})
x = solve(A, b)
```

## Flow control

Use [if](#) to implement a conditional calculation of expressions:

```
a = 10 * rnd()
if (a < 3)
```



```

    k = 7
elif (a > 5)
    k = 0
else
    k = 8
end

```

Use the [while](#) keyword for calculation of repeating operations:

```

a = zero(10)
i = 1
while (i <= 10)
    a(i) = i
    i = i + 1
end

```

It is possible to skip current iteration of a loop by means of [continue](#) or to interrupt a loop by means of [break](#):

```

t = round(100 * rnd())
while (t != 56)
    t = t + 1
    if (t == 13)
        continue
    end
    k = t * t
    if (k == 100)
        break
    end
end

```

Use the [break\(n\)](#) to interrupt some nested loops.

Relational operations:

[<](#) less than  
[>](#) greater than  
[<=](#) less than or equal to  
[>=](#) greater than or equal to  
[==](#) equal to  
[!=](#) not equal to

When comparing arrays and matrices it is useful to use functions [all](#) and [any](#).

Logical operations

[and](#) - logical AND

[or](#) - logical OR

[not](#) - logical NOT

```

if (a < 10 and b >= 0 or not empty)
    y = 16
end

```

## Derivatives, integrals, interpolation

Function [df](#) returns the derivative of function at the point.

1) the derivative of  $f(x) = x^3$  with respect to  $x$  at  $x = 2$ :

$$(x^3)' = 3x^2$$

```
x = 2
df(x, x ^ 3) # 12
```

2) the second derivative of function  $f(x) = x^3$  with respect to  $x$  at  $x = 2$ :

$$(x^3)'' = 6x$$

```
x = 2
df(x, df(x, x ^ 3)) # 12
```

3) the mixed derivative of function  $f(x,y) = y^2x^3$  at  $x = 2, y = 3$ :

```
x = 2, y = 3
df(y, df(x, y ^ 2 * x ^ 3)) # 72
```

Function [quad](#) calculates the definite integral (quadrature)

$$\int_a^b f(x) dx$$

```
quad(x, f(x), a, b)
```

For example, it is necessary to calculate integral

$$\int_0^{\pi/2} \sin(x) dx$$

```
x = 0
quad(x, sin(x), 0, pi / 2) # 1
```

For interpolation of data function [polint](#) uses the interpolating polynomial of degree  $N - 1$  through the  $N$  points.

```
p = {2, 3, 5} # base points
t = {1, 2, 6} # x-coordinates of base points
x = 4
polint(x, t, p)
```

[df\(x, f\(x\)\)](#) the derivative of function  $f(x)$  at the point  $x$

[df\(x, f\(x\), h\)](#) the derivative of function  $f(x)$  at the point  $x$  with the initial step  $h$

[quad\(x, f\(x\), a, b\)](#) numerically evaluate integral, adaptive Gauss-Kronrod quadrature

[quadr\(x, f\(x\), a, b\)](#) numerically evaluate integral, Romberg quadrature

[polint\(x, t, p\)](#) polynomial interpolation  
[polint\(x, p\)](#) polynomial interpolation with default step

## Date-time calculations

VisualCalc provides support for calculating with date / time. There are two types of time variables in the program: time point and time duration. Time point is a location on the time scale:

```
now() # 2009-05-02 18:17:52 - the local date and time
date() # 2009-05-02 - today, the time part is set to 00:00:00
date(1999, 12, 31) # December 31, 1999
```

Time duration represents a length of time:

```
date(2000, 1, 1) - date(1999, 12, 31) # one day
date(1999, 12, 31) + day(1) # January 1, 2000
date(1999, 12, 31) + month(1) # January 31, 2000
date(1999, 12, 31) + time(23, 59, 59) # one second before the 2000
year
```

Duration, expressed in months or years, has various length in days. One month may cover a span of 28 to 31 days, one year - 365 or 366 days. Exact number of days depends on a time point to which duration is added:

```
date(2000, 1, 15) + month(1) # February 15, 2000
date(2000, 1, 31) + month(1) # February 29, 2000
date(2000, 2, 29) + month(1) # March 29, 2000
```

By default, all dates are represented in the Gregorian calendar. However, you can explicitly make a date in the Julian calendar or convert dates from one calendar to another:

```
julian(1582, 10, 4) # last day in the Julian calendar
julian(date(1917, 11, 7)) # October 25, 1917
```

You can extract various parts from the date-time:

```
dt = date(2009, 5, 11) + time(14, 38, 10.43)
y = year(dt) # 2009 - year
m = month(dt) # 5 - May
d = day(dt) # 11 - day of month
h = hour(dt) # 14 - number of hours
mi = minute(dt) # 38 - number of minutes
s = second(dt) # 10.43 - number of seconds
day_of_week = dow(dt) # 1 - Monday
day_of_year = doy(dt) # 131 - day of year
w = week(dt) # 20 - week number
```

If you want to calculate something like "the third Monday in April" see the [ndow](#) function:

```
ndow(2009, 4, 3, 1) # April 20, 2009
```

Date-time functions:

[date\(\)](#) - current date  
[date\(y, m, d\)](#) - date given as year, month, day  
[date\(dt\)](#) - extract the date part from a date-time  
[time\(\)](#) - current time  
[time\(h, m, s\)](#) - time specified in hours, minutes and seconds  
[time\(dt\)](#) - extract the time part from a date-time  
[now\(\)](#) - current date and time  
[ndow\(y, m, n, wd\)](#) - nth day of the week in the month of the specified year  
[year\(n\)](#) - duration in years  
[year\(dt\)](#) - year part of the date  
[month\(n\)](#) - duration in months  
[month\(dt\)](#) - month part of the date  
[day\(n\)](#) - duration in days  
[day\(dt\)](#) - day part of the date  
[hour\(n\)](#) - duration in hours  
[hour\(dt\)](#) - number of hours  
[minute\(n\)](#) - duration in minutes  
[minute\(dt\)](#) - number of minutes  
[second\(n\)](#) - duration in seconds  
[second\(dt\)](#) - number of seconds  
[dow\(dt\)](#) - day of week of the given date  
[doy\(dt\)](#) - day of year of the given date  
[week\(dt\)](#) - week number  
[leap\(y\)](#) - indicates whether the specified year is a leap year  
[leap\(dt\)](#) - indicates whether the given date is in a leap year  
[julian\(y, m, d\)](#) - date in the Julian calendar given as year, month, day  
[julian\(dt\)](#) - converts the Gregorian date to the Julian date  
[gregorian\(dt\)](#) - converts the Julian date to the Gregorian date

## User-defined functions

Repeating operations can be combined in separate function by means of keyword [func](#):

```
func inc(a)
    ret(a + 1)
end
```

Here function with a name *inc* and one argument *a* is defined. Function returns value of the argument increased by 1. This function can be used with any types of arguments, which support addition:

```
inc(5)           # 6
inc(5i)          # 1 + 5i
inc({1, 2, 3})   # {2, 3, 4}
inc(ident(2))    # matrix({{2, 1}, {1, 2}})
```

Another example - find zero element in an array:

```
func find_zero(a)
```

```

i = 1
while (i <= size(a))
    if (not a(i))
        ret(i)
    end
    i = i + 1
end
ret(0)
end
v = {1, 2, 0, 4, 5}
find_zero(v)          # 3
find_zero({1, 1})    # 0

```

Inside of function it is possible to change values of elements of arrays and matrices passed from outside:

```

func set_one(v, n)
    v(n) = 1
end
a = {0, 9, 2, 3, 4, 5}
set_one(a, 2)
a # {0, 1, 2, 3, 4, 5}

```

To access the global variable inside the function put the dot before the variable name:

```

global = 123 # global variable

func f(x)
    ret(.global + x)
end

f(7)    # 130

```

## Working with libraries

Frequently used constants and functions can be carried out into separate expression. Such expression needs a name which is beginning with the letter and not containing spaces (otherwise enclose the name within quotes). After that, the given expression can be used in any other expression by means of function [use](#), simply having specified its name.

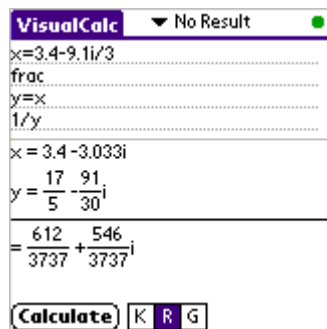
Using the constant:

1. expression with name "constants":
2. `#constants`
3. `g = 9.80665 # m/s^2`  
`c = 299792458 # m/s`
4. use of the constant in other expression:
5. `use(constants)`
6. `y0 = 100 # m`
7. `v0 = 0 # m/s`
8. `t = 3 # s`  
`y = y0 - (v0 * t + (g * t ^ 2) / 2) # m`

Using the user-defined function:

1. expression with name "lib1":
2. #lib1
3. # convert Fahrenheit temperature to Celsius degrees
4. func **Cel**(F)
5.     ret((F - 32) \* 5 / 9)
6.     end
6. use of the function in other expression:
7. **use(lib1)**  
   **Cel**(451)

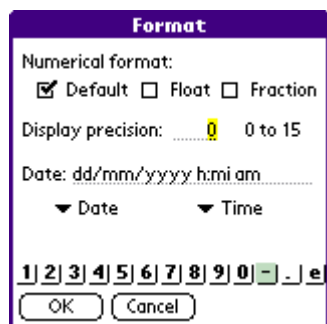
## Results of calculation



To show the results of calculations, tap the "R" button on the switch of screens. Here it is possible to see final values of all variables and the total result. If not all results are visible on the screen, it can be scrolled by stylus.

By default, all numbers are displayed in floating point format. Use [frac](#) if it is necessary to print a value in the form of rational fraction. All the variables, which have received the values after frac, will be displayed in the form of fractions. Use [float](#) to cancel the frac action. Keyword [none](#) allows to hide working variables.

Tapping the name of the variable, you can configure individual display of results:



- **Numerical format:** - controls format of numbers:
  - **Default** - default format
  - **Float** - floating point format
  - **Fraction** - rational fraction
- **Display precision:** - number of displayed significant digits (0 - default)
- **Date:** - string specifying format for displaying date-time variables
  - **Date** - list of predefined formats of date

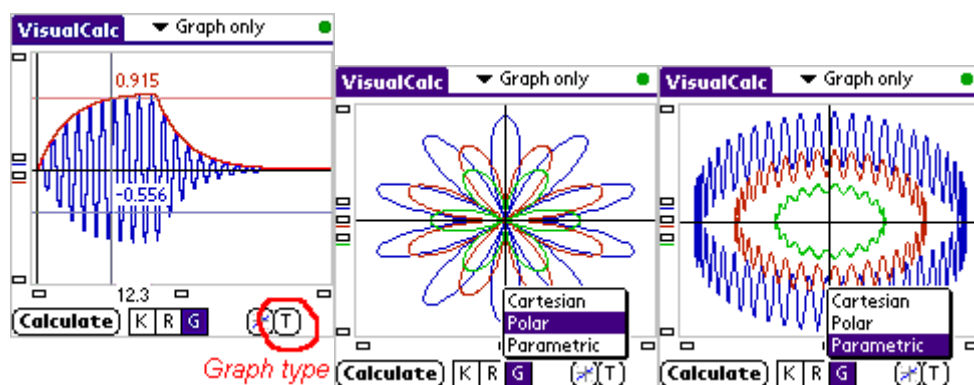
- **Time** - list of predefined formats of time

A string, specifying format for displaying a date-time, is composed of special codes, which are replaced with the appropriate fields of the date-time. Besides the codes this string can contain any symbols, which are displayed as is.

Date-time format codes:

Code	Description
am	meridian indicator
d	day of week (1 - Monday, 2 - Tuesday, ... 7 - Sunday)
day	name of day
dd	day of month (1 - 31)
ddd	day of year (001 - 366)
dy	abbreviated name of day
fff	milliseconds (000 - 999)
h	hour of day (1 - 12)
hh	hour of day (00 - 23)
iw	ISO 8601 week number
mi	minute (00 - 59)
mm	month (01 - 12)
mon	abbreviated name of month
month	name of month
ss	second (00 - 59)
yy	Last 2 digits of year
yyyy	4-digit year

## Graphs



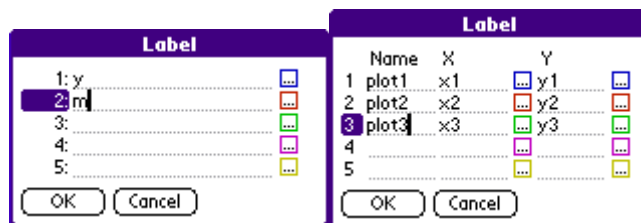
Tap "G" button to switch to graph screen. Tap on button "T" to select from one of three graph types, an array plot or a 3D surface graph:

- **Cartesian** - normal graph in cartesian coordinates.
- **Polar** - graph in polar coordinates. Angle specified in radians.
- **Parametric** - parametric graph. Independent variable controls X and Y values.
- **Plot** - [array plot](#).
- **3D Graph** - [3D surface graph](#).

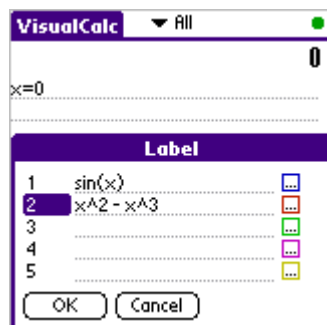
For construction of the graph it is necessary to create expression for argument and value of function. For example, to construct the graph of function  $f(t) = \sin(t)$  it is possible to write:

