Console Calculator
documentation

Joep Geuskens

June 2019

Contents

1 Introduction

The goal of this program is to be an all-round calculator based on console-input, which means that it is designed to function without a graphical interface. It does however support the implementation of an interface. Some of the calculator's features are:

1. Basic operations like addition, multiplication, subtraction and division of numbers, vectors and matrices.
2. Several mathematical objects like real or complex numbers, vectors, matrices, functions and expressions.
3. Declaring and storing variables.
4. Multiple mathematical functions such as trigonometric functions (and their inverse and hyperbolic versions) or the exponential, logarithm, square root, absolute value, factorial and complex conjugate.
5. Several algorithms, including the ABC-formula, matrix row-reduction and decomposition, derivation and integration.
6. A collection of physical constants.

2 Basic operations and functions

Calculations involving operations can be entered in a very natural way, just like one would enter them into any online calculator or programming language. Example:

`>>> 2+3*5/0.34+(8/12)^2` returns `46.5621`

Functions can be used using the following syntax `[name]([argument])`. Example:

`>>> sin(pi/2)` returns `1`

A complete list of functions can be seen through the command `list(functions)`. Information about a function, such as its syntax, arguments and a short description can be gained with the command `help([name])`. Example:

```
>>> help(asinh)
Help for asinh (function):
Syntax: asinh(s)
Description: Calculates the inverse hyperbolic sine of the argument, defined
    by asinh(z)=ln(z+sqrt(z^2+1)). For real arguments: sinh(asinh(x))=x. This
    does not hold for all complex numbers, due to the logarithm giving values
    on the principle branch.
Arguments: Any complex number or a real number.
Result: A complex or real number which equals the inverse hyperbolic sine
   of the argument.
```

A full overview of available functions can be found at in the Appendix.

3 Variables, expressions and custom functions

Variables can be defined with the natural syntax `>>>[name]=[expression]`. For example:

`>>> a=2^3+cos(pi)` returns `7`

After running this command, the result of the expression is stored as a variable with that name (in the example, `a` is set to equal `7`). Now, that variable can be used in other calculations.

```
>>> a=2^3+cos(pi)
7
>>> b=sqrt(a)
2.6458
>>> b^2
7
```

While declaring variables, a colon can be inserted before the equals-sign to *define* a variable. This means that the value of the variable will only be evaluated once it is used in another calculation. This means that the variable *will always be equal to its definition.* Compare the following examples:

Using '='

```
>>> a=5
5
>>> b=a^2
25
>>> a=3
3
>>> b
25
```

Using ':='

```
>>> a=5
5
>>> b:=a^2      note the : in front of =
a^2
>>> b
25
>>> a=3
3
>>> b
9
```

If a variable is defined using `:=` it is called an *expression.* All expressions will be evaluated when needed, and are always equal to their definition. A special type of expression is a function, which essentially is an expression that can be evaluated with specific values for some of the variables in its definition (which are then called *parameters*). For any other variable in the function, the same principle with `=` and `:=` is used. Example:

```
>>> f(x)=3x^2+x
3x^2+x
>>> f(5)
80
>>> f(0)
0
>>> g(x):=3x^2+a*x
The variable a is undefined. Statement was parsed regardless.*
3x^2+a*x
>>> a=2
2
>>> g(5)
85
```

*: *note that the variable* `a` *need not be defined at the moment* `g` *is defined. However, by the time* `g` *is evaluated,* `a` *needs to have been defined. If this is not the case, an error like* `UndefinedException: Variable with name a is not defined` *will be returned.*

It is of course possible for functions to have multiple parameters. The parameters' names have to be separated by a comma in the name of the function:

```
>>> f(x,y)=x^2+y^2
x^2+y^2
>>> f(3,4)
25
```

Names of variables and functions
It is possible to assign variables with multi-character names. There are some restrictions to the names that functions can have:

1. Each character in the name should be alpha-numerical (including capitals and greek letters). Underscores are also allowed.
2. A name cannot start with a number or an underscore.
3. A name cannot equal pi, e, or i.
4. A name cannot be equal to the name of a predefined function or algorithm. See the Appendix for a full list of functions and algorithms.

