

so/\ata

Program for mathematical calculations

Table of contents

1 Preface.....	6
2 Getting Started.....	7
2.1 Installation.....	7
2.2 Quick start.....	7
2.3 Calling the program.....	7
2.3.1 so/Vata interpreter.....	7
2.3.2 The Lua interpreter.....	8
2.3.3 matlib library.....	8
2.4 Launch arguments.....	8
2.5 Basics.....	9
2.5.1 Lua syntax.....	9
2.5.2 Functions of the so/Vata interpreter.....	11
2.6 note files.....	14
2.7 Interpreter commands of so/Vata.....	15
2.8 Setting up the program.....	16
3 data: processing lists of data.....	18
3.1 Introduction.....	18
3.2 Array Operations.....	18
3.2.1 Objects.....	18
3.2.2 Filtering and transforming data.....	20
3.3 Input, output and printing.....	22
3.4 Elements of statistics.....	23
4 asciiplot: visualization in console.....	25
4.1 Introduction.....	25
4.2 Canvas Options.....	25
4.2.1 Image size.....	26
4.2.2 Setting up axes.....	26
4.3 Types of graphs.....	27
4.3.1 Plot function.....	27
4.3.2 Tabular data.....	28
4.3.3 Column chart.....	29
4.3.4 Contours.....	30
4.4 Other operations.....	31
5 matrix: matrix operations.....	34
5.1 Introduction.....	34
5.2 Basic operations.....	35
5.2.1 Arithmetic operations.....	35
5.2.2 Methods.....	35
5.3 Reference objects.....	36
5.4 Vector operations.....	38
5.5 Transformations.....	39
5.6 Other operations.....	41

6 complex: complex numbers	42
6.1 Introduction	42
6.2 Basic operations	42
6.3 Functions	43
7 polynomial	44
7.1 Introduction	44
7.2 Basic operations	44
7.3 Approximation	45
7.3.1 Polynomials	45
7.3.2 Splines	46
8 numeric: numerical methods	48
8.1 Introduction	48
8.2 Functions	48
8.2.1 Root Search	48
8.2.2 Mathematical analysis	48
8.3 Ordinary differential equations	49
9 extremum: optimization methods	52
9.1 Introduction	52
9.2 Finding the minimum of a function	52
9.3 Linear programming	53
9.4 Nonlinear approximation of a function	53
9.5 Others optimization methods	54
9.5.1 Annealing method	54
10 random: generators	56
10.1 Introduction	56
10.2 Distributions of random numbers	56
10.3 Other functions	57
11 bigint: long integers	59
11.1 Introduction	59
11.2 Functions	60
11.3 Number systems	60
11.4 Elements of Combinatorics	61
12 rational: rational numbers	62
12.1 Introduction	62
12.2 Functions	62
12.3 Continued fractions	63
13 quaternion: operations with quaternions	64
13.1 Introduction	64
13.2 Functions	64
13.3 Orientation	65
14 special: functions	66
14.1 Introduction	66
14.2 List of functions	66
14.2.1 Gamma functions	66

14.2.2 Beta functions.....	66
14.2.3 Bessel functions.....	67
14.2.4 Other functions.....	67
15 graph: graph operations.....	68
15.1 Introduction.....	68
15.2 A generation.....	68
15.3 Basic operations.....	69
15.3.1 Modification.....	69
15.3.2 Export.....	70
15.3.3 Other functions.....	71
15.4 Algorithms.....	71
15.4.1 Checking properties.....	71
15.4.2 Finding the way.....	72
16 units: units of measurement.....	73
16.1 Introduction.....	73
16.2 Conversion of units of measurement.....	73
16.2.1 List of rules.....	73
16.2.2 Functions.....	73
17 const: collection of constants.....	75
17.1 Introduction.....	75
17.2 User constants.....	75
18 gnuplot: plotting graphs in GNUplot.....	76
18.1 Introduction.....	76
18.2 Main functions.....	77
18.3 State management.....	79
19 symbolic: Elements of Computer Algebra.....	81
19.1 Introduction.....	81
19.2 Basic operations.....	81
20 qubit: quantum computing emulation.....	83
20.1 Introduction.....	83
20.2 Qubits.....	83
20.3 Gates.....	84
20.4 An example.....	86
21 lens: paraxial optics.....	87
21.1 Introduction.....	87
21.2 Optical systems.....	87
21.2.1 Description.....	87
21.2.2 Analysis.....	88
21.2.3 Other functions.....	89
21.3 Laser radiation.....	89
22 geodesy: coordinate transformations.....	91
22.1 Introduction.....	91
22.2 Direct and inverse problems.....	91
22.3 Coordinate transformations.....	92

22.3.1 Different ellipsoids.....	92
22.3.2 Single ellipsoid.....	92
22.4 Other functions.....	93
23 Applications.....	94
23.1 Setting up the environment.....	94
23.2 Localization of the program.....	94
23.3 Adding a module.....	94
23.4 Program input/output parameters.....	95

1 Preface

so/\ata is a console program for mathematical calculations and modeling, written in the Lua programming language. It consists of a library of mathematical modules *matlib* (can be used independently) and the REPL interpreter.

This program arose from the Author's need for a console calculator. Some of the Linux programs were tested, but they did not meet expectations either because of their specific syntax or because of the limited capabilities. Comparison of programming language interpreters for Python And Lua showed that the latter performs some calculations more efficiently (without using specialized libraries), and also has a smaller delay when starting, so the choice fell on it. Due to the Author's interest in various applications of mathematics, the functionality of the program increased and over time outgrew the capabilities of the calculator, including, I hope, its convenience.

Using pure Lua language without third party dependencies for so/\ata has both its pros and cons. The latter include:

- performance is determined by the efficiency of code interpretation
- the program's capabilities are limited by the standard Lua library
- single-threaded program execution
- interface - command line
- lack of built-in graphical tools

The advantages include:

- ready to use immediately after downloading
- cross-platform
- ease of expansion and customization

With the listed limitations in mind, so/\ata first of all tries to be a handy tool for performing simple calculations and testing hypotheses, the results of which can be used by more powerful programs, such as Matlab.

2 Getting Started

2.1 Installation

To work with `so/\ata` requires the Lua interpreter to be installed. Lua 5.1 and higher could be used, however, the latest versions will work more efficiently. The environment variable `PATH` must also be set as well (See Appendix 1).

`so/\ata` does not require additional installation, just download the latest release version from [Github](#) and unpack it.

2.2 Quick start

If you already have experience in Lua, Python or similar programming languages, then you can speed up your acquaintance with `so/\ata`. To do this, go to the program folder and run the following commands

```
lua sonata.lua notes/intro_ru.note # basics of work
lua sonata.lua notes/modules.note  # module description
```

2.3 Calling the program

To access the program, you need to open the terminal and go to the folder with the program (further we will show how to set up a call from any location). After executing the command

```
lua sonata.lua
```

an invitation should be displayed:

```
# #          ===== so/\ata =====          # #
# #          ===== 1.00 =====          # #

----- help([function]) = get help -----
----- use([module]) = expand functionality -
----- quit() = exit -----

## _
```

There are three possible ways to work with the program: call the `so/\ata` interpreter, call standard Lua interpreter or independent use of the mathematical library.

2.3.1 `so/\ata` interpreter

This is the default mode, which is started by the command

```
lua sonata.lua
```

Interpreter written on Lua, therefore it is “interpreted” during the work process, but at the same time it expands the functionality of the program.

Advantages:

- additional functions (command mode, use of colors, logging)
- more informative output
- the same interaction interface in all versions of Lua 5.x

Flaws:

- the speed of input processing is slightly lower than in the Lua interpreter
- line breaks need to be explicitly marked

2.3.2 The Lua interpreter

To use the standard Lua interpreter execute the command

```
lua -i sonata.lua
```

The necessary libraries will be loaded, after which `so/\ata` will transfer control. The advantages and disadvantages described in the previous paragraph will then switch places.

2.3.3 matlib library

Mathematical library *sonata/matlib* can work without other program components. However, if you plan to use only individual files, you should remember that modules may depend on each other.

2.4 Launch arguments

Call

```
lua sonata.lua -h
```

outputs a list of arguments that could be applied.