```
t = 0 # argument
f = sin(t) # value
```

To set a name of a variable which will be independent argument of graph tap on the marker under the graph in the middle of axis X. Similar markers to the left of axis Y allow to set names of functions for graphs



VisualCalc Pro accepts arbitrary expressions in place of function names:

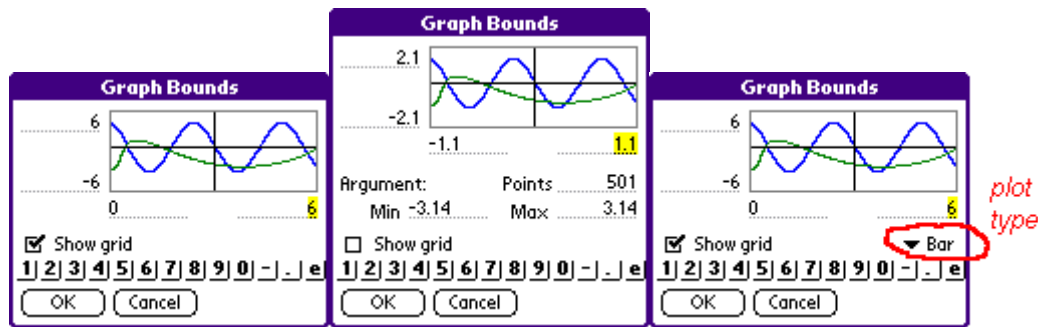


Set the axis limits by tapping on the corresponding markers in corners of the graph. There are additional parameters for polar and parametric graphs:

- **Min, Max** - minimum and maximum argument value.
- **Points** - number of graph points.
- **Show grid** - display a coordinate grid.

Here you can choose type of array plot.

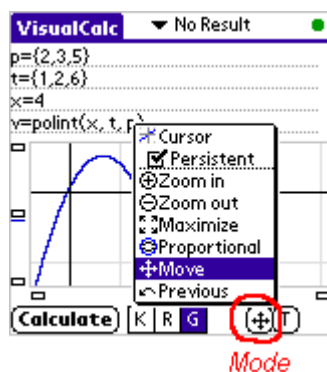




Tap on any place of the graph and you will see the screen cursor in the form of the vertical and horizontal lines crossed on the graph.

## Controlling the graph

The graph can be controlled by stylus. It is preliminary necessary to choose a mode:



- **Cursor** - screen cursor
- **Persistent** - "Persistent cursor" mode, i.e. the cursor remains on the screen after the termination of pressing
- **Zoom in** - increase scale of the graph
- **Zoom out** - decrease scale of the graph
- **Maximize** - fit the graph in the screen
- **Proportional** - identical scale on a vertical and horizontal
- **Move** - move the graph
- **Previous** - return to the previous view
- **Rotate** - rotate the [3D graph](#)

In "Persistent cursor" mode the cursor position can be set numerically. For this purpose it is necessary to press current value of an independent variable. The cursor control window will appear:

**Go to position**

x = 0.87248

Function 1 y 

Value 0

Function 2 z

Enter a new argument value and press the **Go** button.

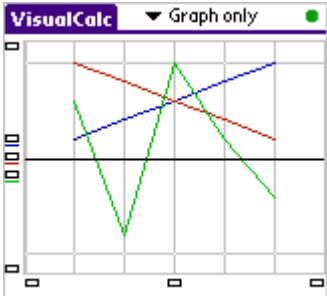
The **Find value** button finds the argument value, at which the **Function 1** obtains the value of **Value**.

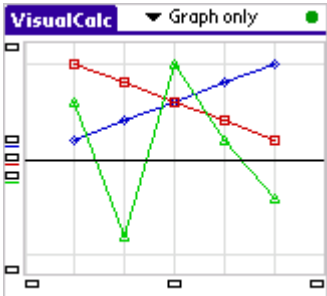
The **Min**, **Max**, **df/dx** buttons find the minimum, maximum and the derivative of the **Function 1** near the current cursor position.

The **Find intersection** button allows to find the intersection of the plotted functions **Function 1** and **Function 2**.

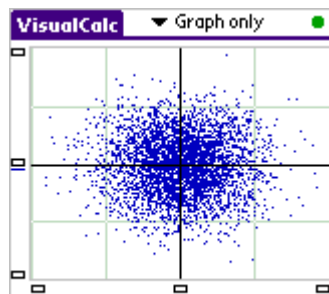
## Array plot

Types of array plot:

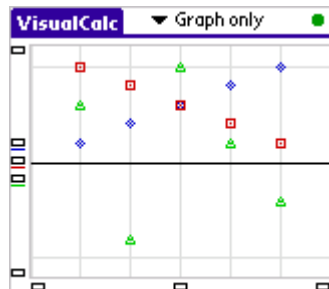
- 

**Line** - linear plot
- 

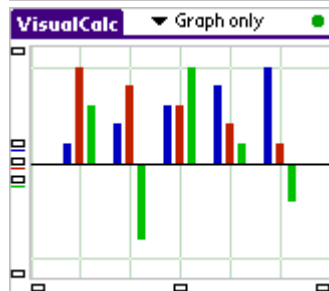
**MarkedLine** - linear plot with markers



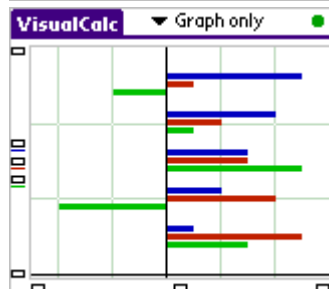
- **Dot** - dot chart



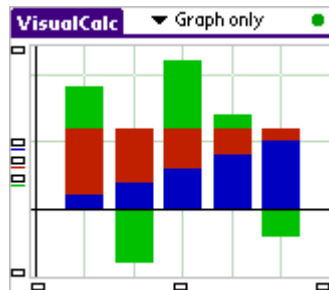
- **Marker** - markers



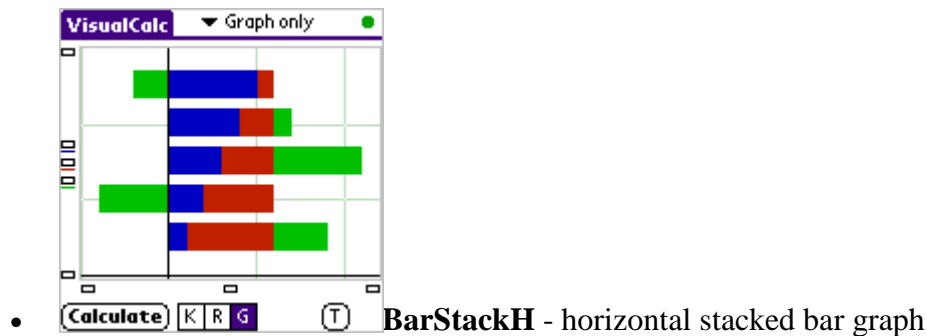
- **Bar** - vertical bar graph



- **BarH** - horizontal bar graph



- **BarStack** - stacked bar graph



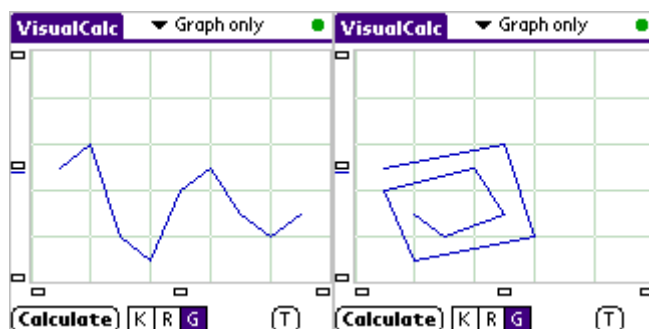
For construction of the plot, an array of values must be provided instead of function. It is possible not to specify argument on the axis X. In this case, elements of an array will be displayed on the plot with step 1. If an array is specified as an argument, elements of this array will be coordinates on the axis X for corresponding elements of an array of values.

For example, construct two plots for expression

$$A = \{5, 6, 2, 1, 4, 5, 3, 2, 3\}$$

$$B = \{1, 5, 6, 2, 1, 4, 5, 3, 2\}$$

Array A is located on the axis Y. In the first case, nothing is set as an argument on the axis X, in the second - array B:



## 3D graphs

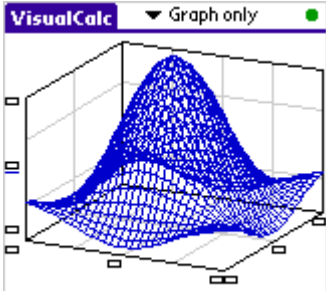
**3D Graph** draws a surface formed by crossing of lines of a uniform grid. Height Z is determined by the function, which depends on two variables. For example:

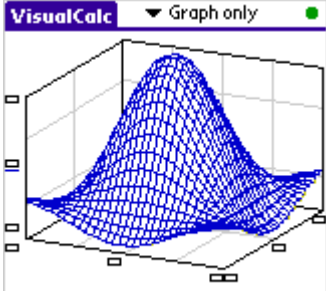
$$x = 0$$

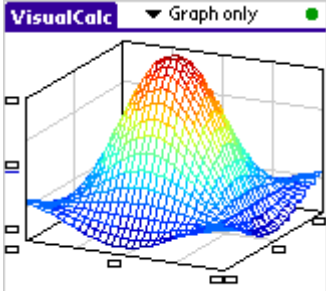
$$y = 0$$

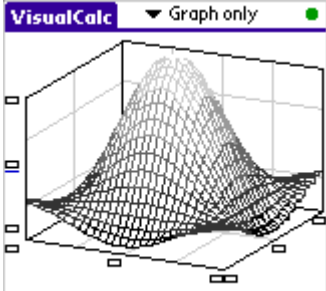
$$z = 9 * \text{sinc}(0.5 * x) * \text{sinc}(0.5 * y)$$

Types of the 3D-graphs:

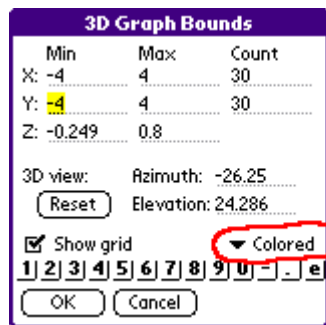
- 
 The image shows a 3D wireframe plot of a bell-shaped surface. The surface is composed of a grid of blue lines. The plot is displayed in a 3D coordinate system with axes. Below the plot, there is a toolbar with buttons labeled 'Calculate', 'K', 'R', 'G', and a circular arrow icon. The title bar of the window says 'VisualCalc' and 'Graph only'.
- Wireframe** - wireframe graph

- 
 The image shows a 3D mesh plot of a bell-shaped surface. The surface is composed of a grid of blue lines. The plot is displayed in a 3D coordinate system with axes. Below the plot, there is a toolbar with buttons labeled 'Calculate', 'K', 'R', 'G', and a circular arrow icon. The title bar of the window says 'VisualCalc' and 'Graph only'.
- Mesh** - wireframe mesh with hidden-surface elimination

- 
 The image shows a 3D colored mesh plot of a bell-shaped surface. The surface is composed of a grid of lines colored with a gradient from blue at the base to red at the peak. The plot is displayed in a 3D coordinate system with axes. Below the plot, there is a toolbar with buttons labeled 'Calculate', 'K', 'R', 'G', and a circular arrow icon. The title bar of the window says 'VisualCalc' and 'Graph only'.
- Colored** - wireframe mesh with color determined by Z so color is proportional to surface height

- 
 The image shows a 3D grayscale mesh plot of a bell-shaped surface. The surface is composed of a grid of lines in various shades of gray, darker at the peak and lighter at the base. The plot is displayed in a 3D coordinate system with axes. Below the plot, there is a toolbar with buttons labeled 'Calculate', 'K', 'R', 'G', and a circular arrow icon. The title bar of the window says 'VisualCalc' and 'Graph only'.
- Grayscale** - graph in shades of grey according to surface height

Set the axis limits by tapping on the corresponding markers in corners of the graph. There are additional parameters for 3D graphs:



**3D Graph Bounds**

Min	Max	Count
X: -4	4	30
Y: -4	4	30
Z: -0.249	0.8	

3D view: Azimuth: -26.25  
 Elevation: 24.286

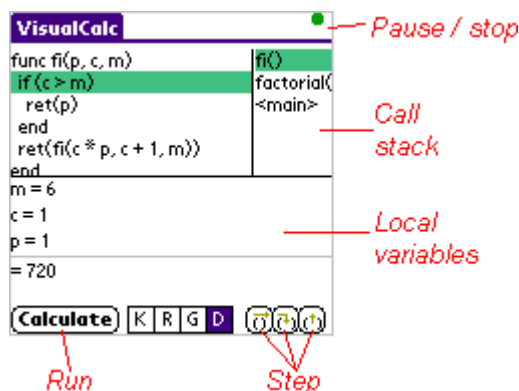
☐ Show grid

3D Graph type

- **Min, Max** - X-, Y-, and Z-axis limits.
- **Count** - quantity of grid lines on X and Y axes.
- **3D view** - viewpoint specification:
  - **Azimuth** - horizontal rotation about the Z-axis in degrees.
  - **Elevation** - vertical elevation of the viewpoint in degrees.
  - **Reset** - sets the default view, Azimuth = -37.5°, Elevation = 30°.
- **Show grid** - display a coordinate grid.




## Debugger

To step through the code and verify it works, you can use built-in debugger:



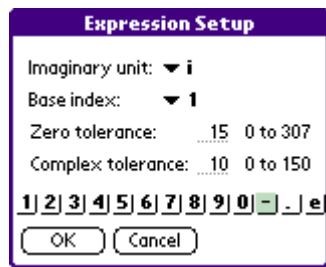
The debugger is activated by pressing the "D" button on the switch of screens.

To step through code the following buttons are used:

-  **Step over** - the debugger executes a step without going into a function
-  **Step into** - the debugger walks through your code one statement at a time
-  **Step out** - the debugger runs the program until execution returns from the current function

## Expression setup

Select "Expression - Setup..." in the menu:



**Imaginary unit** - a name of the variable used as imaginary unit (a square root from -1). Name may be **i** or **j**.

**Base index** - an index of the first element of an array and first row or column of a matrix. Index may be 1 or 0.

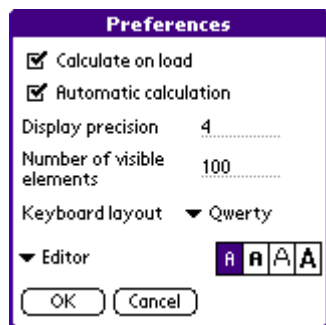
**Zero tolerance** controls how close a result must be to zero before VisualCalc displays it as such. By default numbers smaller than  $10^{-15}$  will be displayed as zero.

**Complex tolerance** controls how much larger the real or imaginary part of a number must be before VisualCalc stops displaying the smaller part.

The adjustments presented in the given panel are individual for each expression and are kept together with it.

## Preferences

Select "Options - Preferences..." in the menu:



**Calculate on load** - calculation of value of expression right after its loadings

**Automatic calculation** - automatic calculation of value of expression during its input or change

**Display precision** - set the number of significant digits to print results

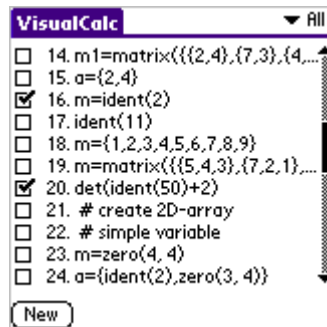
**Number of visible elements** - maximum number of elements of an array or a matrix, which are completely displayed on the screen of results

**Keyboard layout** - layout of the virtual screen keyboard

**Fonts** - selection of fonts for the editor of expression (Editor) and display of results (Results).

## List of expressions

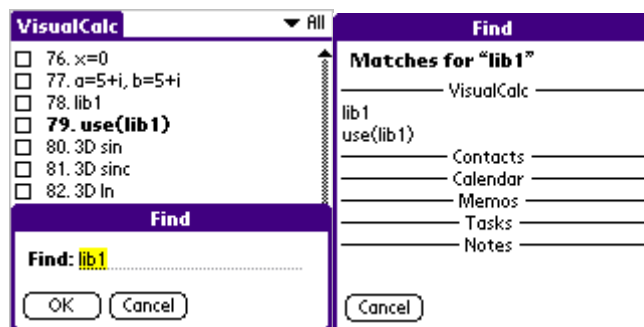
Select "Expression - Done" in the menu:



Here it is possible to assign name and category for expression or to remove unnecessary expressions.

## Global find

The Palm OS find application searches all applications on the Palm device. VisualCalc supports this feature and returns expressions, in name or text of which the desired text is contained:



## Exchange between applications and devices

A text of expression can be exported to the Memo Pad. Using the Palm Desktop such an expression can be edited on a personal computer and then imported again from the Memo Pad into the VisualCalc.

The menu command "Expression - Export to Memo" exports the text of the active expression. The name of this expression will be located on the first line of the Memo.

Choose the menu command "Expression - Import from Memo..." to import a text from the Memo Pad:



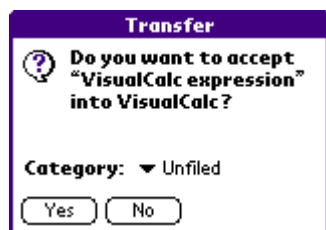


Select the Memo to be imported and tap the Import button. The Replace option affects what is done with the imported text. If the Replace option is checked, the imported text will replace the active expression. If the Replace option is unchecked, the new expression will be created from the imported text.

The menu command "Expression - Send" sends the active expression to another Palm Powered handheld. The transmission can use a variety of transport mechanisms, including infrared (Beam), SMS, Bluetooth, Email:



On the receiving handheld, you can select a category for an incoming expression:



## Reading and writing CSV-files

One of standard formats for data exchange is the CSV - text values separated by commas. VisualCalc can read and write these files by using the [csvread](#), [csvwrite](#). The files should be located on an external memory card.

CSV-file can only contain numeric values separated by commas. Reading the file produces a two-dimensional array, each element in which represents a separate line of the CSV-file. Empty values are filled with zeros.

Given the file `data.csv` that contains the comma-separated values