4 Matrices and vectors

To create a vector, use the notation `>>> v=[v1,v2,v3]` which creates the column-vector $(v_1, v_2, v_3)^T$. Creating a matrix uses the same notation, with semi-colons to separate rows. Analogously to other variables, the `:=` can be used to define a vector or matrix. This also means that the elements for a vector or matrix can be any other object (such as variables, complex numbers, functions or even other vectors/matrices). Example:

```
>>> v=[1,2]
(1,2)
>>> a=3
3
>>> A:=[a,4;6,5]
[a,4;6,5]
>>> A*v
(11,16)
>>> f(x,y)=[-y, x]/sqrt(x^2+y^2)
(-y, x)/sqrt(x^2+y^2)
>>> f(3,4)
(-0.8, 0.6)
```

Operations and functions
The addition/subtraction (`+` or `-`) operators are defined for vectors and matrices of the same shape and work element wise. The multiplication (`*`) operator is implemented as the dot-product, so it can only be applied to vectors of the same shape or two matrices, if the column count of the first and the row count of the second are equal. Division is only defined if the quotient is a scalar (which can be complex). Exponentiation has not yet been implemented, though it is mathematically defined for integer exponents. Furthermore, one can use a single-quotation mark after an object to transpose it (if the object is a scalar, it is not affected by the operation). Finally, $\times$ or $\sim$ can be used as the cross-product of two vectors of equal dimension, which can either be 2 or 3. Note that the character $\times$ is not in the ASCII-alphabet and thus cannot be used in the Commandprompt on Windows or the Terminal in Linux. Examples:

```
>>> v=[1,2,3]
(1, 2, 3)
>>> w=[4,5,6]
(4, 5, 6)
>>> v+w/2
(3, 4.5, 6)
>>> v*w
32
>>> v'w
32
>>> v*w'
[4, 5, 6;8, 10, 12;12, 15, 18]
>>> ans'
```

```
[4, 8, 12;5, 10, 15;6, 12, 18]
>>> v~w
(-3, 6, -3)
```

There are several basic functions for vector and matrices, such as `abs(), det()` and `inv()`. The most used algorithms for matrices, such as Gaussian and Jordan Elimination and (P)LU-Decomposition (`ref rref` and `lu` respectively) are available as well. See the appendix for are full list of functions and algorithms and how to use them.

5 Functions and algorithms

Functions and Algorithms are objects that take one or more arguments and give a specific output. They behave in a well-defined way and can (in theory) be computed by hand. The main difference between functions and algorithms is that algorithms can take more than one argument (often there are several possibilities for combinations of arguments), whereas functions can only take one argument. Both can be used using a natural notation, namely: `>>> [name]([arguments])`. For example:

```
>>> sin(pi/4)
0.707
>>> f(x)=2x^2
2x^2
>>> integral(f, -1, 2)
6
```

See the appendix for a full list of algorithms and functions.

6 Settings

In order to slightly customize (some of) the output and default algorithm parameters, there are several *settings* that can be changed according to user-preferences. The syntax is as follows:

- To get the current value of a setting: `>>> setting([name])`.
- To change the value of a setting: `>>> setting([name], [new value])`.

Example:

```
>>> sqrt(2)
1.414
>>> setting(precision)
Setting PRECISION is set to 3
>>> setting(precision, 5)
>>> sqrt(2)
1.41421
>>> 1/123
0.00813
>>> setting(notation)
Setting NOTATION is set to 1*
>>> setting(notation, 3)
>>> 1/123
8.13008E-3
```

*: *Settings can either hold integer or boolean (true/false) values. For the* notation *setting, 1=normal, 2=engineering, 3=scientific.*

A complete list of settings and their purpose, can be found in the appendix.

7 Commands

There are a couple of calculator unrelated functions (mostly for input and output functionalities). They can be used as any other function, but they do not calculate anything and (except for the delete command) do not change any of the variables in the current computation. An important thing to

note is that a command can not be part of another input statement, which means that they have to be entered one at a time and can't be combined with other functions/statements etc. For example, to print the LaTeXcode representing an object and the binary graph structure of a function in a Dot format:

```
>>> f(x)=2x^2/sin(sqrt(x))
2x^2/sin(sqrt(x))
>>> dot(f, dotexport)
>>> latex(f, latexexport)
```

After executing the code in the example, two files are created with the names dotexport.dot and latexexport.tex. More information can be found in the appendix.