- **-e expression**

Allows you to execute a command, display the result, and terminate the program.

```
lua sonata.lua -e "1 + 2 + 3 + 4"
echo "1+2+3+4" | lua sonata.lua -e
```

- **--doc [language]**

Generates *html* page with a description of the current version of the program, the file is saved to the current directory. If there are localization files, you can additionally specify the file name and get the translated text.

```
lua sonata.lua --doc      # english text
lua sonata.lua --doc en # Russian text
```

- **--lang [language]**

Generates a localization file in the folder *sonata/locale*, the argument is the name of the language, details in Appendix 2. If there is no argument, the current language setting will be displayed.

```
lua sonata.lua --lang      # displays the current language
lua sonata.lua --lang eo # generates/updates localization file
```

- **--new module alias [description]**

Generates a template for a new math library, the arguments are the full name of the module (the name of the generated file in *sonata/matlib*), its short nickname and (optionally) a description. Details are in Appendix 3.

```
lua sonata.lua matrices Mat "Matrix operations"
```

- **--test [module]**

Runs unit tests for *matlib*, optional argument is the name of the module being tested.

```
lua sonata.lua --test      # test all libraries
lua sonata.lua --test complex # test the complex numbers library
```

- **--io mode [options]**

Configures the program's input/output according to the specified parameters, third-party libraries may be used, but they must be pre-installed on the computer. Details are in Appendix 4.

```
lua sonata.lua --io tcp 23456 # run the program in server mode
lua sonata.lua --io w        # run program in output window mode
```

Arguments can be one or more *.lua* or *.note* files, the program will execute them sequentially and then exit.

```
lua sonata.lua fileA.lua fileB.note
```

2.5 Basics

2.5.1 Lua syntax

The main data types in *sonata* are numbers, strings, and tables. Numbers are usually divided into integers and floating point, but Lua processes it as “real”, i.e. the result of the operation “1/2” will be “0.5” (for integer division in the latest Lua versions the operator “//” is introduced). A string is a sequence of characters enclosed in single or double quotes. If the string is long (contains hyphens), the `[[` and `]]` signs are used instead of quotes. Line comments begin with the “`--`” character. The combination of the “`--`” and “`[[]]`” allows you to get a “long” comment.

The table is the key Lua element, which is a container that holds other objects. Table elements are enclosed in curly brackets and can be ordered either by key (dictionaries) or by index (lists) if the key is not specified. Indexing starts with 1. To get

an element, you must write its index/key in square brackets. If the key is a string without spaces (and starts with a letter or “_”), it is enough to specify it after the table name with a dot.

```
a = {10, 20, 30} -- sorted by index
a[1]             -- 10, the first element of the table
b = {x=10, y=20, z=30} -- sorted by key
b.x             -- 10, element with index x
c = {x=10, 20, 30} -- combination is possible
c.x             -- 10
c["x"]          -- 10, equivalent to c.x
c[1]            -- 20
```

Dictionaries in so/\ata are most often used to describe parameters, while lists represent arrays.

Functions are another key component of Lua. Its call looks like in most programming languages: the function name and a list of arguments, separated by commas and enclosed in parentheses. The number of elements returned may be greater than one. If the function takes a single argument, which is a string or table, the parentheses can be omitted.

```
print(1, 2, 3) -- prints 1 2 3
print('abc')   -- prints abc
print 'abc'    -- brackets for string are not required
```

Below is an example of a function declaration.. The definition is between the keywords *function* and *end*. If a result needs to be returned, it is specified after *return*.

```
function foo(x, y)
  local x2, y2 = x*x, y*y
  return x2+y2, x2-y2
end
a, b = foo(3, 4)
```

All variables in Lua are global by default, i.e. available for modification and reading in the entire code. To limit the scope, use the keyword *local*. Declaring local variables is not mandatory, but it allows you to avoid many non-obvious errors and speeds up calculations.

Since a function is also a data type, it can be treated like other variables, including being stored in a table. If a function needs to process data from the same table in which it resides, the “:” (colon) operator is provided, allowing the table to be implicitly passed as the first argument.

```
t = {x = 0}
t.foo = function(var) var.x = 1 end -- change the field in the table
t.foo(t)    -- explicit call
t:foo()     -- implicit call
```

A few words should also be said about the standard library of mathematical functions Lua, which is called *math*. This library is a wrapper over the standard mathematical functions of the C language. Its content may differ in different versions of Lua, but usually includes standard functions such as **exp()**, **sqrt()**, **log()**, trigonometric functions, random number generator, constants **pi**, **huge** (infinity) and a number of auxiliary functions. If you are concerned about performance and you are working with ordinary numbers, use the standard library functions, but if you need flexibility, use the overridden functions from the module *main*.

If you need to download a Lua file from code or an interpreter, use the function **dofile(file_name)**. Also, the following functions may be useful in your work: **tonumber()**, tries to convert a string to a number, **tostring()** returns a string representation for the specified object.

See the official description [manual.html](http://lua.org/manual.html) to get more about the Lua language. For those who wish to delve deeper into the language, we recommend [Programming in Lua](#).

2.5.2 Functions of the so/\ata interpreter

After running the so/\ata interpreter you find yourself in an interactive mode of working with the program, which is not much different from working in the Lua interpreter: the program returns the result for each entered command (if it is not *nil*) and proceeds to wait for the next input. For example, if you enter the expression “1 + 2 + 3” and press *Enter*, the program will print “6” and store the result in a variable *ANS*.

The interpreter of so/\ata is relatively simple, it can only process the current line. Therefore, to continue entering a long expression, you must put the “\” sign before the transition to a new line. This character is not standard for Lua, so you can specify it in the comment to avoid conflicts.

```
## 1 + 2 + \  
.. 3 + 4  
-- or  
## 1 + 2 + --\  
.. 3 + 4
```

If there are more than 2 lines, you can put two backslashes, and the end of the input will be set to an empty line.

```
## 1 + \  
.. 2 +  
.. 3 +  
.. 4  
..  
10
```

When the result does not need to be printed, you can put the “;” sign at the end of the line (as in Matlab).

When the program starts it loads the *main* module, which contains a number of standard mathematical functions, as well as some additional procedures. To get a list of currently available functions, call

```
## help()
```

Functions in *main* can be divided into two categories. The first category includes commonly used mathematical functions such as sine or logarithm. All of them are redefined to work with the data types used in so/\ata, for example, with complex or rational numbers. You can get more information about a specific function using the expression **help(function)**, For example,

```
## help(sin)
```

The second category includes auxiliary so/\ata functions. With one of them, **help()**, we have already met. If its argument is another function and there is help information for it, then this info is printed. For most functions, an example of their call is also displayed. The argument can be a string with the module name, for example,

```
## help 'Main'
```

displays the contents of the specified module. Call

```
## help '*'
```

outputs information about all loaded modules and functions. When the argument of **help()** is an arbitrary object, its type and value are displayed, as well as a list of available methods.

The function **use()** allows you to download new modules. When called without arguments, a list of available libraries and their status (loaded or not) are displayed on the screen.

```
## use()
MODULE      ALIAS      USED
...
main        Main      ++
...
```

You can load one or more modules at a time.

```
## use 'data'                -- load one module
## use {'complex', 'matrix'} -- load the listed modules
## use '*'                   -- load all modules
```

Explicit call of **use()** is not necessary for a single module, the component is loaded the first time the corresponding constructor is called.

A reference to a module is stored in a variable with a specific name (alias), which is usually shorter in length and starts with a capital letter. Aliases can also be used in a function **use()**, i.e. allowed to write **use('Z')** instead of **use('complex')**. You can use the

standard function **require**('matlib.module') to download modules, in this case you need to explicitly specify the name of the variable (alias) to store the link to the library.

As an example, let's consider the module *matrix*, it can be loaded via alias command

```
## use 'Mat'
```

To get information about the module, run

```
## help 'Mat' -- list of module functions
## help(Mat.T) -- help about a specific function
```

It should be noted that all module functions are called using ":", this is done to avoid ambiguity regarding the sign used (dot or colon). However, **the help()** function expects dot notation.