```
1, , 2, 3.4
32424,
9977766, 9939
```

To read the file, use

```
csvread('data.csv')
```

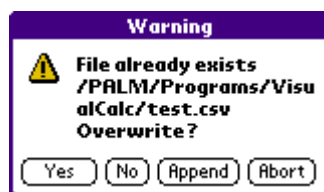
The result will be an array

```
{{1, 0, 2, 3.4}, {32424, 0}, {9977766, 9939}}
```

The array of real numbers can be stored in the file

```
a = {{1, -3}, {pi, e}}
csvwrite(a, 'test.csv')
```

If the file with the specified name already exists, then the warning will appear



If you click **Yes** the file will be overwritten, **No** - the file will remain unchanged, but the program will continue execution. When you click **Append** the data will be added to the end of the file. **Abort** button cancels the execution.

In the functions [csvread](#), [csvwrite](#) the file path can be relative or absolute.

Relative paths are counted from the base directory /PALM/Programs/VisualCalc/. In this case it is enough to set the name of file only. For example, writing the name 'test.csv', the full path to a file will be /PALM/Programs/VisualCalc/test.csv.

Absolute paths are counted from a root directory and begin with a symbol '/' (the forward slash).

If your PDA has multiple volumes, you can specify the name of the volume before the file name. The volume name is determined by the media type:

- sd: - Secure Digital
- mmc: - MultiMedia Card
- ms: - Memory stick
- cf: - Compact Flash
- sm: - SmartMedia
- ram: - RAM disk

'sd:/CSV/data.csv' - a file with the name data.csv on the Secure Digital card in the directory /CSV/.

---

# Built-in functions

- [Operator precedence](#)
  - [Arithmetic operations](#)
  - [Logic operations](#)
  - [Common functions](#)
  - [Trigonometric and exponential functions](#)
  - [Complex functions](#)
  - [Statistical functions](#)
  - [Matrix functions](#)
  - [Control functions](#)
  - [Date-time functions](#)
- 

## Operator precedence

Operators	Priority	Associativity
() group { } array function	highest	=>
() array or matrix subscripts		=>
! factorial		=>
^ power		<=
+ unary - unary not logical NOT		<=
* multiplication / division		=>
+ addition - subtraction		=>
< less than > greater than <= less than or equal to >= greater than or equal to		=>
== equal to != not equal to		=>
and logical AND		=>
or logical OR		=>
= assignment	lowest	<=

Operators in the same category have equal precedence with each other. Each category has an associativity rule: ">" - left to right, "<" - right to left.

---

# Arithmetic operations

$+$  addition or unary plus  
 $-$  subtraction or unary minus  
 $*$  multiplication  
 $/$  division  
 $^$  power  
 $!$  factorial

---

$+$

$+A$   
 $A + B$

Addition or unary plus.

*arguments: real and complex numbers, arrays, matrices; for date-time, arguments can be a time point and duration or two durations*

Arguments must have the same size, unless one of them is a scalar.

*Date-time.* Adding a time point and a duration results in the time point. Adding two durations results in the duration.

```
1.3 + 6.9i  
{4.8, 16.5, -3.9} + {2.1, 9, 32}  
3.7 + {2.9, 9.1}  
date(1999, 12, 31) + day(1)  
day(1) + hour(3)
```

---

$-$

$-A$   
 $A - B$

Subtraction or unary minus.

*arguments: real and complex numbers, time points and durations, arrays, matrices*

Arguments must have the same size, unless one of them is a scalar.

*Date-time.* A difference of two time points or two durations is the duration. A difference between a time point and a duration is the time point.

```
-1.3 - 6.9i  
{4.8, 16.5, -3.9} - {2.1, 9, 32}  
3.7 - {2.9, 9.1}  
date(2000, 1, 1) - date(1999, 12, 31)  
date(1999, 12, 31) - day(4)  
-day(1) - hour(3)
```

---

\*

A \* B

Multiplication.

*arguments: real and complex numbers, arrays, matrices*

Arrays must have the same size, unless one of arguments is a scalar.

If A and B are both matrices, the number of columns of A must equal the number of rows of B. A scalar can multiply a matrix of any size.

```
-1.3 * 6.9i  
{4.8, 16.5, -3.9} * {2.1, 9, 32}  
m = matrix({{1, 2, 3, 4}, {5, 6, 7, 8}})  
n = matrix({{1, 2}, {3, 4}, {5, 6}, {7, 8}})  
m * n
```

---

/

A / B

Division.

*arguments: real and complex numbers, arrays, matrices*

Arrays must have the same size, unless one of arguments is a scalar.

The matrix can be divided by a scalar. For a matrix division use A \* [inv](#)(B).

```
-1.3 / 6.9i  
{4.8, 16.5, -3.9} / {2.1, 9, 32}  
m = matrix({{1, 2, 3, 4}, {5, 6, 7, 8}})  
m / 5.8
```

---

^

A ^ B

Power.

*arguments: real and complex numbers, arrays, matrices*

Arrays must have the same size, unless one of arguments is a scalar.

One of arguments can be a matrix. In this case, operation is made on each element of a matrix.