A List of functions

`sin`	syntax:	`sin(s)`
	description:	Calculates the sine of the argument.
	arguments:	Any real or complex number in radians.
	result:	A complex or real number which equals the sine of the argument. For real arguments the result will be in $[-1, 1]$.
`cos`	syntax:	`cos(s)`
	description:	Calculates the cosine of the argument.
	arguments:	Any real or complex number in radians.
	result:	A complex or real number which equals the cosine of the argument. For real arguments the result will be in $[-1, 1]$.
`tan`	syntax:	`tan(s)`
	description:	Calculates the tangent of the argument.
	arguments:	Any real or complex number in radians.
	result:	A complex or real number which equals the tangent of the argument.
`sind`	syntax:	`sind(s)`
	description:	Calculates the sine of the argument.
	arguments:	Any real number in degrees.
	result:	A real number from $[-1, 1]$ which equals the sine of the argument.
`cosd`	syntax:	`cosd(s)`
	description:	Calculates the cosine of the argument.
	arguments:	Any real number in degrees.
	result:	A real number from $[-1, 1]$ which equals the cosine of the argument.
`tand`	syntax:	`tand(s)`
	description:	Calculates the tangent of the argument.
	arguments:	Any real number in degrees.
	result:	A real number which equals the tangent of the argument.
`asin`	syntax:	`asin(s)`
	description:	Calculates the inverse sine (arcsine) of the argument. That is, the number whose sine equals the argument.
	arguments:	Any complex number or a real number from $[-1, 1]$.
	result:	A complex or real number whose sine equals the argument. For real arguments the result will be in $[-\pi/2, \pi/2]$.
`acos`	syntax:	`cosd(s)`
	description:	Calculates the inverse cosine (arccosine) of the argument. That is, the number whose cosine equals the argument.
	arguments:	Any complex number or a real number from $[-1, 1]$.
	result:	A complex or real number whose cosine equals the argument. For real arguments the result will be in $[0, \pi]$.
	syntax:	`tand(s)`

`atan`

	description:	Calculates the inverse tangent (arctangent) of the argument. That is, the number whose tangent equals the argument.
	arguments:	Any real or complex number.
	result:	A complex or real number whose tangent equals the argument. For real arguments the result will be in $[-\pi/2, \pi/2]$.
`sinh`	syntax:	`sinh(s)`
	description:	Calculates the hyperbolic sine of the argument, defined as $\sinh(s) = (\exp\{(s)\} - \exp\{(-s)\})/2$.
	arguments:	Any real or complex number in radians.
	result:	A complex or real number which equals the hyperbolic sine of the argument.
`cosh`	syntax:	`cosh(s)`
	description:	Calculates the hyperbolic cosine of the argument, defined as $\cosh(s) = (\exp\{(s)\} + \exp\{(-s)\})/2$.
	arguments:	Any real or complex number in radians.
	result:	A complex or real number which equals the hyperbolic cosine of the argument.
`tanh`	syntax:	`tanh(s)`
	description:	Calculates the hyperbolic tangent of the argument, defined as $\tanh(s) = \sinh(s)/\cosh(s)$.
	arguments:	Any real or complex number in radians.
	result:	A complex or real number which equals the hyperbolic tangent of the argument.
`asinh`	syntax:	`asinh(s)`
	description:	Calculates the inverse hyperbolic sine of the argument, defined by $\operatorname{asinh}(z) = \ln(z + \sqrt{z^2+1})$. For real arguments: $\sinh(\operatorname{asinh}(x)) = x$. This does not hold for all complex numbers, due to the logarithm giving values on the principle branch.
	arguments:	Any complex number or a real number.
	result:	A complex or real number which equals the inverse hyperbolic sine of the argument.
`acosh`	syntax:	`acosh(s)`
	description:	Calculates the inverse hyperbolic cosine of the argument, defined by $\operatorname{acosh} z = \ln(z + \sqrt{z^2-1})$. For real arguments: $\cosh(\operatorname{acosh}(x)) = x$. This does not hold for all complex numbers, due to the logarithm giving values on the principle branch.
	arguments:	Any complex number or a real number.
	result:	A complex or real number which equals the inverse hyperbolic cosine of the argument.
`atanh`	syntax:	`atanh(s)`
	description:	Calculates the inverse hyperbolic tangent of the argument, defined by $\operatorname{atanh} z = \ln((z+1)/(z-1))/2$. For real arguments: $\tanh(\operatorname{atanh}(x)) = x$. This does not hold for all complex numbers, due to the logarithm giving values on the principle branch.
	arguments:	Any complex number or a real number.
	result:	A complex or real number which equals the inverse hyperbolic tangent of the argument.
`toRad`	syntax:	`torad(s)`
	description:	Converts the argument from degrees to radians. This is done using $rad = deg \cdot \pi/180$
	arguments:	Any real number.
	result:	A real number which corresponds to the same angle in radians as the argument in degrees.
`toDeg`	syntax:	`todeg(s)`