Function **Round**(object, position) takes a number or object (such as a matrix) and rounds it to a specified number of decimal places (by default, to the integer part).

```
## a = Mat {{1,2},{3,4}} * PI
## a[1][1]
3.1415926535898
## b = Round(a, 3)
## b[1][1]
3.142
```

Often, we work with lists or other ordered collections, and we need to get a new list by applying some operation to the elements of the original list. For this purpose, the function **Map**(function, list) is defined:

```
## lst = Map(exp, {0, 1, 2})
## lst
{ 1.0, 2.7182, 7.3890, }
```

This example builds a list of numbers e^n for n from 0 to 2. **Map()** can also be used for some data types, such as matrices. For example, a random matrix can be constructed as follows:

```
## function rnd() return math.random() end
## Map(rnd, Mat:zeros(2, 2))
```

Helper function **Fn**(description) generates a function based on a string of the form "arguments -> expression", which allows some expressions to be written more compactly. If the expression depends on a single variable named x, the argument list can be omitted.

```
## sum = Fn "x, y -> x + y"
-- equivalent to
-- sum = function(x, y) return x + y end
## sum(1, 2)
```

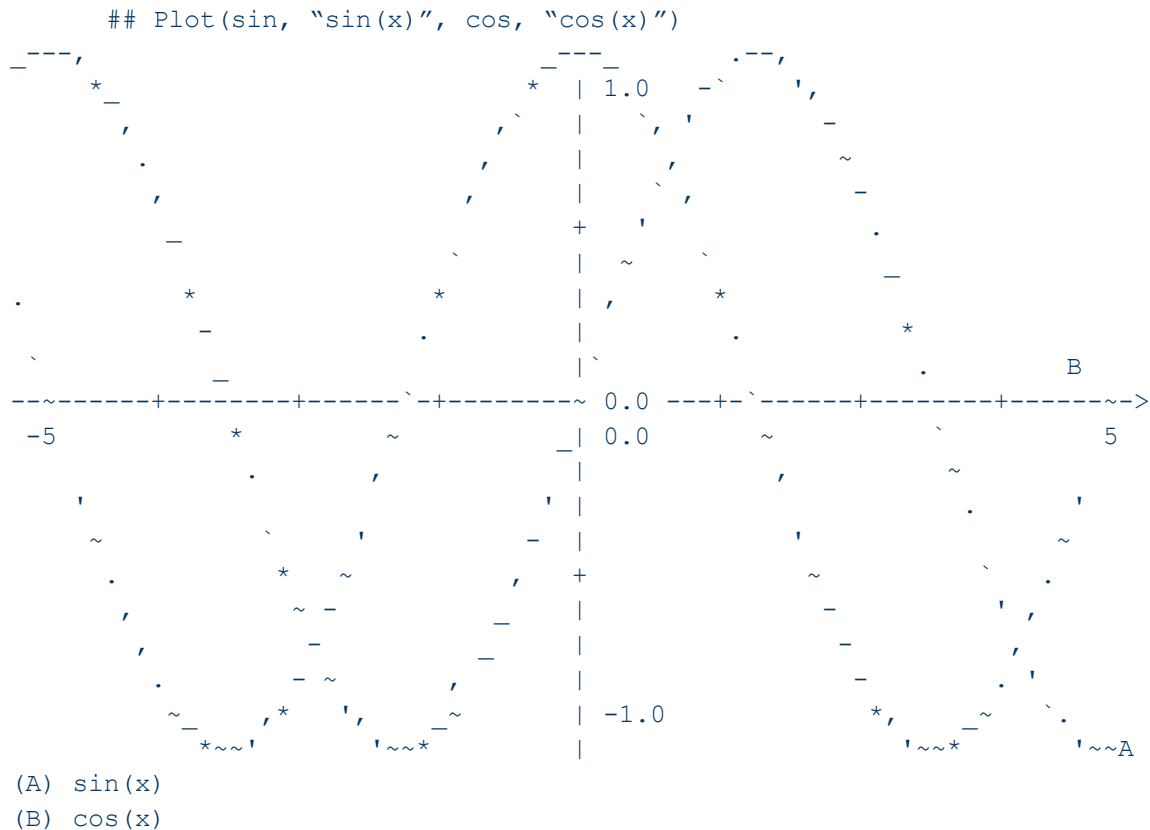
```

3
## odd = Fn "x % 2 ~= 0"
## odd(3)
true

```

If performance is important, it is better to use the standard function definition.

If you need to quickly build a graph without having to explicitly work with the canvas, you can use the function **Plot(...)**, which takes the same set of arguments as **plot()** from *asciipLOT* module, but hides all actions with the object, and therefore does not allow it to be edited.



As can be seen from the graph, the x range is from -5 to 5. If you need to change it then add lists with the required end points.

To terminate the program, run

```

## quit()

```

2.6 note files

Files with extension *note* are processed as if they were commands entered manually by the user. The file is interpreted sequentially, line by line. If the expression returns something other than *nil*, it is printed, long lines are transferred using the “\” and “\\” signs.

The *note* file can be divided into “blocks”, the separator is the line “-- PAUSE”, upon reaching which the interpreter switches to interactive mode. In this mode, the

processing of the source file is suspended, and `so/\ata` executes user commands as long as they are different from the empty string. After that, it begins interpreting the next block in the list.

Line comments are considered part of the general description and are printed. If you need to exclude a file fragment from processing, use multi-line comments. If the comment text is separated from the “--” symbols by a tab, it will be additionally highlighted if the color option is enabled. In general, the block header is its first line.

2.7 Interpreter commands of `so/\ata`

Interpreter `so/\ata` can do more than just work with standard Lua expressions, but also provides a number of additional commands that control its operation. Such commands begin with the “:” sign (similar to Vim).

General commands

- **:help [name]**
Displays a list of commands or outputs information about a command.
- **:q**
Terminates and exits the program. Equivalent to calling **quit()**.
- **:log on/off**
Enable/disable program I/O logging. The result is saved in *note*-file, so it can be used to replay commands in the future.
- **:set module f1, f2, f3 as f4**
The command allows you to import functions and values with a given name from a given module, the “as” word allows you to define an alias. It is similar to the “from module import f3 [as f4]” construction in Python.
- **:clear v1, v2**
Deleting the specified variables from memory. The expression “clear *” deletes all variables.
- **:w expr**
Redirects the calculation result to a window opened with the “--io w” option. The function is intended for “graphical” data, but can be used for any text output.

Working with note files

- **:o name**
Opens the file and loads the data. New blocks are appended to the previously loaded ones.
- **:ls**
Displays a list of command blocks.
- **:rm**
Clears the block list.
- **:show N**
Displays the contents of the Nth or next block in the list
- **:1, 2, 7:9, -3**

Calls commands from the specified blocks. The ':' character separates the beginning and end of the range, negative indices are counted from the end.

Debugging tools

- **:time func**

Evaluates average function execution time in milliseconds.

```
## function sumAB(a,b) return sin(a)*cos(b) + cos(a)*sin(b) end
## :time function() sumAB(PI/3, PI/4) end
```

- **:trace func**

Displays the list and number of function calls made when evaluating a given expression.

```
## :trace function() sumAB(PI/3, PI/4) end
```

2.8 Setting up the program

The program configuration is written in the file *sonata.lua*. You can change the settings of the file inside the folder *sonata*, or create a copy of the file in a convenient location for access.

There are 2 sections for change: *MODULES* and *CONFIGURATION*. The first section presents the modules that are included in *matlib* and can be loaded via the function **use()**. Here you can add or remove (comment) any module. If you don't like the suggested name abbreviations then define your own aliases, for example, rename "Z" to "Cx" or "Mat" to "M".

Section *CONFIGURATION* contains variables corresponding to program parameters. To perform the configuration, you need to uncomment the corresponding parameter and set the required value.

- **SONATA_ADD_PATH**
Path to program modules. If you want to run *so/\ata* from any location, specify the absolute path to the folder in this variable. In Unix systems, you can also configure the program launch command by making an alias in *bashrc* for file *sonata.lua*.
- **SONATA_USE_COLOR**
Highlight the prompt line, error messages, help, etc. This option is system dependent, but generally works fine on Unix.
- **SONATA_READLINE**
Using the library *readline* for maintaining command history and autocompletion. This library is a third-party component and must be installed separately.
- **SONATA_ASCII_PLOT_UNICODE**
This option allows the module *asciipLOT* to use Unicode symbols to improve the appearance of diagrams.
- **SONATA_PROTECT_ALIAS**

If this option is enabled, so/\ata will not allow creating variables whose names coincide with module aliases. The shorter the alias length, the more likely it is to be accidentally "erased".

- SONATA_LOCALIZATION

Contains the name of the localization file. In particular, to enable the Russian language, specify “ru.lua” here. In Windows OS you may additionally need to enable Unicod to correctly display non-Latin letters in the terminal:

`chcp 65001`

3 data: processing lists of data

3.1 Introduction

Module *data* contains methods for processing lists (tables) of data, primarily numbers. Some functions allow working with a list of lists, i.e. with a two-dimensional array. Classes are defined that simplify the generation of sequences and access to array elements. This module also includes statistical processing functions.

3.2 Array Operations

3.2.1 Objects

Some of the module's functions are constructors of objects that behave like Lua tables, but are either generators or references to other tables.