```
-1.3 ^ 6.9i
{4.8, 16.5, -3.9} ^ {2.1, 9, 32}
m = matrix({{1, 2, 3, 4}, {5, 6, 7, 8}})
m ^ 5.8
```

If A is negative and B is non-integer then operation defined for complex arguments only:

```
(-3) ^ 2.1 # nan
(-3 + 0i) ^ 2.1 # 9.5535 + 3.1041i
```

---

!

n!

Factorial.

*arguments: positive integers, arrays, matrices*

When n is an array or matrix, n! is the factorial for each element of n.

```
9!
{4, 5, 3}!
m = matrix({{1, 2, 3, 4}, {5, 6, 7, 8}})
m!
```

---

## Logic operations

== equal to

!= not equal to

< less than

> greater than

<= less than or equal to

>= greater than or equal to

all(a) check that all the elements are nonzero

any(a) check that any of the elements are nonzero

and logical AND

or logical OR

not logical NOT

---

==

A == B

Equal to.

*arguments: real and complex numbers, time points and durations, arrays, matrices*

If arguments are equal, result will be 1. Otherwise, it returns 0. Arrays and matrices are compared element-by-element. Arguments must have the same size, unless one of them is a scalar.

```
3 == 4 # 0
1 == (1 + 0i) # 1
{1, 2, 3} == {1, 2, 5} # {1, 1, 0}
{1, 2, 3} == 2 # {0, 1, 0}
date(2008, 10, 21) == date(2009, 10, 21) - year(1) # 1
day(1) - hour(3) == hour(21) # 1
```

See also [any](#), [all](#)

---

**!=**

A != B

Not equal to.

*arguments: real and complex numbers, time points and durations, arrays, matrices*

If arguments are not equal, result will be 1. Otherwise, it returns 0. Arrays and matrices are compared element-by-element. Arguments must have the same size, unless one of them is a scalar.

```
3 != 4 # 1
1 != (1 + 0i) # 0
{1, 2, 3} != {1, 2, 5} # {0, 0, 1}
{1, 2, 3} != 2 # {1, 0, 1}
date(2008, 10, 21) != date(2009, 10, 21) # 1
day(1) != hour(24) # 0
```

See also [any](#), [all](#)

---

**<**

A < B

Less than.

*arguments: real numbers, time points and durations, arrays, matrices*

If A is less than B result will be 1, otherwise 0. Arrays and matrices are compared element-by-element. Arguments must have the same size, unless one of them is a scalar.

```
3 < 4 # 1
5 < 5 # 0
{5, 1, 6} < {5, 3, 3} # {0, 1, 0}
```

```
{1, 2, 3} < 2 # {1, 0, 0}
date(2008, 10, 21) < date(2009, 10, 21) # 1
day(1) < hour(24) # 0
```

See also [any](#), [all](#)

---

>

A > B

Greater than.

*arguments: real numbers, time points and durations, arrays, matrices*

If A is greater than B result will be 1, otherwise 0. Arrays and matrices are compared element-by-element. Arguments must have the same size, unless one of them is a scalar.

```
-2 > -3 # 1
5 > 5 # 0
{5, 1, 6} > {5, 3, 3} # {0, 0, 1}
{1, 2, 3} > 2 # {0, 0, 1}
date(2008, 10, 21) > date(2009, 10, 21) # 0
day(1) > hour(24) # 0
```

See also [any](#), [all](#)

---

<=

A <= B

Less than or equal to.

*arguments: real numbers, time points and durations, arrays, matrices*

If A is less than or equal to B result will be 1, otherwise 0. Arrays and matrices are compared element-by-element. Arguments must have the same size, unless one of them is a scalar.

```
3 <= 4 # 1
5 <= 5 # 1
{5, 1, 6} <= {5, 3, 3} # {1, 1, 0}
{1, 2, 3} <= 2 # {1, 1, 0}
date(2008, 10, 21) <= date(2009, 10, 21) # 1
day(1) <= hour(24) # 1
```

See also [any](#), [all](#)

---



**>=**

A >= B

Greater than or equal to.

*arguments: real numbers, time points and durations, arrays, matrices*

If A is greater then or equal to B result will be 1, otherwise 0. Arrays and matrices are compared element-by-element. Arguments must have the same size, unless one of them is a scalar.

```
-2 >= -3 # 1
5 >= 5 # 1
{5, 1, 6} >= {5, 3, 3} # {1, 0, 1}
{1, 2, 3} >= 2 # {0, 1, 1}
date(2008, 10, 21) >= date(2009, 10, 21) # 0
day(1) >= hour(24) # 1
```

See also [any](#), [all](#)

---

## **all**

all(A)

Check that *all* the elements are nonzero.

*arguments: real and complex arrays, matrices; arrays of time points or durations*

all(A) returns 1 if all the elements are nonzero and returns 0 if one or more elements are zero.

```
all({1, 3, 4}) # 1
all({1, 0, 4}) # 0
all({1, 0} < {2, 4}) # 1
if (all(M < ident(3)))
  #
end
```

See also [any](#)

---

## **any**

any(A)

Check that *any* of the elements are nonzero.

*arguments: real and complex arrays, matrices; arrays of time points or durations*

`any(A)` returns 1 if any of the elements of A are nonzero, and returns 0 if all the elements are zero.

```
any({1, 3, 4}) # 1
any({1, 0, 4}) # 1
any({0, 0, 0}) # 0
any({1, 10} < {2, 4}) # 1
if (any(M < ident(3)))
  #
end
```

See also [all](#)

---

## **and**

A and B

Logical AND.

*arguments: real and complex numbers, time points and durations, arrays, matrices in any combinations*

If all elements of A are not equal to zero AND all elements of B are not equal to zero result will be 1, otherwise 0.

```
3 and 4 # 1
1 and 0 # 0
{1, 2, 3} and (ident(3) + 1) # 1
```

---

## **or**

A or B

Logical OR.

*arguments: real and complex numbers, time points and durations, arrays, matrices in any combinations*

If all elements of A are not equal to zero OR all elements of B are not equal to zero result will be 1, otherwise 0.

```
{0, 1} or 0 # 0
1 or 0 # 1
{1, 2, 3} or ident(3) # 1
```

---

## **not**

not A

Logical NOT.

*arguments: real and complex numbers, time points and durations, arrays, matrices*

If all elements of A are not equal to zero result will be 0, otherwise 1.

```
not 0 # 1
not {1, 2, 3} # 0
not ident(3) # 1
```

---

## Common functions

[sqrt\(x\)](#) - square root

[round\(x\)](#) - round to nearest integer

[floor\(x\)](#) - round towards minus infinity

[ceil\(x\)](#) - round towards infinity

[abs\(x\)](#) - absolute value (module)

[mod\(x, y\)](#) - remainder of division x on y

[hypot\(x, y\)](#) - hypotenuse of a right triangle

[deg\(x\)](#) - convert radians to degrees

[rad\(x\)](#) - convert degrees to radians

[gcd\(x, y\)](#) - greatest common divisor

[lcm\(x, y\)](#) - least common multiple

[rat\(x\)](#) - rational fraction approximation

[rnd\(\)](#) - random numbers uniformly distributed in the interval (0, 1)

[rndn\(\)](#) - normally distributed random numbers

[seed\(n\)](#) - initialization of rnd()

[seedn\(n\)](#) - initialization of rndn()

[df\(x, f\(x\)\)](#) the derivative of function f(x) at the point x

[df\(x, f\(x\), h\)](#) the derivative of function f(x) at the point x with the initial step h

[quad\(x, f\(x\), a, b\)](#) numerically evaluate integral, adaptive Gauss-Kronrod quadrature

[quadr\(x, f\(x\), a, b\)](#) numerically evaluate integral, Romberg quadrature

[polint\(x, t, p\)](#) polynomial interpolation

[polint\(x, p\)](#) polynomial interpolation with default step

[fzero\(x, f\(x\), a, b\)](#) - find a zero of a function f(x) lying between a and b

[fmin\(x, f\(x\), a, b\)](#) - find the local minimum of a function f(x) within the interval (a, b)

---

## sqrt

`sqrt(A)`

Square root.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
sqrt(4)
sqrt({4, 9, 25})
sqrt(vector({8, 10}))
```

The square root of a negative number is defined for complex numbers only:

```
sqrt(-1) # nan
sqrt(-1 + 0i) # i
```

---

## **round**

```
round(A)
```

Round to nearest integer.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
round(-4.6 + 8.9i)
round({4.5, 9.0, -3.3})
round(vector({0.8, 5.4}))
```

See also [floor](#), [ceil](#)

---

## **floor**

```
floor(A)
```

Round towards minus infinity.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
floor(-4.6 + 8.9i)
floor({4.5, 9.0, -3.3})
floor(vector({0.8, 5.4}))
```

See also [round](#), [ceil](#)

---

## **ceil**

```
ceil(A)
```

Round towards infinity.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
ceil(-4.6 + 8.9i)
ceil({4.5, 9.0, -3.3})
ceil(vector({0.8, 5.4}))
```

See also [round](#), [floor](#)

---

## **abs**

```
abs(A)
```

Absolute value (module).

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

For complex numbers absolute value is calculated as

```
sqrt(real(A) ^ 2 + imag(A) ^ 2)
abs(-3)
abs(3 + 4i)
abs({4.5, -9.0, -3.3i})
```

See also [arg](#), [norm](#)

---

## **mod**

```
mod(A, B)
```

Remainder of division A on B.

*arguments: real numbers, arrays*

Arrays must have the same size, unless one of arguments is a scalar.

```
mod(5, 3)
mod({9, 15}, 4)
mod({9, 15, 8}, {4, 3, 5})
```

---

## **hypot**

```
hypot(A, B)
```

Hypotenuse of a right triangle with sides A and B.

*arguments: real numbers, arrays*

Arrays must have the same size.

hypot(A, B) uses the formula:

```
sqrt(A ^ 2 + B ^ 2)
hypot(3, 4)
hypot({3, 15, 8}, {4, 3, 5})
```

---

## deg

deg(A)

Convert radians to degrees.

*arguments: real numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

deg(A) uses the formula:

```
A * 180 / pi
deg(pi)
deg({pi / 3, asin(1)})
```

See also [rad](#)

---

## rad

rad(A)

Convert degrees to radians.

*arguments: real numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

rad(A) uses the formula:

```
A * pi / 180
sin(rad(45))
rad({180, -90})
```

See also [deg](#)

---

## gcd

`gcd(A, B)`

Greatest common divisor of A and B.

*arguments: integer numbers, arrays, matrices*

Arguments must have the same size, unless one of them is a scalar.

```
gcd(24, 36)
gcd({24, 12, 30, 36, 0}, 36)
```

See also [lcm](#)

---

## lcm

`lcm(A, B)`

Least common multiple of A and B.

*arguments: integer numbers, arrays, matrices*

Arguments must have the same size, unless one of them is a scalar.

```
lcm(24, 36)
lcm({24, 12, 30, 36, 0}, 36)
```

See also [gcd](#)

---

## rat

`rat(A)`

Rational fraction approximation.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

`rat(A)` returns array {N, D} so that  $N / D = A$ . Accuracy of approximation is 1e-6.

```
rat(3 / 5)
rat(pi)
rat(0.6 + 0.7i)
```

```
rat(ident(2) / 5)
```

See also [frac](#)

---

## **rnd**

```
rnd()
```

Random numbers uniformly distributed in the interval (0, 1).

*arguments: none*

```
x = rnd()
```

See also [seed](#), [rndn](#)

---

## **rndn**

```
rndn()
```

Normally distributed random numbers.

*arguments: none*

```
x = rndn()
```

See also [seedn](#), [rnd](#)

---

## **seed**

```
seed(N)
```

Initialization of rnd().

*arguments: non-negative integers*

Returns a current state of the random generator.

If  $N = 0$ , then function returns the state without changing the generator.

```
s = seed(0)
a1 = rnd()
b = rnd()
seed(s)
a2 = rnd() # a2 == a1
```



See also [rnd](#), [seedn](#)

---

## **seedn**

```
seedn(N)
```

Initialization of rndn().

*arguments: non-negative integers*

Returns a current state of the normal generator.

If  $N = 0$ , then function returns the state without changing the generator.