	description:	Converts the argument from radians to degrees. This is done using $deg = rad \cdot 180/\pi$
	arguments:	Any real number.
	result:	A real number which corresponds to the same angle in degrees as the argument in radians.
`sqrt`	syntax:	`sqrt(s)`
	description:	Calculates the square root of the argument. If the setting `COMPLEX_ENABLED` is set to true, the argument can be negative too, in which case the result will be complex.
	arguments:	A positive (zero included) number, or a negative or complex number (see above).
	result:	A real non-negative number whose square equals the argument, or a complex number (see above).
`abs`	syntax:	`abs(x)`
	description:	Calculates the absolute value $\|x\|$ of the argument.
	arguments:	Any number or vector, complex or real.
	result:	A real non-negative number which equals the distance from x to the coordinate origin.
`ln`	syntax:	`ln(s)`
	description:	Calculates the natural logarithm of the argument. If s is complex, it returns the natural logarithm of s in the principle branch, defined by $\mathrm{Ln}\, s = \ln(\|(s)\|) + i \cdot \|s\|$
	arguments:	Any non-zero complex or real number.
	result:	If s is real: a number y such that $e^y = x$. If s is complex: a number w such that $\mathrm{Exp}(w) = s$, where Exp is the complex exponential on the principle branch.
`log`	syntax:	`log(s)`
	description:	Calculates the base-10 logarithm of the argument. If s is complex, it returns the natural logarithm of s in the principle branch divided by $\ln 10$, see ln.
	arguments:	Any non-zero complex or real number.
	result:	If s is real: a number y such that $10^y = x$. If s is complex: $\mathrm{Ln}(s)/\ln(10)$, where Ln is the complex natural logarithm on the principle branch.
`exp`	syntax:	`exp(s)`
	description:	Calculates the exponential of the argument. If s is complex, it returns the exponential of s in the principle branch.
	arguments:	Any complex or real number.
	result:	If s is real: e^s, where e is Euler's number. If s is complex: $\mathrm{Exp}(s) = e^{\mathrm{Re}(s)} \cdot (\cos(\mathrm{Im}(s)) + i \cdot \sin(\mathrm{Im}(s)))$, where $\mathrm{Re}(s)$ and $\mathrm{Im}(s)$ are the real and imaginary parts of s.
`root`	syntax:	`root([z,n])`
	description:	Calculates the n-th root of z. If z is complex, it returns the root in the principle branch.
	arguments:	A vector with two complex or real numbers, the second of which is the order of the root.
	result:	$z^{1/n}$.
`fact`	syntax:	`fact(n)`
	description:	Calculates the factorial of the argument (written as $n!$)
	arguments:	Any non-negative real integer.
	result:	the factorial of n: $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$
`conj`	syntax:	`conj(z)`
	description:	Calculates the complex conjugate of the argument: $\mathrm{conj}(z) = \bar{z} = \mathrm{conj}(x+iy) = x - iy$

	arguments:	Any real or complex number, vector or matrix.
	result:	The complex conjugate of the argument (if the argument is vector or matrix shaped, a new vector or matrix of the conjugates of each element is returned).
`det`	syntax:	`det(A)`
	description:	Calculates the determinant of the argument.
	arguments:	Any real-valued square matrix.
	result:	The determinant of A.