Function **range**(*start*, *end*, [*step*]) allows you to get an object that behaves like a list in a given interval, the default step is 1. If the value for the end is greater than for the beginning, the list will be built in reverse order. The elements of such list are not stored anywhere, they are calculated during access to the corresponding index.

```
## rng = D:range(-10, 10)
## #rng
21
## for i, v in ipairs(rng) do print(i, v) end
1   -10
...
21   10
```

With the object **range** you can perform linear transformations such as addition and multiplication by a number.

```
## rng2 = 2*rng + 1
## rng2
{-19, -17 .. 21}
```

You can also apply a function to all elements of a list using **map**(). For example, you can get the sine value on a given interval in the following way

```
## rng3 = D:range(-3, 3, 0.3):map(math.sin)
## rng3[21]
0.14112
```

Sequential calling of **map**() is allowed, however, it should be remembered that the calculation will be performed at the moment of accessing the object index, i.e. it behaves “lazily”. For *range* object you can also apply functions by “redirecting” the flow via the “|” operator.

```
## rng4 = D:range(-3, 3, 0.3) | math.sin | math.exp
## rng4[10]
0.74414437789776
```

range() builds an immutable list, if you need a regular table that is suitable for writing, you can get it using the function **copy(table)**, which creates a deep copy for a given list. For example, the following code generates a table of a given number of zeros.

```
## zeros = D:copy(D:range(1,8) * 0)
## zeros
{ 0, 0, 0, 0, 0, 0, 0, 0, }
```

In general, to generate an array of arbitrary size filled with zeros, you can use the function **zeros(size1, [size2, ...])**.

```
## D:zeros(8)
{ 0, 0, 0, 0, 0, 0, 0, 0, }
```

In some cases, an object that returns a reference to a range of list data may be useful. It is generated by the function **ref(table, [start, end])**. The first and last indexes by default are the start and end of the table.

```
## src = {0, -1, -2, -3, -4, -5}
## ref = D:ref(src, 2, 4)
## #ref
3
## for i, v in ipairs(ref) do print(i, v) end
1    -1
2    -2
3    -3
```

This object allows you to change the data in the source table, both individually and in a range of values. To replace a subset of the list, you need to equate it to another reference.

```
## ref[1] = 10
## ref[2] = D:ref {20, 30} -- set sequence from index 2
## src
{ 0, 10, 20, 30, -4, -5, }
```

To simplify access to elements of a two-dimensional array, functions **row(table, index)** and **col(table, index)** are defined, which return references to a given row and column, respectively. These references mimic one-dimensional lists and allow both reading and assigning values. They also allow you to assign values to all elements at once by setting the field *data* equal to new values.

```
## t = {{1,2,3},{4,5,6}}
## a = D:col(t, 2)
## #a
2
## a[2]
```

```

5
## D:row(t, 1).data = {7, 8, 9}
## t[1]
{ 7, 8, 9, }

```

Library components *data* are implemented as ordinary functions that operate on Lua tables, so to perform a sequence of transformations, the result of one calculation must be explicitly passed to the next call. To simplify such transformations, the constructor **D**(*table*) is used, which “wraps” its argument in an object and allows access by indexes. The table itself can be obtained through the field *data*.

```

## t = {7,3,5,1,8}
## b = D(t)
## b:filter "x > 3" \
.. :sum()
20

```

Objects **row()**, **col()** and **ref()** have similar properties.

3.2.2 Filtering and transforming data

You can apply some criterion to all elements of the list and filter based on it. The criterion is a function that takes a list element as input, performs the check, and returns *true* or *false*.

Function **is**(*list*, *condition*) returns a list of weights: 1 if the element satisfies the criterion and 0 otherwise (**isNot**(*list*, *condition*) vice versa). The condition can be either a function or a string.

```

## a = {1, 2, 3, 4}
## D:is(a, odd)
{ 1, 0, 1, 0, }
## D:isNot(a, "x % 2 ~= 0")
{ 0, 1, 0, 1, }

```

Since the result is interpreted as “weights”, it can be used in some statistical functions of the module *data*, as well as for filtering data.

The function **filter**(*list*, *condition*) allows you to select specific elements from the list. The selection criterion can be specified by a list of weights of equal size (zeros are discarded, the rest is kept), a function returning *true* and *false*, or a string.

```

## b = D:filter(a, "x > 2")
## b
{ 3, 4, }
## D:filter(a, D:is(a, odd))
{ 1, 3, }

```

Functions **Map()** and **filter()** can be obtained from a more general function **gen**(*table*, *transformation*, [*condition*]), which builds a new list by applying a given transformation to each element of the original list if the condition is true (a function of

the form **f**(value, [index]) -> bool). Both the transformation and the condition can be either functions or strings.

```
-- sum of squares of odd elements of the list
-- transformation function (x) return x^2 end
-- condition function (x, i) return i % 2 ~= 0 end
## D(a):gen("x^2", "x, i -> i % 2 ~= 0"):sum()
10.0
```

If **Map()** applies a function to a single list, the **zip**(function, list1, list2, ...) applies a function of several arguments element by element to arguments and forms a list of results. The size of the result is equal to the length of the shortest of the lists. As before, the operation can be specified as a Lua function or a string.

```
## :set D zip
## c = zip("{x1-x2, x1+x2}", a, b)
## #c
2
## c[1]
{-2, 4, }
```

The zip function can make it easier to work with rows or columns of tables.

```
## t = {{1,2,3},{4,5,6}}
## D:col(t, 3).data = zip("x,y -> x^2+x*y", D:col(t, 1), D:col(t, 2))
## D:col(t, 3)
{ 3.0, 36.0, }
```

Another operation borrowed from functional programming is **reduce**(function, list, [x0]). It applies a function of two arguments successively to the next element of the list and the previous result, and returns the result. As before, the function can be specified using a string if performance is not required. x0 is 0.

```
## D:reduce("x,y -> x*y", a, 1)
24
```

You can change the order of list items using functions **sort**(table, criterion) and **reverse**(table). The first method sorts the elements, and the criterion can be either a function of two elements or a string. The second method reverses the order of the elements.

```
## D:reverse(a)
## a
{ 4, 3, 2, 1, }
## D:sort(a, "x,y -> x % 2 == 0 and y % 2 ~= 0")
## a -- divide even and odd numbers
{ 4, 2, 3, 1, }
```

The search for an element in a sorted list is performed by the function **binsearch**(table, value, [function]); it returns the index of the found element, as well as

its value. The optional third argument allows you to specify a function to extract data from the structure.

```
## D:sort(a, "x,y -> x < y")
## D:binsearch(a, 2)
2
```

3.3 Input, output and printing

Module *data* allows you to work with csv-files, namely, reading data into a Lua table and saving two-dimensional tables into a file. Reading is performed by the function **csvread**(*file*, *separator*), and the separator can be not only a comma (the default value), but also any other single character. The element is converted to a number, if possible. The file is written by the function **csvwrite**(*file*, *table*, *separator*).

```
## a = {      \
.. {1, 2, 3}, \
.. {4, 5, 6}, \
.. {7, 8, 9}}
## csvwrite('data.csv', a)
## b = csvread('data.csv')
## b[2][2]
5
```

There are different ways to display list elements. The most universal is to go through them in a loop and print them out element by element. To visualize two-dimensional lists, the function **md**(*list*, [*headers*, *function*]) can be used, which aligns the columns and uses the Markdown presentation type for better readability. You can optionally specify a list of headers. If you want to show individual columns or the result of applying some operation, then specify a function that transforms each row into a given form.

```
## D:md(a, {'c1', 'c2', 'c3'})
| c1 | c2 | c3 |
|----|----|----|
| 1  | 2  | 3  |
| 4  | 5  | 6  |
| 7  | 8  | 9  |
## D:md(a, nil, \
.. function (in) return{v[1]+v[2], v[1]-v[3]}end)
|----|----|
| 3  | -2 |
| 9  | -2 |
| 15 | -2 |
```

Combination of **csvread**() and **md**() can be used to read some file, perform operations on the strings and output the result in a table.

Functions **pack**(*data*) and **unpack**(*binary_string*) allow you to pack data into a binary string and perform the inverse transformation accordingly. The **pack**() argument

can be multidimensional arrays of numbers and strings, as well as the basic data types used in so/\ata, such as complex numbers, matrices, etc.

```
## str = D:pack(a)
## #str
38
## a2 = D:unpack(str)
## a2[1]
{ 1, 2, 3, }
```

3.4 Elements of statistics

This section describes functions that accept an array of data as input and, as a rule, return some of its characteristics. **max(list)** and **min(list)** find the maximum/minimum element of the list, as well as its index. The median is calculated with **median(list)**, the sum with **sum(list)**.

```
## a = {2, 0, 1, -3, 4, 1, 2, 1}
## v, ind = D:min(a)
## a[ind] == v
true
## D:max(a)
4
## D:median(a)
1
## D:sum(a)
8
```

A number of functions accept a list of weights as an optional argument. If a weight table is specified but a value for some element is missing, it is assumed to be 1. Thus, if the table is empty, all elements of the list are taken into account, and to exclude individual values, it is enough to set zeros in the corresponding indices. The functions for calculating the average (**mean(list, weight)**, **geomean(list, weight)**, **harmmean(list, weight)**) and standard deviation **std(list, weight)** work with weights. The calculation of the moment can also be performed taking into account the weights of the elements by the function **moment(degree, list, weight)**.

```
## D:mean(a)
1
## w = {[2]=0, [4]=0} -- skip 0, -3
## D:mean(a, w)
1.83333333333333
## D:geomean(a, w)
1.5874010519682
## D:harmmean(a, w)
1.4117647058824
## D:moment(2, a) - D:std(a)^2
0
```