```
s = seedn(0)
a1 = rndn()
b = rndn()
seedn(s)
a2 = rndn() # a2 == a1
```

See also [rndn](#), [seed](#)

---

## **df**

```
df(X, F(X))
df(X, F(X), H)
```

df(X, F(X)) - the derivative of function F(X) at the point X.

df(X, F(X), H) - the derivative of function F(X) at the point X with the initial step H.

*arguments: X - real variable; F(X) - real valued function; H - initial step of algorithm (real number)*

If step H is not set, value 0.3 used by default.

```
x = 2
df(x, x ^ 3) # 12
```

F(X) can be a user defined function:

```
func f(x)
    ret(x ^ 2 + sin(x))
end
x = 2
df(x, f(x))
```

---

## quad

```
quad(X, F(X), A, B)
```

Numerically evaluate integral, adaptive Gauss-Kronrod quadrature

*arguments: X - real variable; F(X) - real valued function; A, B - real numbers*

$$\int_a^b f(x) dx$$

```
x = 0  
quad(x, sin(x), 0, pi / 2) # 1
```

quad allows calculating the integrals having singularities at limits:

```
# singularity at x = 0  
x = 0  
quad(x, 1 / sqrt(x) * exp(- x ^ 2), 0, 1) # 1.689677
```

F(X) can be a user defined function:

```
func f(x, a)  
    a(1) = a(1) + 1  
    ret(e ^ x)  
end  
x = 0, n = {0}  
q = quad(x, f(x, n), 0, 1)  
steps = n(1) # the number of function evaluations
```

See also [quadr](#)

---

## quadr

```
quadr(X, F(X), A, B)
```

Numerically evaluate integral, Romberg quadrature

*arguments: X - real variable; F(X) - real valued function; A, B - real numbers*

$$\int_a^b f(x) dx$$

```
x = 0  
quadr(x, sin(x), 0, pi / 2) # 1
```

F(X) can be a user defined function:

```
func f(x, a)
```

```

    a(1) = a(1) + 1
    ret(e ^ x)
end
x = 0, n = {0}
q = quadr(x, f(x, n), 0, 1)
steps = n(1) # the number of function evaluations

```

See also [quad](#)

---

## polint

```

polint(X, T, P)
polint(X, P)

```

polint(X, T, P) - polynomial interpolation.

polint(X, P) - polynomial interpolation with default step.

*arguments: X - real number, array or matrix; T, P - real arrays*

Arrays T and P must have the same size.

Returns the value in point X of the polynom, which is passing through set of points. P - array of base points, T - array of x-coordinates of base points. If array T is not set, it is considered, that base points are located with step 1.

```

# the straight line passing through points [1, 3] and [2, 5]
p = {3, 5}
polint({4, -2, 7}, p) # {9, -3, 15}
# the parabola passing through points [-2, 4], [0, 0] and [2, 4]
p = {4, 0, 4}
t = {-2, 0, 2}
polint(3, t, p) # 9

```

---

## fzero

```

fzero(X, F(X), A, B)

```

Find a zero of a function F(X) lying between A and B.

*arguments: X - real variable; F(X) - real valued function; A, B - real numbers*

The sign of F(A) must differ from the sign of F(B).

```

x = 0
fzero(x, cos(x), 0, pi) # pi/2

```

F(X) can be a user defined function:

```

func f(x)

```

```

    v = sin(x) - cos(x)
    ret(v)
end
fzero(x = 0, f(x), 0, 2) # pi/4

```

See also [fmin](#)

---

## fmin

```
fmin(X, F(X), A, B)
```

Find the local minimum of a function  $F(X)$  within the interval  $(A, B)$ .

*arguments:  $X$  - real variable;  $F(X)$  - real valued function;  $A, B$  - real numbers*

```

x = 0
fmin(x, sin(x), -pi, pi) # -pi/2

```

$F(X)$  can be a user defined function:

```

func f(x, a)
    ret((x - a) ^ 2)
end
x = 0, t = 0.5
fmin(x, f(x, t), 0, 1) # 0.5

```

See also [fzero](#)

---

## Trigonometric and exponential functions

[sin\(x\)](#) - sine

[cos\(x\)](#) - cosine

[tan\(x\)](#) - tangent

[asin\(x\)](#) - arcsine

[acos\(x\)](#) - arccosine

[atan\(x\)](#) - arctangent

[atan\(y, x\)](#) - arctangent of  $y / x$

[sinh\(x\)](#) - hyperbolic sine

[cosh\(x\)](#) - hyperbolic cosine

[tanh\(x\)](#) - hyperbolic tangent

[asinh\(x\)](#) - inverse hyperbolic sine

[acosh\(x\)](#) - inverse hyperbolic cosine

[atanh\(x\)](#) - inverse hyperbolic tangent

[sinc\(x\)](#) - sinc,  $\sin(x) / x$

[exp\(x\)](#) - exponential  $e$  to the  $x$

[ln\(x\)](#) - natural logarithm

[lg\(x\)](#) - base 10 logarithm

---

## **sin**

`sin(A)`

Sine, argument in radians.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
sin(pi / 4)
sin(3 + 2i)
sin({0, rad(45), pi / 3})
```

See also [asin](#), [sinh](#)

---

## **cos**

`cos(A)`

Cosine, argument in radians.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
cos(pi / 4)
cos(3 + 2i)
cos({0, rad(45), pi / 3})
```

See also [acos](#), [cosh](#)

---

## **tan**

`tan(A)`

Tangent, argument in radians.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
tan(pi / 4)
tan(3 + 2i)
tan({0, rad(45), pi / 3})
```

See also [atan](#), [tanh](#)

---

## **asin**

`asin(A)`

Arcsine, result in radians.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

For real argument in the domain  $[-1, 1]$  result is in the range  $[-\pi/2, \pi/2]$ . Outside of this domain the result is defined for complex numbers only.

```
asin(-1)
asin(2 + 0i)
asin({0, 0.5, 1})
```

See also [sin](#), [sinh](#)

---

## **acos**

`acos(A)`

Arccosine, result in radians.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

For real argument in the domain  $[-1, 1]$  result is in the range  $[\pi, 0]$ . Outside of this domain the result is defined for complex numbers only.

```
acos(-1)
acos(2 + 0i)
acos({0, 0.5, 1})
```

See also [cos](#), [cosh](#)

---

## **atan**

`atan(A)`

Arctangent, result in radians.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

For real argument result is in the range  $[-\pi / 2, \pi / 2]$ .

```
atan(-1)
atan(20i)
atan({0, 5, 15})
```

See also [atan\(y, x\)](#), [tan](#), [tanh](#)

---

## atan

```
atan(Y, X)
```

Arctangent of  $Y / X$ , result in radians.

*arguments: real numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

atan(Y, X) returns a value in the range  $[-\pi, \pi]$ .

```
atan(-1, -1)
atan({0, 5, -15}, {-1, 2, 4})
```

See also [atan\(x\)](#), [tan](#), [tanh](#)

---

## sinh

```
sinh(A)
```

Hyperbolic sine, argument in radians.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

sinh(z) uses the formula:

```
sinh(z) = (exp(z) - exp(-z)) / 2
sinh(pi)
sinh(3 + 2i)
sinh({0, rad(45), pi / 3})
```

See also [sin](#), [asin](#)

---

## cosh

`cosh(A)`

Hyperbolic cosine, argument in radians.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

`cosh(z)` uses the formula:

```
        cosh(z) = (exp(z) + exp(-z)) / 2
cosh(pi)
cosh(3 + 2i)
cosh({0, rad(45), pi / 3})
```

See also [cos](#), [acos](#)

---

## tanh

`tanh(A)`

Hyperbolic tangent, argument in radians.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

`tanh(z)` uses the formula:

```
        tanh(z) = sinh(z) / cosh(z)
tanh(pi)
tanh(3 + 2i)
tanh({0, rad(45), pi / 3})
```

See also [tan](#), [atan](#)

---

## asinh

`asinh(A)`

Inverse hyperbolic sine, result in radians.

*arguments: real and complex numbers, arrays, matrices*



For arrays and matrices value of function is calculated for each element.

```
asinh(11.548)  
asinh(-4.17 + 9.15i)
```

See also [sinh](#)

---

## **acosh**

```
acosh(A)
```

Inverse hyperbolic cosine, result in radians.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
acosh(11.59)  
acosh(-4.19 + 9.11i)
```

See also [cosh](#)

---

## **atanh**

```
atanh(A)
```

Inverse hyperbolic tangent, result in radians.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
atanh(0.9963)  
atanh(1 + 0.5i)
```

See also [tanh](#)

---

## **sinc**

```
sinc(A)
```

Sinc function,  $\sin(A) / A$ , argument in radians.

*arguments: real numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

If  $A = 0$  then  $\text{sinc}(0) = 1$ .

```
sinc(pi / 4)
sinc({0, rad(45), pi / 3})
```

---

## **exp**

`exp(A)`

Exponential e to the A.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
exp(1)
exp(3 + 2i)
exp({0, 4, i * pi / 3})
```

See also [ln](#), [lg](#)

---

## **ln**

`ln(A)`

Natural logarithm.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
ln(1)
ln(-1 + 0i)
ln({2, e, 10})
```

See also [exp](#), [lg](#)

---

## **lg**

`lg(A)`

Base 10 logarithm.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
lg(1000)
lg(10i)
lg({2, e, 10})
```

See also [exp](#), [ln](#)

---

## Complex functions

[real\(z\)](#) - real part of complex number

[imag\(z\)](#) - imaginary part of complex number

[abs\(z\)](#) - absolute value (module)

[arg\(z\)](#) - argument of complex number

[norm\(z\)](#) - norm of complex number (a square of the module)

[conj\(z\)](#) - complex conjugate

[polar\(x, y\)](#) - complex number with the module x and argument y

[sign\(z\)](#) - signum function

---

### real

```
real(A)
```

Real part of complex number.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
real(2 + 3i)
real({0, i, 10 - 5i})
```

See also [imag](#)

---

### imag

```
imag(A)
```

Imaginary part of complex number.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
imag(2 + 3i)
imag({0, i, 10 - 5i})
```

See also [real](#)

---

## **arg**

```
arg(A)
```

Argument of complex number.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
deg(arg(2 + 2i))
arg({0, i, 10 - 5i})
```

See also [abs](#), [norm](#)

---

## **norm**

```
norm(A)
```

Norm of complex number (a square of the module).

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
norm(2 + 2i)
norm({0, i, 10 - 5i})
```

See also [abs](#), [arg](#)

---

## **conj**

```
conj(A)
```

Complex conjugate.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

```
conj(2 + 2i)
conj({3, i, 10 - 5i})
```

See also [imag](#), [real](#)

---

## polar

```
polar(A, B)
```

Complex number with the module A and argument B.

*arguments: real numbers, arrays*

Arrays must have the same size.

```
polar(5, rad(45))
polar({9, 15, 8}, {0, pi / 2, pi})
```

---

## sign

```
sign(A)
```

Signum function.

*arguments: real and complex numbers, arrays, matrices*

For arrays and matrices value of function is calculated for each element.

For real numbers function returns

- 1, if A is greater than 0
- 0, if A is equal 0
- -1, if A is less than 0.

For nonzero complex z,  $\text{sign}(z) = z / \text{abs}(z)$ .

```
sign(2 + 2i)
sign({-3, 0, 3, 3i, -3i, 10 - 5i})
```

See also [abs](#)

---

## Statistical functions

[zero\(n\)](#) - create an array of n zeros

[seq\(from, to\)](#), [seq\(from, step, to\)](#) - arithmetic sequence

[rep\(v, n\)](#) - replicate a value multiple times  
[min\(a\)](#) - smallest element  
[max\(a\)](#) - largest element  
[find\(a\)](#) - find indices of nonzero elements  
[sum\(a\)](#) - sum of elements  
[prod\(a\)](#) - product of elements  
[sort\(a\)](#) - sort array elements in ascending order  
[rev\(a\)](#) - inverse order of elements  
[mean\(a\)](#) - average value of elements  
[var\(a\)](#) - variance (the square of the standard deviation), biased estimate  
[sdev\(a\)](#) - standard deviation, biased estimate  
[med\(a\)](#) - median  
[corr\(a, b\)](#) - correlation of two arrays  
[range\(a, from, size\)](#) - returns size elements of an array starting with from  
[size\(a\)](#) - size of an array  
[join\(a, b\)](#) - join two arrays  
[merge\(a\)](#) - merge elements of the nested arrays  
[csvread\(name\)](#) - read a CSV-file  
[csvwrite\(a, name\)](#) - write a CSV-file  
[csvwrite\(a, name, opt\)](#) - write a CSV-file using additional options

---

## zero

`zero(A)`

`zero(A, B)`

`zero(A)` - create an array of A zeros.

`zero(A, B)` - create A-by-B zero matrix.

*arguments: real and complex numbers*

For real numbers function returns a real array (matrix), for complex numbers returns a complex array (matrix)

```
zero(3) # array with 3 elements {0, 0, 0}
zero(3i) # complex array with 3 elements {0+0i, 0+0i, 0+0i}
zero(2, 2) # matrix 2x2, matrix({{0, 0}, {0, 0}})
zero(2i, 2i) # complex matrix 2x2, matrix({{0+0i, 0+0i}, {0+0i, 0+0i}})
```

See also [seq](#), [rep](#), [ident](#), [diag](#)

---

## seq

`seq(From, To)`

```
seq(From, Step, To)
```

Arithmetic sequence of numbers in range  $[From, To]$  with step *Step*

*arguments: real numbers*

If step *Step* is not set

- if  $From < To$ , than  $Step = 1$ ;
- if  $From > To$ , than  $Step = -1$ .