B List of algorithms

`derivative`	syntax:	`derivative(f)`, `derivative(f, x)` or `derivative(f, x, a)`
	description:	Analytically calculates the derivative of the function with respect to the variable x (if provided) at location a (if provided).
	arguments:	f: a function depending on one or more variables. x: the name of the variable to which f should be derived (not need if the function only has one variable). For multivariate functions x can also be a vector for a directional derivative. a: (optional) the value to be used as the input for the derivative.
	result:	If a is given $\frac{\mathrm{d}f}{\mathrm{d}x}$ (or $(x \cdot \boldsymbol{\nabla}(f))(a)$) will be returned, otherwise $\frac{\mathrm{d}f}{\mathrm{d}x}$ is returned.
`grad`	syntax:	`grad(f)` or `grad(f, a)`
	description:	Analytically calculates the gradient of the given scalarfunction. The gradient is defined as $(\boldsymbol{\nabla} f)_i = \frac{\partial f}{\partial x_i}$. If a location a is provided, the gradient at a will be calculated.
	arguments:	f: a scalar-shaped function (that means `f.shape()==0`), depending on one or more variables. a (optional): the value to be used as the input for the gradient.
	result:	If a is given, $\boldsymbol{\nabla}(f)(a)$ will be returned as a vector, otherwise $\boldsymbol{\nabla} f$ is returned as a vectorfunction.
`integral`	syntax:	`integral(f, a, b)` or `integral(f, a, b, s)`
	description:	Numerically approximates the definite integral of the function f from a to b with (optional) s steps (nodes) using Gaussian-Legendre Quadrature.
	arguments:	f: a function depending on n variables. a: a vector containing n lower bound values. If $n = 1$, this can be a (real) scalar. b: a vector containing n upper bound values. If $n = 1$ this can be a (real) scalar only if a is one as well. s (optional): the amount of steps in the numerical approximation, the default is set in the `INT_DEF_STEPS` setting. The order of the lower- and upper bound values should correspond to the order of the variables of f.
	result:	An approximation of the definite integral of f between a and b. The approximation is exact for polynomials with degree $< 2N + 1$, where N is the amount of steps.
`ref`	syntax:	`ref(M)` or `ref(M, b)`
	description:	Calculates the row echilon form of the matrix m (optionally augmented with b) using Gaussian Elimination.
	arguments:	M: a (nxm)-matrix with real elements. b (optional): a (nxk)-matrix or n-vector with real elements, this is the augmented matrix (or vector) in the Gaussian Elimination.

	result:	A $(nxm + k)$-matrix ($k = 0$ if b is not given) containing the result of the Gaussian Elimination on M augmented with b.
`rref`	syntax:	`ref(M)` or `ref(M, b)`
	description:	Calculates the reduced row echilon form of the matrix m (optionally augmented with b) using Jordan-Gauss Elimination.
	arguments:	M: a (nxm)-matrix with real elements. b (optional): a (nxk)-matrix or n-vector with real elements, this is the augmented matrix (or vector) in the Jordan-Gauss Elimination.
	result:	A $(nxm + k)$-matrix ($k = 0$ if b is not given) containing the result of the Jordan-Gauss Elimination on M augmented with b.
`lu`	syntax:	`lu(M)`
	description:	Calculates the PLU decomposition of M. This means that three matrices (L, U, P) are calculated, such that $PM = LU$ (so $M = P^{-1}LU$. L is a lower triangular matrix, U is an upper triangular matrix, and P is a pivot matrix.
	arguments:	M: a (nxn)-matrix with real elements.
	result:	A vector containing (L, U, P), with L upper triangular, U lower triangular and P a pivot matrix. Such that $LU = PM$.
`abc`	syntax:	`abc(a, b, c)`
	description:	Uses the ABC-Formula to calculate the root of a quadratic equation of the form $y = ax^2 + bx + c$. If the setting `COMPLEX_ENABLED` is set to `true`, the a complex result may be returned.
	arguments:	a, b and c: real numbers
	result:	A vector with 2 elements containing both roots, if one or both of the roots is complex and the setting `COMPLEX_ENABLED` is set to `false`, one or both of the elements may be `null`. If only one element is `null`, it will be the second one.
`simplify`	syntax:	`simplify(e)`
	description:	Tries to simplify a given expression (or function). This involves computing all numerical operations (like $2 * 3$ or $\log(12)$, removing trivial operations (such as $x * 0$, $x * 1$, $y + 0$, y^1) and reordering the expression (see help(reorder)).
	arguments:	e: a function or an expression
	result:	An object (function or expression) which (mathematically) equals e, but without numerical operations and trivial operations.
`reorder`	syntax:	`reorder(e)`
	description:	Tries to reorder a given expression (or function). This involves moving numberical values to the front, and functions to he back of the expression.
	arguments:	e: a function or an expression
	result:	An object (function or expression) which (mathematically) equals e, but in a different (more natural) order.