The relationship between two lists can be estimated using the correlation function **corr**(list1, list2). The covariance matrix for an arbitrary number of lists can be found through **those**(table), whose argument is a table of source lists.

```
## p = D:range(1, 6)
## q = 1 - p
## D:corr(p, q)
-1.0
## D:cov {p, q}
 2.917 -2.917
-2.917  2.917
```

The analysis of element frequencies is performed by the function **freq**(list), which returns a dictionary: keys are the elements of the original list, values are the corresponding frequencies. Function **histcounts**(list, boundaries) calculates the distribution of elements over ranges. The bounds can be specified by a table of increasing numbers, or by a number of intervals, which defaults to 10. The function returns a number of elements in each range, as well as a list of the bounds.

```
## b = D:freq(a)
## b[1]
3
## sum, rng = D:histcounts(a, 3)
## sum
{ 1, 6, 1, }
## rng
{ -1.25, 2.25, }
```

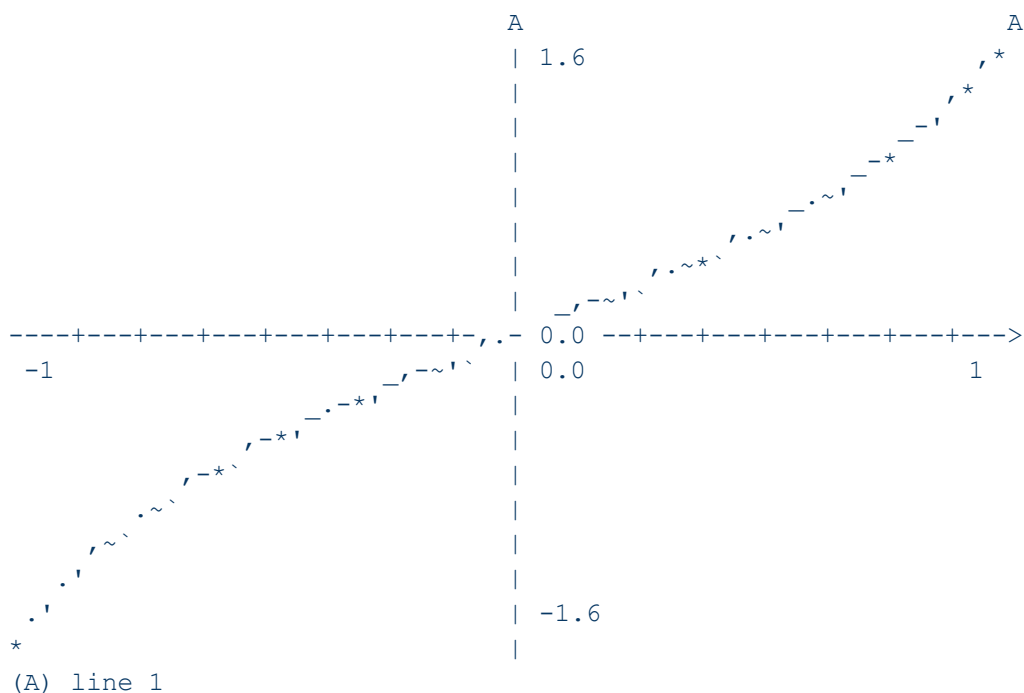

4 asciiplot: visualization in console

4.1 Introduction

When working in the console without using additional libraries, the only option for visualization is pseudo-graphics. Module *asciiplot* provides a set of functions for creating such diagrams. For sure pseudo-graphics impose restrictions on both the resolution of the “image” and the form of data presentation. However, in many cases this may be sufficient to draw the behavior of the function on a given interval. For better visual perception, you can use color output and Unicode symbols (see the section on setting up `so/\ata`).

As an example, let's depict the graph of the function **tan()**. To do this, you need to create an *asciiplot* object using the function **Ap()**, then call the method **plot()** and print the obtained result.

```
## fig = Ap()
## fig:plot(math.tan)
## fig
```



4.2 Canvas Options

By canvas we mean an array of strings that is displayed on the screen when printing an object *asciiplot*. The image size, value range, and axis position and type can be customized.

4.2.1 Image size

Call of the function **Ap()** without arguments uses default values stored in variables `Ap.WIDTH`, `Ap.HEIGHT`. In general, the constructor has the form **Ap**(width, height), where the desired number of characters is indicated in brackets. The following requirements are imposed on the sizes: the number must be odd and not less than 9 (the empirical “readability” limit of the graph). The function **scale**(coefficient) allows you to proportionally increase or decrease the canvas, which multiplies the width and height by a given number. If the argument is another canvas, the function borrows its dimensions.

```
## fig1 = Ap(65, 19) -- explicitly specified size
## fig2 = Ap()
## fig2:scale(fig1) -- set equal size
```

4.2.2 Setting up axes

Each axis (X - horizontal, Y - vertical, Z - for contours) has its own **set*(parameters)** function, where the argument is a table with fields

- *range* - a range of values defined by a list of two numbers. Used to construct functions, defaults to $\{-1, 1\}$.
- *fix* - a boolean flag indicating the possibility of changing the range when plotting a graph. For example, if the flag for the Y axis is *fix=false*, then the range of values will be scaled to fit all points in the given interval X, otherwise some values will be discarded. By default it is *false*.
- *view* - line 'min'/'mid'/'max', which specifies, in which part of the canvas to draw the axis (from left to right, from bottom to top). When *view=false* axis is not displayed. The default is 'mid'.
- *log* - boolean flag equal to *true*, if you want a logarithmic scale for this axis. The default is *false*.
- *size* - axis width (number of characters).

You can get the current values using the function **axes()** which returns a table of parameters for {x, y, z}.

```
## fig3 = Ap()
## src = fig1:axes()
## src.x.range
{ -1, 1, }
## fig2:setX(src.x); fig2:setY(src.y) -- copy axes properties
```

The axis labels are not related to units of measurement and are intended for ease of perception. They are displayed when the axis is divided into 2^n parts, so the division price is equal to $(x_{max} - x_{min})/2^n$. For the same reason, the intersection of the axes does not necessarily pass through 0. Numbers on the edges correspond to limits of axes.

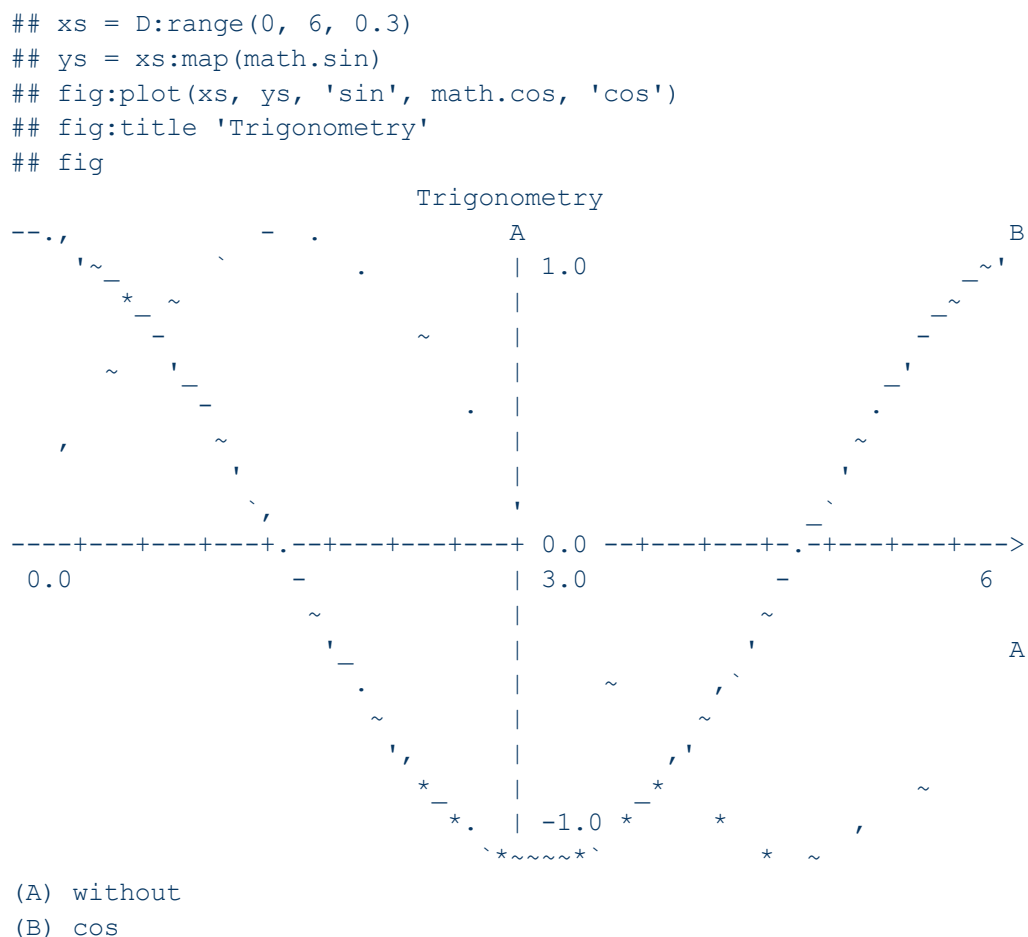
4.3 Types of graphs

4.3.1 Plot function

plot(G1, ... Gn) is a universal function for visualizing calculation results. Its call is similar to working with the Matlab or matplotlib (Python) functions of the same name. Each of the arguments G1, ... Gn is a sequence of elements of the form:

- list Y, name (if this is the only argument, name can be omitted)
- list X, list Y
- list X, list Y, name
- function
- function, name

Thus, the arguments are lists of numbers, functions, and strings. If the axis ranges are not pinned, the graph will scale to fit all the values given.