```
seq(1, 5) # {1, 2, 3, 4, 5}
seq(5, 1) # {5, 4, 3, 2, 1}
seq(0, 0.2, 0.6) # {0, 0.2, 0.4, 0.6}
seq(10, -2, 4) # {10, 8, 6, 4}
```

See also [zero](#), [rep](#)

---

## rep

```
rep(V, N)
```

```
rep(V, M, N)
```

`rep(V, N)` - create an array of  $N$  identical elements  $V$ .

`rep(V, M, N)` - create an  $M$ -by- $N$  matrix of identical elements  $V$ .

*arguments: V - real and complex numbers, time points and durations, arrays, matrices; M, N - nonnegative integers*

```
rep(1, 4)      # array of four ones {1, 1, 1, 1}
rep(2+3i, 3)   # complex array of 3 elements {2+3i, 2+3i, 2+3i}
rep({9, -2, 7}, 2) # array of two arrays {{9, -2, 7}, {9, -2, 7}}
rep(date(2000, 11, 30), 3) # array of 3 dates
rep(1, 2, 2)   # 2-by-2 matrix, matrix({{1, 1}, {1, 1}})
rep(2i, 2, 2)  # 2-by-2 complex matrix, matrix({{2i, 2i}, {2i, 2i}})
```

See also [zero](#), [seq](#), [ident](#), [diag](#)

---

## min

```
min(A)
```

Smallest element.

*arguments: real arrays and matrices; arrays of time points or durations*

```
min({2, 3, 1, 5}) # 1
min({{2, 3, 1}, {8, 5}}) # {1, 5}
min(ident(3)) # 0
min({date(2009, 5, 16), date(1900, 1, 1)}) # 1900-01-01
min({hour(72), day(2)}) # +0000+00 +02
```

See also [max](#), [mean](#), [med](#)

---

## max

max(A)

Largest element.

*arguments: real arrays and matrices; arrays of time points or durations*

```
max({2, 3, 1, 5}) # 5
max({{2, 3, 1}, {8, 5}}) # {3, 8}
max(ident(3)) # 1
max({date(2009, 5, 16), date(1900, 1, 1)}) # 2009-05-16
max({hour(72), day(2)}) # +0000+00 +03
```

See also [min](#), [mean](#), [med](#)

---

## find

find(A)

Find indices of nonzero elements.

*arguments: arrays and matrices*

```
x = {2, 0, 1, 5}
find(x) # {1, 3, 4}
find(x >= 2) # {1, 4}
find({{2, 0, 1}, {8, 5}}) # {{1, 3}, {1, 2}}
find(ident(3)) # {{1, 1}, {2, 2}, {3, 3}}
a = {date(2009, 5, 16), date(1900, 1, 1)}
find(a > date(2000, 1, 1)) # {1}
```

See also [min](#), [max](#)

---

## sum

sum(A)

Sum of elements.



*arguments: real and complex arrays, matrices*

```
sum({2+3i, 3, 1-7i, 5}) # 11-4i
sum({{2, 3, 1}, {8, 5}}) # {6, 13}
sum(ident(3)) # 3
```

See also [prod](#), [tr](#)

---

## **prod**

```
prod(A)
```

Product of elements.

*arguments: real and complex arrays, matrices*

```
prod({2+3i, 3, 1-7i, 5}) # 345-165i
prod({{2, 3, 1}, {8, 5}}) # {6, 40}
prod(ident(3)) # 0
```

See also [sum](#), [tr](#)

---

## **sort**

```
sort(A)
```

Sort array elements in ascending order.

*arguments: real arrays and matrices; arrays of time points or durations*

```
sort({2, 3, 1, 5}) # {1, 2, 3, 5}
sort({{2, 3, 1}, {8, 5}}) # {{1, 2, 3}, {5, 8}}
sort(ident(3)) # matrix({{0, 0, 0}, {0, 0, 0}, {1, 1, 1}})
sort({date(2009, 5, 16), date(2008, 3, 22), date(2009, 2, 3)})
```

See also [rev](#)

---

## **rev**

```
rev(A)
```

Inverse order of elements.

*arguments: real and complex arrays, matrices; arrays of time points or durations*

```
rev({2+3i, 3, 1-7i, 5}) # {5, 1-7i, 3, 2+3i}
```

```
rev(sort({2, 3, 1, 5})) # sorting in descending order, {5, 3, 2, 1}
rev({{2, 3, 1}, {8, 5}}) # {{8, 5}, {2, 3, 1}}
rev(sort(ident(3))) # matrix({{1, 1, 1}, {0, 0, 0}, {0, 0, 0}})
```

See also [sort](#)

---

## mean

```
mean(A)
```

Average value of elements.

*arguments: real and complex arrays, matrices*

```
mean({2+3i, 3, 1-7i, 5}) # 2.75 - i
mean({{2, 3, 1}, {8, 5}}) # {2, 6.5}
mean(ident(3)) # 0.3333
```

See also [min](#), [max](#), [med](#)

---

## var

```
var(A)
```

Variance (the square of the standard deviation), biased estimate.

*arguments: real and complex arrays, matrices*

To calculate unbiased estimate use:

- $\text{var}(A) / (1 - 1 / \text{size}(A))$ , if A is an array
- $\text{var}(A) / (1 - 1 / \text{prod}(\text{size}(A)))$ , if A is a matrix

```
var({2+3i, 3, 1-7i, 5})
var({{2, 3, 1}, {8, 5}})
var(ident(3))
```

See also [sdev](#), [corr](#)

---

## sdev

```
sdev(A)
```

Standard deviation, biased estimate.

*arguments: real and complex arrays, matrices*

To calculate unbiased estimate use:

- `sdev(A) / sqrt(1 - 1 / size(A))`, if A is an array
- `sdev(A) / sqrt(1 - 1 / prod(size(A)))`, if A is a matrix

```
sdev({2+3i, 3, 1-7i, 5})  
sdev({{2, 3, 1}, {8, 5}})  
sdev(ident(3))
```

See also [var](#), [corr](#)

---

## med

```
med(A)
```

Median.

*arguments: real arrays and matrices*

```
med({2, 3, 1, 5}) # 2.5  
med({{2, 3, 1}, {8, 5}}) # {2, 6.5}  
med(ident(3)) # 0
```

See also [min](#), [max](#), [mean](#)

---

## corr

```
corr(A, B)
```

Correlation of two arrays.

*arguments: real and complex arrays, matrices*

Arguments must have the same size.

```
corr({1, 2, 3, 4}, {4, 3, 2, 1})  
corr({8-1i, 2, 3i}, {4, 3, 6i})  
corr(ident(3), ident(3))
```

See also [var](#), [sdev](#)

---

## range

```
range(A, From, Size)
```

Returns Size elements of an array A starting with From.

*arguments: A - real or complex array, array of time points or durations; From, Size - real numbers*

Values of From and Size round to the nearest integers. Numbering of elements begins with 1 or 0 (it is set in [Expression setup](#)).

- If Size > 0 then range(A, From, Size) extracts elements with indexes From, From+1, From+2 ... From+Size-1.
- If Size < 0 then range(A, From, Size) extracts elements in reverse order: From, From-1, From-2 ... From+Size+1.

```
a = {1, 2, 3, 4, 5}
range(a, 3, 3) # {3, 4, 5}
range(a, 3, -3) # {3, 2, 1}
```

See also [size](#), [join](#)

---

## size

```
size(A)
```

Size of an array or matrix.

*arguments: real and complex arrays, matrices; arrays of time points or durations*

If A is a matrix, size(A) returns an array with two elements: {number\_of\_rows, number\_of\_columns}.

```
size({2, 3, 1, 5}) # 4
size({{2, 3, 1}, {8, 5}}) # 2
size(ident(3)) # {3, 3}
```

See also [range](#), [block](#), [join](#), [joinh](#), [joinv](#)

---

## join

```
join(A, B)
```

Join two arrays.

*arguments: real and complex arrays; arrays of time points or durations*

```
join({1, 2, 3}, {4, 5}) # {1, 2, 3, 4, 5}
join({2-i, 3i}, {4, 3}) # {2-i, 3i, 4, 3}
```

See also [range](#)

---

## merge

`merge(A)`

Merge elements of the nested arrays.

*arguments: real and complex multidimensional arrays; arrays of time points or durations*

```
merge({{1.1, 1.2}, {2.1, 2.2}}) # {1.1, 1.2, 2.1, 2.2}
a = {1, 2, 3}
v = vector(a) # matrix 3x1
a2 = array(v) # array {{1}, {2}, {3}}
merge(a2) # original array {1, 2, 3}
```

---

## csvread

`csvread(Name)`

Read a CSV-file into an array.

*arguments: Name - a string representing a name of a file*

```
data = csvread('data.csv')
```

See also [csvwrite](#), [Reading and writing CSV-files](#)

---

## csvwrite

```
csvwrite(A, Name)
csvwrite(A, Name, Options)
```

`csvwrite(A, Name)` - write array A into a CSV-file.

`csvwrite(A, Name, Options)` - write array A into a CSV-file using additional Options.

*arguments: A - one- or two-dimensional array of real numbers; Name - a string representing a name of a file; Options - a string of additional options*

`csvwrite` returns the number of lines written to the file.

```
a = {{1, -3}, {2.8, 5}}
csvwrite(a, 'data.csv')
```

content of data.csv:

```
1, -3
2.8, 5
```

```
b = {1, -3}
csvwrite(b, 'data2.csv')
```

```
content of data2.csv:
1, -3
```

Additional Options:

Option	Parameter	Description
-a		Append - append the data to the file
-o		Overwrite - overwrite any existing data in the file
-s		Silent - don't ask for confirmation to overwrite the file. If the file exists, calculations are terminated with the error message
-p	Number from 1 to 15	Precision - numeric precision to use in writing data to the file
-n	r - CR n - LF	NewLine - a symbol of the end of a line. By default, CR/LF is used

Options and additional parameters are separated by spaces.

```
c = {1.23456789, -3.5}
csvwrite(c, 'data3.csv', '-o -p 3') # overwrite the file, precision -
3 significant digits
```

```
contents of data3.csv:
1.23, -3.5
```

See also [csvread](#), [Reading and writing CSV-files](#)

---

## Matrix functions

[matrix\(a\)](#) - convert a two-dimensional array to a matrix  
[vector\(a\)](#) - convert an array to a matrix with one column  
[ident\(n\)](#) - n-by-n identity matrix  
[diag\(a\)](#) - diagonal matrix  
[zero\(n, m\)](#) - n-by-m zero matrix  
[rep\(v, m, n\)](#) - create a matrix of identical elements  
[inv\(m\)](#) - matrix inverse  
[det\(m\)](#) - matrix determinant  
[trans\(m\)](#) - matrix transpose  
[solve\(A, B\)](#) - solution of set of linear equations  $Ax = B$   
[tr\(m\)](#) - sum of diagonal elements  
[eig\(m\)](#) - eigenvalues and eigenvectors of a symmetric matrix  
[block\(m, row, col, n\\_rows, n\\_cols\)](#) - return the block of matrix elements  
[array\(m\)](#) - transform a matrix into a two-dimensional array  
[size\(m\)](#) - size of a matrix

[row\(m, n\)](#) - row of a matrix

[col\(m, n\)](#) - column of a matrix

[joinh\(m1, m2\)](#) - join two matrices horizontally

[joinv\(m1, m2\)](#) - join two matrices vertically

---

## **matrix**

`matrix(A)`

convert a two-dimensional array to a matrix.

*arguments: real and complex multidimensional arrays*

Each element of an array represents a row of a matrix.