C List of Commands

`del`	syntax:	`del(a,b,...)` or `delete(a,b,...)`
	description:	Deletes the given variables. This can be used to free memory if you are working with large objects which are no longer needed.
	arguments:	a, b, ...: arbitrary amount of variable names.
	result:	-
`dot`	syntax:	`dot(a)` or `dot(a, name)`

	description:	Exports the given variable as a tree graph in a dot format (see Graphviz for more info) to a file with the given name or `'output_dot*.dot'` where * is the first available integer. If the argument equals 'dependencies' the dependencies graph is exported (developer feature).
	arguments:	a: an expression, function or 'dependencies'. name (optional): the name of the file to which the variable is exported (without extension).
	result:	A .dot file containing the dot code representing the tree graph of the argument
`execute`	syntax:	`execute(file)`
	description:	Executes the statements in the given file.
	arguments:	file: the name of the file to be executed (including the .cal extension and the path if the file is not in the same directory as the program).
	result:	The statements in the file and their results printed to the console.
`help`	syntax:	`help(command)`
	description:	Displays the entry in the help document for the given command.
	arguments:	command: the name of the command which's help page should be displayed.
	result:	The help page of the given command printed to the console.
`latex`	syntax:	`latex(a)` or `latex(a, name)`
	description:	Exports the given variable in its LaTeXformat to a file with the given name or `'output_latex*.tex'` if no name is given, where * is the first available integer.
	arguments:	a: the name of the variable to be exported as LaTeX. name (optional): the name of the file to which the LaTeXcode should be written (without the .tex extension).
	result:	A .tex file containing the LaTeXcode representing the argument.
`list`	syntax:	`list(type)` or `list()`
	description:	Prints a list of available help pages for commands, functions, algorithms or settings. If no argument is given, all pages are listed.
	arguments:	type (optional): the category which should be listed. Can be 'algorithms', 'commands', 'functions' or 'settings'.
	result:	A list of available commands/functions/algorithms/settings.
`print`	syntax:	`print(a)`
	description:	Prints the given variable to the console.
	arguments:	a: the name of the variable to be printed to the console.
	result:	The variable in text printed to the console.
`setting`	syntax:	`setting(key)` or `setting(key, value)`
	description:	Prints the setting if only the key is given, or sets the setting to a new value if both key and value are given. If the key equals 'reset' the settings will be reset to their default values. See help(⟨setting⟩) or list(setting) for more info.
	arguments:	key: the name of the setting or 'reset'. value (optional): the new value of the setting (note that each setting requires a specific value type (e.g. integer, string, boolean etc.)
	result:	The current value of the setting printed to the console if only the key is given.
`shape`	syntax:	`shape(a)`
	description:	Prints the shape of the given variable.
	arguments:	a: the name of the variable.
	result:	The shape of the given variable.
`time`	syntax:	`time(expression)` or `time(expression, runs)`

	description:	Measures the time needed to execute the given expression. To increase precision, the expression is run multiple times. How often the expression is run can be given in the second argument or else is set to the `TIMER_DEF_RUNS` setting.
	arguments:	expression: the expression whose execution time should be measured. runs (optional): how often the expression should be run.
	result:	The total time required for all runs and the average time per run.
`type`	syntax:	`type(a)`
	description:	Prints the type (=java class name) of the given variable.
	arguments:	a: the name of the variable whose type should be printed.
	result:	The type of the given variable.