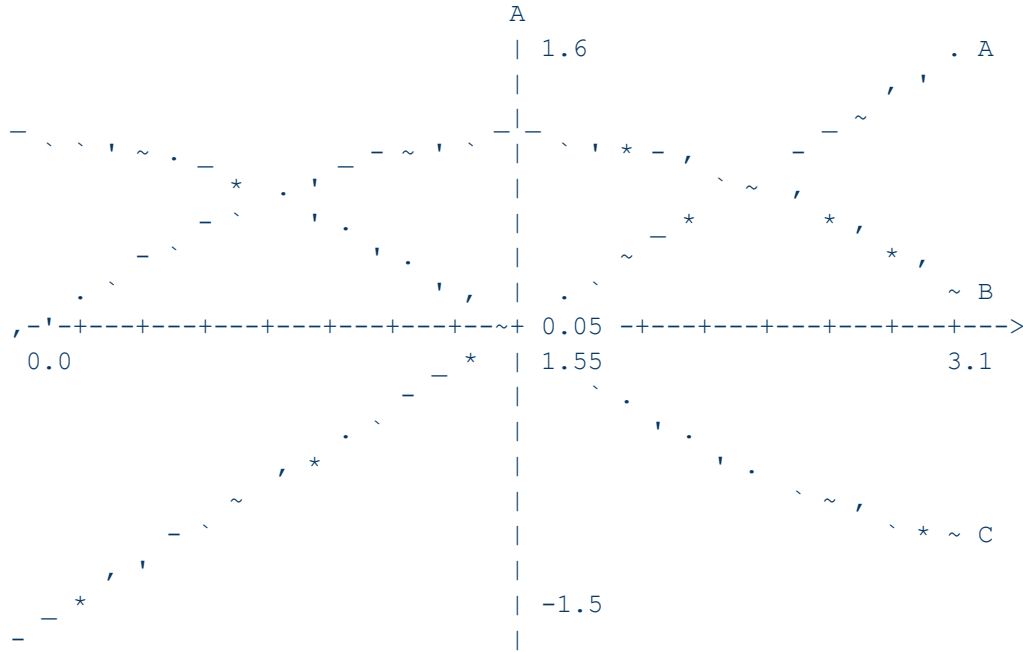


Function **title**(name) sets the name of the graph. You can also add or change the legend with the function **legend**(list[flag]). The argument can be a table with the names of the curves shown, or an “on”/“off” flag to enable or disable the legend display.

4.3.2 Tabular data

If the data is presented as a two-dimensional array, you can use the function **tplot**(table, parameters) to visualize it. In this case, the array is considered a list of strings, the default independent variable is the first element of the string, the rest are its functions.


```
## tbl = {}
## for x = 0, 3.1, 0.1 do \
..   tbl[#tbl+1] = {x, x-1.5, sin(x), cos(x)} \
.. end
## fig:tplot(tbl)
## fig:legend {'line', 'sin', 'cos'}
## fig
```

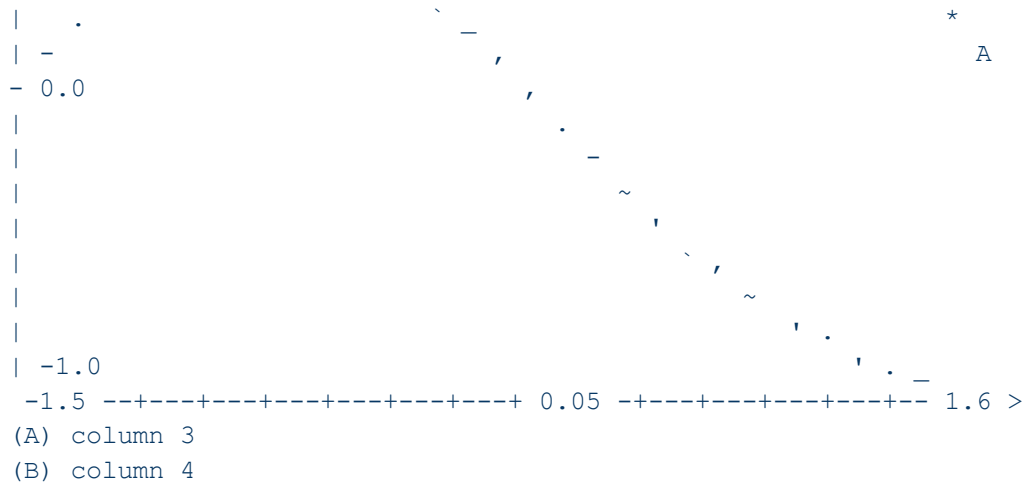


(A) line
(B) sin
(C) cos

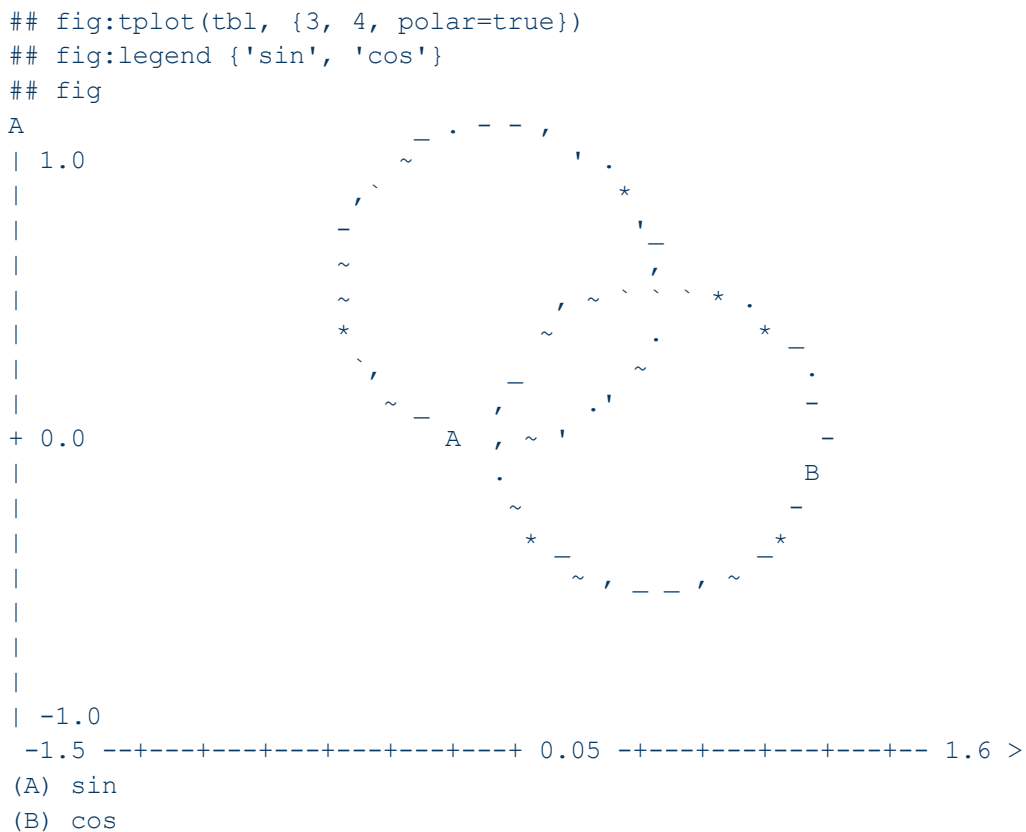
Function **tplot()** plots all columns of the table. If you want to visualize individual columns, their numbers must be listed in the table with options. In the field *x* you can specify the index of the column with independent variables. Field *sym* allows you to specify a string of characters that should be used to render data in columns.

```
## fig:setX {view='min'}
## fig:setY {view='min'}
## fig:tplot(tbl, {3, 4, x=2})
```





Function **tplot()** also allows you to plot graphs in polar coordinates, i.e. the independent variable is considered an angle, and its functions are radii. To do this, you need to set the option *polar=true*.