```
matrix({{1.1, 1.2}, {2.1, 2.2}}) # matrix 2x2
matrix({zero(3i), {1, 2, 3}}) # matrix 2x3
```

See also [vector](#), [ident](#), [diag](#), [zero](#)

---

## **vector**

`vector(A)`

Convert an array to a matrix with one column.

*arguments: real and complex arrays*

Each element of an array represents a row of a matrix.

```
vector({1, 2, 3}) # matrix 3x1
vector({1-2i, 7+8i, i, 0}) # complex matrix 4x1
```

See also [matrix](#), [ident](#), [diag](#), [zero](#)

---

## **ident**

`ident(N)`

N-by-N identity matrix.

*arguments: real and complex numbers*

`ident(N)` returns an N-by-N matrix with 1's on the main diagonal and 0's elsewhere.

For real N function returns a real matrix, for complex N returns a complex matrix.

```
ident(2) # matrix 2x2, matrix({{1, 0}, {0, 1}})
ident(2i) # complex matrix 2x2, matrix({{1+0i, 0+0i}, {0+0i, 1+0i}})
```

See also [matrix](#), [vector](#), [diag](#), [zero](#)

---

## diag

```
diag(A)
```

Diagonal matrix.

*arguments: real and complex arrays*

Returns a square matrix of order size(A), with the elements of A on the main diagonal.

```
diag({1, 2, 3}) # matrix 3x3, matrix({{1, 0, 0}, {0, 2, 0}, {0, 0, 3}})
diag({2i, 10-i}) # complex matrix 2x2, matrix({{2i, 0+0i}, {0+0i, 10-i}})
```

See also [matrix](#), [vector](#), [ident](#), [zero](#)

---

## inv

```
inv(M)
```

Matrix inverse.

*arguments: real and complex matrices, arrays of matrices*

The matrix should be square and nonsingular.

To solve the system of linear equations  $Ax = B$  it is better to do this with function [solve](#).

```
m = matrix({{3, 8}, {4, 3}})
mi = inv(m) # matrix inverse
m * mi # product of matrix and its inverse is identity matrix
inv(matrix({{3i, 8-2i}, {4, -3i}})) # inverse of a complex matrix
```

See also [solve](#), [trans](#), [det](#)

---

## det



`det(M)`

Matrix determinant.

*arguments: real and complex matrices, arrays of matrices*

The matrix should be square.

```
det(matrix({{3, 8}, {4, 3}}))  
det(matrix({{3i, 8-2i}, {4, -3i}}))
```

See also [inv](#), [solve](#), [trans](#)

---

## **trans**

`trans(M)`

Matrix transpose.

*arguments: real and complex matrices, arrays of matrices*

```
trans(matrix({{1, 2, 3}, {4, 5, 6}}))  
trans(matrix({{i, -2i}}))
```

See also [inv](#), [solve](#), [det](#)

---

## **solve**

`solve(A, B)`

Solution of set of linear equations  $A \cdot X = B$ .

*arguments: real and complex matrices, arrays of matrices*

A and B must be matrices that have the same number of rows. The matrix A should be square and nonsingular. Matrix B may consist of several columns. In this case columns of result consists of solutions  $A \cdot X_n = B_n$ , where  $B_n$  - column of matrix B.

```
A = matrix({{2, 3, 7}, {3, 5, 9}, {4, 7, 6}})  
B = vector({-1, 2, 1})  
solve(A, B)  
A = matrix({{2, 3, 7}, {3, 5, 9}, {4, 7, 6}})  
B = matrix({{-1, 3}, {2, 5}, {1, 4}})  
solve(A, B)
```

See also [inv](#), [trans](#), [det](#)

---

## tr

```
tr(M)
```

Sum of diagonal elements.

*arguments: real and complex matrices, arrays of matrices*

```
tr(matrix({{1, 2, 3}, {4, 5i, 6}})) # 1+5i  
tr(ident(3)) # 3
```

See also [sum](#), [prod](#)

---

## eig

```
eig(M)
```

Eigenvalues and eigenvectors of a symmetric matrix.

*arguments: real symmetric matrices, arrays of matrices*

eig(M) returns an array with two elements {matrix\_of\_eigenvectors, matrix\_of\_eigenvalues}:

$$\left( \begin{pmatrix} v_{11} & v_{21} & v_{31} \\ v_{12} & v_{22} & v_{32} \\ v_{13} & v_{23} & v_{33} \end{pmatrix}, \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix} \right)$$

```
m = matrix({{1, 2, 3}, {2, 5, 6}, {3, 6, 0}})  
vd = eig(m)  
v = vd(1) # matrix of eigenvectors  
d = vd(2) # matrix of eigenvalues
```

---

## block

```
block(M, Row, Col, NRows, NCols)
```

Returns a submatrix that consists of NRows rows and NCols columns, starting with Row and Col.

*arguments: M - real or complex matrix; Row, Col, NRows, NCols - real numbers*

Values of Row, Col, NRows, NCols round to the nearest integers. Numbering of rows and columns begins with 1 or 0 (it is set in [Expression setup](#)).

- If NRows > 0 then function extracts elements from rows Row, Row+1, Row+2 ... Row+NRows-1.

- If `NRows < 0` then function extracts rows in reverse order: `Row`, `Row-1`, `Row-2` ... `Row+NRows+1`.
- If `NCols > 0` then function extracts elements from columns `Col`, `Col+1`, `Col+2` ... `Col+NCols-1`.
- If `NCols < 0` then function extracts columns in reverse order: `Col`, `Col-1`, `Col-2` ... `Col+NCols+1`.

```
m = matrix({{1, 2, 3}, {4, 5, 6}, {7, 8, 9}})
block(m, 2, 2, 2, 2) # matrix({{5, 6}, {8, 9}})
block(m, 2, 2, -2, 2) # matrix({{5, 6}, {2, 3}})
block(m, 2, 2, 2, -2) # matrix({{5, 4}, {8, 7}})
block(m, 2, 2, -2, -2) # matrix({{5, 4}, {2, 1}})
```

See also [size](#), [joinh](#), [joinv](#)

---

## array

`array(M)`

Transform a matrix into a two-dimensional array.

*arguments: real and complex matrices*

The array which each element represents a row of matrix `M` is created.

```
array(ident(2)) # {{1, 0}, {0, 1}}
```

See also [row](#), [col](#)

---

## row

`row(M, N)`

Row `N` of a matrix `M`.

*arguments: real and complex matrices*

The array which elements represent row `N` of matrix `A` is created. Numbering of rows begins with 1 or 0 (it is set in [Expression setup](#)).

```
row(ident(3), 2) # {0, 1, 0}
```

See also [col](#), [array](#)

---

## col

```
col(M, N)
```

Column N of a matrix M.

*arguments: real and complex matrices*

The array which elements represent column N of matrix A is created. Numbering of columns begins with 1 or 0 (it is set in [Expression setup](#)).

```
col(ident(3), 3) # {0, 0, 1}
```

See also [row](#), [array](#)

---

## joinh

```
joinh(A, B)
```

Join two matrices horizontally.

*arguments: real and complex matrices, arrays of matrices*

A and B must be matrices that have the same number of rows. Elements of matrix B are added on the right to elements of matrix A.

```
A = matrix({{1, 2, 3}, {4, 5, 6}, {7, 8, 9}})
B = vector({10, 11, 12})
joinh(A, B)
A = matrix({{1, 2, 3}, {4, 5, 6}, {7, 8, 9}})
B = ident(3)
joinh(A, B)
```

See also [joinv](#), [block](#), [size](#)

---

## joinv

```
joinv(A, B)
```

Join two matrices vertically.

*arguments: real and complex matrices, arrays of matrices*

A and B must be matrices that have the same number of columns. Elements of matrix B are added from below to elements of matrix A.

```
A = matrix({{1, 2, 3}, {4, 5, 6}, {7, 8, 9}})
B = matrix({{10, 11, 12}})
```

```
joinv(A, B)
A = matrix({{1, 2, 3}, {4, 5, 6}, {7, 8, 9}})
B = ident(3)
joinv(A, B)
```

See also [joinh](#), [block](#), [size](#)

---

## Control functions

[float](#) - display results in floating point format  
[frac](#) - display results in the form of rational fractions  
[none](#) - hide working variables  
[if \(c\)](#) - conditionally execute statements  
[elif \(c\)](#) - alternative condition for if  
[else](#) - alternate block of statements  
[end](#) - terminate if and while statements or definition of user function  
[while \(c\)](#) - repeat statements  
[break](#) - terminate execution of a while loop  
[continue](#) - pass control to the next iteration of a while loop  
[func](#) - definition of user function  
[ret](#) - return from function  
[use](#) - use of other expression

---

### float

```
float
```

Display results in floating point format.

*arguments: none*

All the variables, which have received the values after *float*, will be displayed in floating point format.

```
frac
a = 3 / 7  # it is displayed in the form of rational fraction
float
b = a      # it is displayed in floating point format
```

See also [frac](#), [none](#), [rat](#)

---

### frac

```
frac
```

Display results in the form of rational fractions.

*arguments: none*

All the variables, which have received the values after *frac*, will be displayed in the form of fractions.

**frac**

```
a = 3 / 7 # it is displayed in the form of rational fraction
```

See also [float](#), [none](#), [rat](#)

---

**none**

none

Hide working variables.

*arguments: none*

No variables, which have received the values after *none*, will be displayed on [the screen of results](#).

```
# first 10 Fibonacci numbers
fib = zero(10)
none
fib(2) = 1, i = 3
while (i <= 10)
  fib(i) = fib(i - 2) + fib(i - 1)
  i = i + 1
end
# the working variable i is not displayed
```

See also [float](#), [frac](#), [rat](#)

---

**if**

```
if (condition)
  statements
end
```

Conditionally execute statements.

*condition: any expressions or variables combined by logic operations*

*statements: one or several expressions*

When *condition* is nonzero, *statements* execute.

Each *if* must be paired with a matching *end*.

The general form of the statement is:

```
if (condition1)
    statements1
elif (condition2)
    statements2
else
    statements3
end
# analogue of function sign() for real numbers:
if (x < 0)
    y = -1
elif (x == 0)
    y = 0
else
    y = 1
end
```

See also [elif](#), [else](#), [end](#), [while](#), [logic operations](#)

---

## **elif**

```
if (condition1)
    statements1
elif (condition2)
    statements2
end
```

Alternative condition for if.

*condition1*, *condition2*: any expressions or variables combined by logic operations

*statements1*, *statements2*: one or several expressions

When *condition1* is zero and *condition2* is nonzero, *statements2* execute. See [if](#) for more information.

See also [if](#), [else](#), [end](#), [while](#), [logic operations](#)

---

## **else**

```
if (condition)
    statements1
else
    statements2
end
```

Alternate block of statements.

*condition: any expressions or variables combined by logic operations*

*statements1, statements2: one or several expressions*

When *condition* is zero, *statements2* execute. See [if](#) for more information.

See also [if](#), [elif](#), [end](#), [while](#), [logic operations](#)

---

## **end**

```
if (condition)
    statements
end
while (condition)
    statements
end
func name(parameters)
    statements
end
```

Terminate *if* and *while* statements or definition of user function.

*arguments: none*

See also [if](#), [elif](#), [else](#), [while](#), [break](#), [continue](#), [func](#)

---

## **while**

```
while (condition)
    statements
end
```

Repeat statements.

*condition: any expressions or variables combined by logic operations*

*statements: one or several expressions*

*Statements* execute repeatedly until the value of *condition* is nonzero.

Each *while* must be paired with a matching *end*.