D List of Settings

`abc_show_text`	name:	`abc_show_text`
	type:	boolean
	description:	whether or not the ABC-Formula algorithm should display more information as text. See help(abc) for more info.
	default:	`true`
`complex_enabled`	name:	`complex_enabled`
	type:	boolean
	description:	whether or not complex numbers can be returned by functions.
	default:	`false`
`complex_in_polar`	name:	`complex_in_polar`
	type:	boolean
	description:	whether or not complex number should be expressed in their polar form. If false, complex numbers will be given in cartesian form.
	default:	`false`
`int_def_steps`	name:	`int_def_steps`
	type:	integer
	description:	The default amount of steps (nodes) used by the Gaussian Quadrature algorithm. See help(integral).
	default:	`16`
`notation`	name:	`notation`
	type:	integer
	description:	The type of notation which is used to display numbers. normal=1, engineering=2, scientific=3.
	default:	`1`
`precision`	name:	`precision`
	type:	integer
	description:	The amount of decimal places to be displayed in the console.
	default:	`3`
`timer_def_runs`	name:	`timer_def_runs`
	type:	boolean
	description:	whether or not the stacktrace of error messages should be shown. The stacktrace can be used by the developer to pinpoint the code generating the error.
	default:	`false`

`timer_def_runs`	name:	`timer_def_runs`
	type:	integer
	description:	The default amount of runs of the time command. See help(time) for more info.
	default:	`100`

E List of Constants

`pi`	name:	Mathematical constant π: the ratio of the circumference and diameter of a circle.
	value:	3.1415926...
`e`	name:	Mathematical constant e (Euler's number): the limit of $(1+1/n)^n$ for $n \to \infty$.
	value:	2.7182818285...
`i`	name:	Imaginary unit: a number such that $i^2 = -1$.
	value:	i
`_c`	name:	Physical constant c: speed of light in a vacuum.
	value:	299792458 (m/s)
`_e`	name:	Physical constant e: electron charge.
	value:	$1.60217662 \cdot 10^{-19}$ (C)
`_mu0`	name:	Physical constant μ_0: magnetic permeability of vacuum.
	value:	$1.2566370614359173 \cdot 10^{-06}$ (N/A^2)
`_epsilon0`	name:	Physical constant ϵ_0: the electric permittivity of vacuum (also: (di)electric constant.
	value:	$8.854187817620389 \cdot 10^{-12}$ (F/m)
`_h`	name:	Physical constant h: Plank's constant
	value:	$6.62607004 \cdot 10^{-34}$ (Js)
`_hbar`	name:	Physical constant $\hbar$: reduces Plank's constant.
	value:	$h/(2\pi) = 1.0545718001391127 \cdot 10^{-34}$ Js
`_G`	name:	Physical constant G: Newton's gravitational constant.
	value:	$6.67408 \cdot 10^{-11}$ (m^3/(kg s^2))
`_g`	name:	Physical constant g: standard gravitational acceleration on earth.
	value:	9.80665 (m/s^2)
`_R`	name:	Physical constant R: molar gas constant.
	value:	8.3144598 (J/(K mol))
`_alpha`	name:	Physical constant α: fine structure constant
	value:	0.0072973525664
`_NA`	name:	Physical constant N_A: Avogadro's number.
	value:	$6.022140857 \cdot 10^{23}$ (1/mol)
`_k`	name:	Physical constant k (or k_B): Boltzmann constant.
	value:	$1.38064852 \cdot 10^{-23}$ (J/K)
`_sigma`	name:	Physical constant σ: Stefan-Boltzmann constant.
	value:	$5.670367 \cdot 10^{-8}$ (W/(m^2 K^4))
`_wien`	name:	Physical constant b: Wien's displacement contant.
	value:	0.0028977729 (m K)
`_Ry`	name:	Physical constant Ry: Rydberg constant
	value:	10973731.568508 (m/s)
`_m_e`	name:	Physical constant m_e: electron mass
	value:	$9.10938356 \cdot 10^{-31}$ (kg)

`_m_p`	name:	Physical constant m_p: proton mass
	value:	$1.672621898 \cdot 10^{-27}$ (kg)
`_m_n`	name:	Physical constant m_n: neutron mass
	value:	$1.674927471 \cdot 10^{-27}$ (kg)
`_u`	name:	Physical constant u: atomic mass unit
	value:	$1.66053904 \cdot 10^{-27}$ (kg)