4.3.3 Column chart

Some of the data are more conveniently represented by column charts. Given the specifics of pseudo-graphical display, such charts are "turned" on their sides, i.e. the columns are arranged from top to bottom, with the argument displayed on the left and the value on the right. The function **bar(tableX, tableY)** performs the construction, the arguments are lists of numbers, where the first argument can be omitted or contain a list of strings.

```

## fig:setX {view='mid'}
## fig:bar(D:col(tbl, 1), D:col(tbl, 4))
## fig
0.0      ===== 1.0
0.2      ===== 0.980066
0.4      ===== 0.921060
0.6      ===== 0.825335
0.8      ===== 0.696706
1.0      ===== 0.540302
1.2      ===== 0.362357
1.4      ===== 0.169967
1.6      ===== -0.02919
1.8      ===== -0.22720
2.0      ===== -0.41614
2.2      ===== -0.58850
2.4      ===== -0.73739
2.6      ===== -0.85688
2.8      ===== -0.94222
3.0      ===== -0.98999

```

Function **bar()** tries to fit data into the given size by y. If the table is long or short, you can explicitly set the height of the field to be rendered.

```

## t1 = D:ref(tbl, 1, 11)
## fig:setX {view='min'}
## fig:setY {size=#t1}
## tx, ty = D:col(t1, 4), D:col(t1, 1)
## fig:bar(tx, ty)
## fig
-1.5 ===== 1.0
-1.4 ===== 0.995004
-1.3 ===== 0.980066
-1.2 ===== 0.955336
-1.1 ===== 0.921060
-1.0 ===== 0.877582
-0.9 ===== 0.825335
-0.8 ===== 0.764842
-0.7 ===== 0.696706
-0.6 ===== 0.621609
-0.5 ===== 0.540302

```

4.3.4 Contours

Module *asciipLOT* cannot construct isometric images. However, it allows you to display projections of a function of two arguments on a plane in the form of lines of equal height. The method used for this is **contour**(function, options). The options table defines the number of levels (*level*) used for visualization, as well as the projection type (*view*='XY'|'XZ'|'YZ').

```

## fig:setX {range={-5,5}}
## fig:setY {range={-5,5}}
## function saddle (x, y) return x*x - y*y end

```

```
## fig:contour(saddle)

X-Y view
A
      b      a      a      b
      b aa | 5 aa b
d      c      b      bb      c      d
      d      bb      c      cc      d
      c      cc      cc      d
      d      ccc      ccc      d
-----e-----d-----cccc+ 0.0 -----d-----e-->
-5      d      ccc      0.0 c      d      5
      d      cc      cc      d
      c      cc      c
      d      bb      bb      d
d      c      b      b      c      d
      b aa | -5 aa b
      b      a
(Z1) a(-16.50) b(-8.00) c(0.50)
(Z2) d(9.00) e(17.50)
```

It should be noted that in the case of symmetrical objects, some of the lines may coincide.

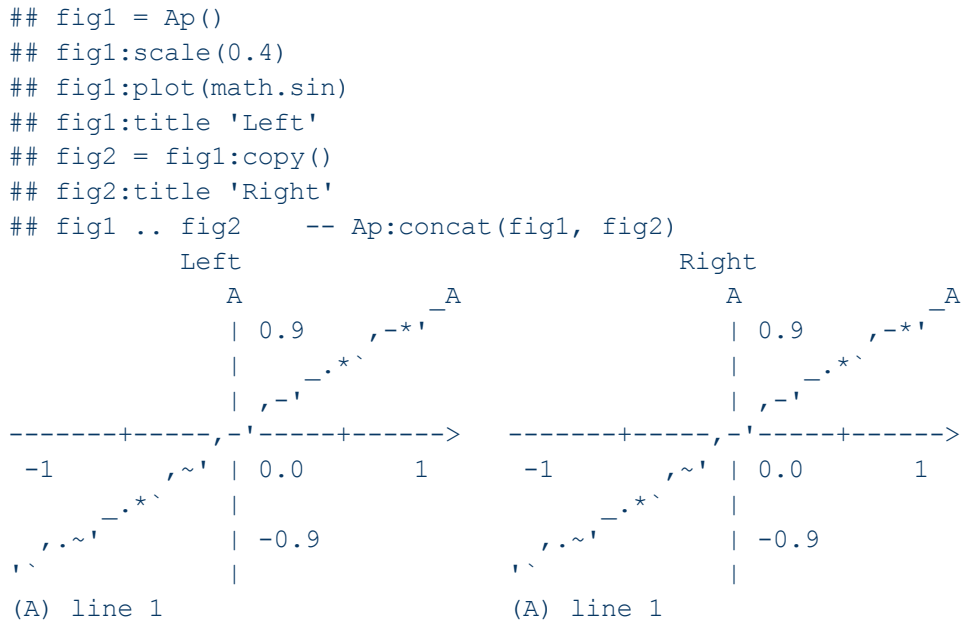
```
## fig:contour(saddle, {view='XZ', level=3})
## fig

X-Z view
A
      b      bb      bb
      bb      bb      bb
a bb      bb      bb
      aa bb      bb aa
      aa bbb      bbb aa
      aa bbb      bbb aa
      aaa bbb      bbb aaa
      aaa bbbbbb | bbbbbb aaa
-----aaa-----bbbbbbbbbb 0.5 bbb-----aaa----->
-5 aaaaa | 0.0 aaaaa 5
      aaaaaaaaaaaaaaaaaa
      |
      +
      |
      |
      | -25.0
      |
(Y1) a(-2.50) b(0.00) c(2.50)
```

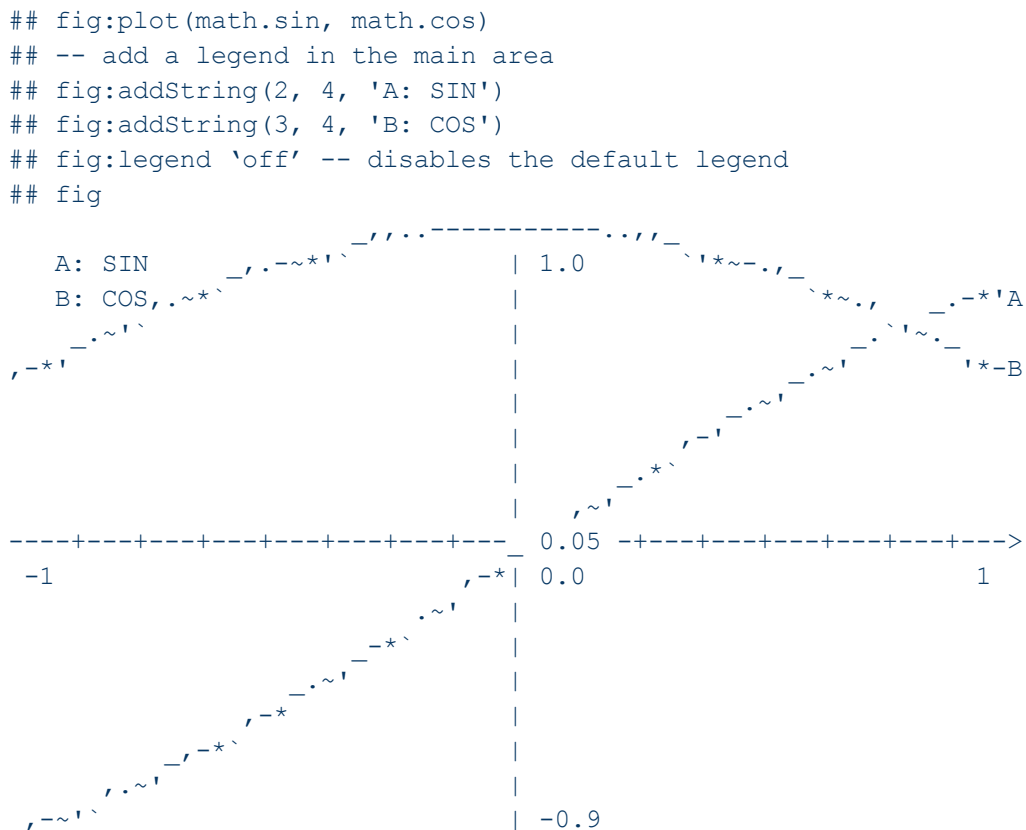
4.4 Other operations

A number of methods are used to manipulate previously created charts. First, you can get a full copy using **copy()**. Secondly, you can horizontally concatenate the graphs

with the function **concat**(graphic1, graphic2) or the operator '!. In this case, the heights of both graphs must match. The result of the function is text (a string object).