```
# analogue of function sum() for array A:
s = 0
i = 1
while (i <= size(A))
    s = s + A(i)
```



```
    i = i + 1
end
```

See also [break](#), [continue](#), [end](#), [if](#), [logic operations](#)

---

## **break**

```
while (condition)
    statements1
    break
    statements2
end
while (condition1)
    statements1_1
    while (condition2)
        statements2_1
        break(N)
        statements2_2
    end
    statements1_2
end
```

Terminate execution of a *while* loop.

*condition1, condition2: any expressions or variables combined by logic operations*

*statements: one or several expressions*

*N: integer constant greater then zero*

*break* terminates the execution of a *while* loop. Statements in the loop that appear after the *break* statement are not executed.

*break(N)* terminates the execution of several nested loops (the quantity of loops is set by parameter N).

```
# search of a zero element in array A:
i = 1
while (i <= size(A))
    if (not A(i)), break, end
    i = i + 1
end
# search of a zero element in matrix M:
i = 1
while (i <= size(M)(1))
    j = 1
    while (j <= size(M)(2))
        if (not M(i, j)), break(2), end
        j = j + 1
    end
    i = i + 1
end
```

See also [while](#), [continue](#), [end](#), [if](#)

---

## **continue**

```
while (condition)
    statements1
    continue
    statements2
end
while (condition1)
    statements1_1
    while (condition2)
        statements2_1
        continue(N)
        statements2_2
    end
    statements1_2
end
```

Pass control to the next iteration of a *while* loop.

*condition1, condition2: any expressions or variables combined by logic operations*

*statements: one or several expressions*

*N: integer constant greater than zero*

*continue* passes control to the next iteration of the *while* loop in which it appears, skipping any remaining statements in the body of the loop.

*continue(N)* passes control to the next iteration of the outer *while* loop (the loop number is set by parameter N).

```
# count nonzero elements of array A:
count = 0, i = 0
while (i < size(A))
    i = i + 1
    if (not A(i)), continue, end
    count = count + 1
end
```

See also [while](#), [break](#), [end](#), [if](#)

---

## **func**

```
func name(parameters)
    statements
    ret(x)
end
```

Definition of user function.

*name: the name of function; should begin with the letter; can contain letters, digits and the underscore character "\_"*

*parameters: up to 5 parameters separated by commas; parameters can be absent*

*statements: one or several expressions*

Function definition is terminated with a keyword *end*. Value from function comes back by means of the keyword *ret()*.

Definitions of functions cannot be nested. Recursive functions in full are not supported.

```
# maximum of two numbers
func max(a, b)
    if (a > b)
        ret(a)
    end
    ret(b)
end
max(2, -5) # 2
```

See also [ret](#), [end](#)

---

## **ret**

```
func name(parameters)
    statements
    ret
end
func name(parameters)
    statements
    ret(x)
end
```

Return (a value) from user function.

*without arguments: return from function*

*with one argument: return a value from user function*

```
func f(a)
    if (size(a) < 3)
        ret
    end
    a(3) = 5
end
a = {1, 1, 1}, b = {1, 1}
f(a), f(b)
```

See also [func](#), [end](#)

---

## use

`use(name)`

Use of other expression.

*name: the name of included expression; should begin with the letter; can contain letters, digits and the underscore character "\_"; if the name contains spaces or begins with a digit, then the name should be enclosed in the single quotation marks*

- expression with the name "expr":
  - `#expr`
  - `c = 3e8 # m/s`
  - `# variable Distance is not defined here`  
`t = Distance / c # s`
- use in other expression:
  - `Distance = 40000e3 # 40'000 km`
  - **`use(expr)`**  
`n = 1 / t`

The name of an expression contains spaces:

`use('name with spaces')`

See also [func](#), [Working with libraries](#)

---

## Date-time functions

[date\(\)](#) - current date

[date\(y, m, d\)](#) - date given as year, month, day

[date\(dt\)](#) - extract the date part from a date-time

[time\(\)](#) - current time

[time\(h, m, s\)](#) - time specified in hours, minutes and seconds

[time\(dt\)](#) - extract the time part from a date-time

[now\(\)](#) - current date and time

[ndow\(y, m, n, wd\)](#) - nth day of the week in the month of the specified year

[year\(n\)](#) - duration in years

[year\(dt\)](#) - year part of the date

[month\(n\)](#) - duration in months

[month\(dt\)](#) - month part of the date

[day\(n\)](#) - duration in days

[day\(dt\)](#) - day part of the date

[hour\(n\)](#) - duration in hours

[hour\(dt\)](#) - number of hours

[minute\(n\)](#) - duration in minutes

[minute\(dt\)](#) - number of minutes  
[second\(n\)](#) - duration in seconds  
[second\(dt\)](#) - number of seconds  
[dow\(dt\)](#) - day of week of the given date  
[doy\(dt\)](#) - day of year of the given date  
[week\(dt\)](#) - week number  
[leap\(y\)](#) - indicates whether the specified year is a leap year  
[leap\(dt\)](#) - indicates whether the given date is in a leap year  
[julian\(y, m, d\)](#) - date in the Julian calendar given as year, month, day  
[julian\(dt\)](#) - converts the Gregorian date to the Julian date  
[gregorian\(dt\)](#) - converts the Julian date to the Gregorian date

---

## date

```
date()  
date(Y, M, D)  
date(DT)
```

The date function returns the time point.

**date()** - current date.

**date(Y, M, D)** - date given as year, month, day (in the Gregorian calendar).

*arguments: integers, arrays*

Arguments must have the same size.

Y - year, M - month (1..12), D - day of month (1..31). If the arguments are out of range, the date is adjusted accordingly.

**date(DT)** - extract the date part from a date-time.

*arguments: time points, arrays of time points*

```
date() # today; default time is 00:00:00  
date(1999, 12, 31) # 1999-12-31, December 31, 1999  
date({1993, 2000}, {8, 1}, {31, 1}) # {1993-08-31, 2000-01-01}  
d = date(2009, 5, 16) + time(18, 37, 54)  
date(d) # 2009-05-16
```

See also [now](#), [time](#), [ndow](#), [julian](#)

---

## time

```
time()  
time(H, M, S)  
time(DT)
```

The time function returns the duration.

**time()** - current time.

**time(H, M, S)** - time specified in hours, minutes and seconds.

*arguments: H, M - integers, arrays; S - real numbers, arrays*

Arguments must have the same size.

H - hour of day (0..23), M - minute (0..59), S - second (0 .. 59.999). If the arguments are out of range, the time is adjusted accordingly. The accuracy of the time - 1 millisecond (0.001 sec).

**time(DT)** - extract the time part from a date-time.

*arguments: time points, arrays of time points*

```
time() # current time
time(14, 38, 10.43) # 14:38:10.430
time({0, 19}, {45, 10}, {0, 1}) # {00:45, 19:10:01}
d = date(2009, 5, 16) + time(18, 37, 54)
time(d) # 18:37:54
```

See also [now](#), [date](#)

---

## **now**

```
now()
```

Current date and time.

The now function returns the time point.

```
now() # current date and time
```

See also [date](#), [time](#)

---

## **ndow**

```
ndow(Y, M, N, WD)
```

Nth day of the week in the month of the specified year.

The ndow function returns the time point.

*arguments: integers, arrays*

Arguments must have the same size.

Y - year; M - month (1..12); N - week number (1..5; five is the equivalent of last);  
WD - day of week (1..7). Days are numbered from Monday (1 - Monday, 2 - Tuesday, ... 7 - Sunday).

```
# third Monday of April, 2009
ndow(2009, 4, 3, 1) # April 20, 2009
# last Saturday of October, 2009
ndow(2009, 10, 5, 6) # October 31, 2009
```

See also [date](#), [time](#)

---

## year

```
year(N)
year(D)
```

**year(N)** - duration in years.

*arguments: integers, arrays*

**year(D)** - year part of the date or the duration.

*arguments: time points and durations, arrays*

```
year(3)
year({8, 1})
dt = date(2009, 5, 16)
year(dt) # 2009
d = month(26)
year(d) # 2
```

See also [month](#), [day](#)

---

## month

```
month(N)
month(D)
```

**month(N)** - duration in months.

*arguments: integers, arrays*

**month(D)** - month part of the date or the duration.

*arguments: time points and durations, arrays*

```
month(3)
month({8, -1})
dt = date(2009, 5, 16)
month(dt)    # 5
d = month(15)
month(d)     # 3
```

See also [year](#), [day](#)

---

## day

```
day(N)
day(D)
```

**day(N)** - duration in days.

*arguments: integers, arrays*

**day(D)** - day part of the date or the duration.

*arguments: time points and durations, arrays*

```
day(3)
day({28, -21})
dt = date(2009, 5, 16)
day(dt)    # 16
d = hour(48)
day(d)     # 2
```

See also [year](#), [month](#), [dow](#), [doy](#)

---

## hour

```
hour(N)
hour(D)
```

**hour(N)** - duration in hours.

*arguments: integers, arrays*

**hour(D)** - hour of the date or the duration.

*arguments: time points and durations, arrays*

```
hour(18)
hour({32, -2})
dt = date(2009, 5, 16) + time(8, 37, 54)
hour(dt)    # 8
t = time(18, 56, 30)
hour(t)     # 18
```



See also [minute](#), [second](#)

---

## minute

```
minute(N)  
minute(D)
```

**minute(N)** - duration in minutes.

*arguments: integers, arrays*

**minute(D)** - minute of the date or the duration.

*arguments: time points and durations, arrays*

```
minute(18)  
minute({32, -2})  
dt = date(2009, 5, 16) + time(8, 37, 54)  
minute(dt) # 37  
t = time(18, 56, 30)  
minute(t)  # 56
```

See also [hour](#), [second](#)

---

## second

```
second(N)  
second(D)
```

**second(N)** - duration in seconds.

*arguments: real numbers, arrays*

The accuracy of the time - 1 millisecond (0.001 sec).

**second(D)** - second of the date or the duration.

*arguments: time points and durations, arrays*

```
second(18)  
second({0.032, -2.5})  
dt = date(2009, 5, 16) + time(8, 37, 54)  
second(dt) # 54  
t = time(18, 56, 30)  
second(t)  # 30
```

See also [hour](#), [minute](#)

---

## dow

dow(DT)

Day of week of the given date.

*arguments: time points, arrays*

Days are numbered from Monday (1 - Monday, 2 - Tuesday, ... 7 - Sunday).

```
dt = date(2009, 5, 16)
dow(dt)    # 6 - Saturday
```

See also [day](#), [doy](#)

---

## doy

doy(DT)

Day of year of the given date.

*arguments: time points, arrays*

Days are numbered starting with 1 (1 - January 1, 32 - February 1, etc.).

```
dt = date(2009, 1, 1)
doy(dt)    # 1
dt = date(2009, 12, 31)
doy(dt)    # 365
```

See also [day](#), [dow](#), [week](#)

---

## week

week(DT)

ISO 8601 week number.

*arguments: time points, arrays*

Weeks are numbered starting with 1. Week 1 of a year is the first week that has the Thursday in this year.

```
dt = date(2009, 1, 1)
week(dt)    # 1
dt = date(2006, 1, 1)
```

```
week(dt)    # 52 - the last week of the year 2005
```

See also [day](#), [dow](#), [doy](#)

---

## leap

```
leap(Y)
leap(DT)
```

**leap(Y)** - indicates whether the specified year is a leap year.

*arguments: integers, arrays*

Year is considered specified in the Gregorian calendar.

**leap(DT)** - indicates whether the given date is in a leap year.

*arguments: time points, arrays*

```
leap(2008)    # 1
leap({1900, 2000}) # {0, 1}
dt = date(2008, 5, 16)
leap(dt)     # 1
```

See also [date](#)

---

## julian

```
julian(Y, M, D)
julian(DT)
```

The julian function returns the time point.

**julian(Y, M, D)** - date in the Julian calendar given as year, month, day.

*arguments: integers, arrays*

Arguments must have the same size.

Y - year, M - month (1..12), D - day of month (1..31). If the arguments are out of range, the date is adjusted accordingly.

**julian(DT)** - converts the Gregorian date to the Julian date.

*arguments: time points, arrays*

```
julian(1582, 10, 4)    # the last day in the Julian calendar
```

```
d = date(1917, 11, 7)
j = julian(d)          # October 25, 1917
d == j                 # 1
```

See also [gregorian](#), [date](#)

---

## **gregorian**

```
gregorian(DT)
```

converts the Julian date to the Gregorian date.

*arguments: time points, arrays*

The gregorian function returns the time point.

```
j = julian(1582, 10, 4) + day(1)
gregorian(j)          # the first day in the Gregorian calendar
```

See also [julian](#)

---