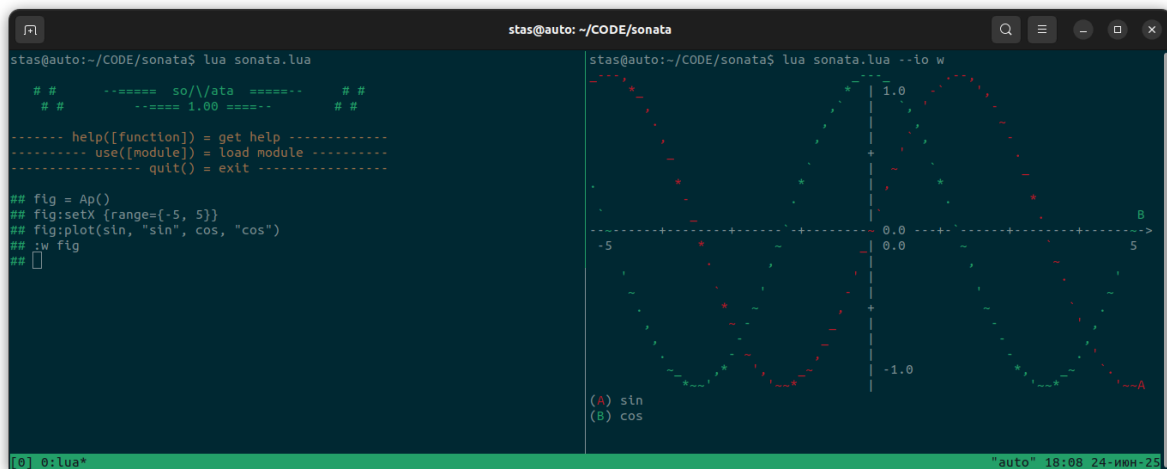
You can plot elements on a chart manually using the following functions: **addPoint**(x, y, symbol) marks the point (x,y) given symbol, **addPose**(row, column, symbol) behaves similarly, but places the symbol in a specific position on the canvas, **addLine**(x1, y1, x2, y2, symbol) draws a line between two points, **addString**(row, column, text) places text in the specified line. The function **reset**() allows you to clear the canvas.



Sometimes it is useful to qualitatively evaluate the contents of the matrix. Function **stars(matrix, criterion)** prints spaces and asterisks instead of numbers based on the specified Boolean condition. By default, a check for equality to zero is performed.

```
## Ap:stars(Mat:eye(3))
* |
* |
* |
```

Graphs take up relatively much space, so it is convenient to print them in a separate window. To do this, use the command “:w *expression*”, and in the second terminal the interpreter must be started in the “--io w” mode (see Appendix 4).



5 matrix: matrix operations

5.1 Introduction

Simply put, a matrix is a two-dimensional array of numbers for which algebraic operations are defined. To access a matrix element, you must specify 2 numbers: the row and column numbers.

In the module *matrix* matrices are represented by two-dimensional tables, the number of rows is returned by the function **rows()**, columns with **cols()**. The base constructor is **Mat(table)**, which transforms a given table into a matrix. Indexing starts with 1 as in mathematics.

```
## a = Mat { \
..  {1, 2}, \
..  {3, 4}}
## a
1  2
3  4
## a:cols()
2
## a:rows()
2
## a[2][1]
2
## a[4][5] -- outside the matrix
0
```

Standard size operation (“#”) is not specified for the matrix, so the number of rows and columns must be used to traverse the array. If you try to access an element outside the allowed range, 0 will be returned.

The method **zeros(rows, [columns])** allows to create a zero matrix; if the number of columns is not specified, a square matrix is constructed. In this case, an empty table is formed, for uninitialized elements 0 will be returned. To construct a matrix, all elements of which are equal to a given number, you can use the function **fill(rows, columns, value)**, the default value is 1. The identity matrix (i.e. the zero matrix with ones on the main diagonal) can be formed by the function **eye(rows, [columns])**. Function **D(elements, offset)** generates a square matrix with the given diagonal elements; the second optional argument specifies the offset above or below the main diagonal.

```
## b = Mat:zeros(2)
## b[1][1] = 3
## b
3  0
0  0
## Mat:fill(1, 3)
1  1  1
## Mat:eye(2, 3)
```

```

1  0  0
0  1  0
## Mat:D({1,2}, 1)
0  1  0
0  0  2
0  0  0

```

5.2 Basic operations

5.2.1 Arithmetic operations

The basic arithmetic operations defined for matrices are addition, subtraction, multiplication, raising to an integer power, and unary minus. It is also possible to check matrices for equality and inequality. Addition and multiplication with scalar numbers are performed element by element.

```

## a + b
4  2
3  4
## a - b
-2  2
3  4
## a * b
3  0
9  0
## a^3
37  54
81  118
## a == Mat({1,2},{3,4})
true
## a + 1
2  3
4  5

```

Matrix division is not defined, multiplication by the (pseudo-) inverse matrix must be used.

If the matrix contains one element after all calculations, it is converted to a number. You can use the constructor **Mat()** to ensure that matrix operations are performed.

5.2.2 Methods

A number of functions are defined only for square matrices. Among them is the calculation of the determinant **det()** and the inverse matrix **inv()**. Additional minor for an element with coordinates *i, j* can be found using the function **minor(i, j)**. Method **eig()** returns two matrices, the elements of the first are the eigenvectors of the matrix, the elements of the second are the corresponding eigenvalues located on the diagonal.

```
## a:det()
```

```

-2
## a:inv()
-2    1
1.500 -0.500
## a:minor(1, 1)
4
## p, q = a:eig()
## p
0.416  0.825
0.909 -0.566
## q
5.372      0
0 -0.372
## p * q * p:inv()
1.000  2.000
3.000  4.000

```

The following methods are universal, they can be applied to any type of matrix. Function **rank()** defines the rank of the matrix, **norm()** finds the Euclidean norm. The function **diag()** allows one to express the diagonal elements as a column vector, and with the help of **tr()** you can find the trace of the matrix, i.e. the sum of the diagonal elements. The pseudoinverse of the matrix is calculated by the function **pinv()**.

```

## b:rank()
1
## a:norm()
5.4772255750517
## a:tr()
5
## c = Mat {{2, 3}}
## ci = c:pinv()
## c * ci
1.0

```

5.3 Reference objects

A number of functions return objects that behave like matrices, but do not store the data themselves, instead they refer to another object. The advantages of such objects are the speed of creation and the relatively small size of the memory consumed. The disadvantages include the overhead of calculating indexes, as well as the impossibility of modification independent of the original. To increase the processing speed and make the object independent, use the method **copy()**, which creates a deep copy of the original matrix.

Function **T()** returns the transposed matrix (reference), and **H()** - transposed complex conjugate.

```

## c = Mat{{4,5},{6,7}}
## d = c:T()
## d
4  6

```

```

5  7
## d[1][2] = 1
## c
4  5
1  7

```

There is no specific method for selecting a data range, but this functionality is implemented when calling the matrix as a function, i.e. using parentheses. In this case, you need to pass two lists as arguments, indicating the ranges of rows and columns, respectively. In the general case, the range contains 3 numbers: the first index, the last index, and the step. A missed step is replaced by one, the last index by the end of the list, and the first index by the beginning, so an empty list means all rows (or columns). A negative start or end means counting from the end, a negative step means changing the order of elements. If we are interested in a specific column or row, instead of a range, we need to pass the corresponding number. If both arguments are numbers, the corresponding element will be returned, and negative indices can also be used.

All referenced objects contain a field *data*, but for a range of values its use is especially relevant. When reading this field, the object to which the reference is made will be returned; when writing, the elements of the object to the right of the equal sign are copied to the corresponding elements of the reference object.

```

## c = Mat:eye(3)
1  0  0
0  1  0
0  0  1
## c({1,2},{1,2}).data = a
## c
1  2  0
3  4  0
0  0  1
## c(1, {})
1  2  0
## c(-1,-1)
1

```

Function **reshape**(rows, columns) allows you to change the size of the matrix. The movement occurs line by line, unnecessary elements are discarded, and zeros are added instead of the missing ones.

```

## c:reshape(4,2)
1  2
0  3
4  0
0  0

```

To combine matrices, functions **hor**(list) and **ver**(list) are used, the first performs horizontal concatenation, the second - vertical, the arguments are lists of matrices.

```

## Mat:ver { \
..  Mat:hor {b, a}, \

```

```

..    Mat:hor {a:T(), b} \
.. }
3  0  1  2
0  0  3  4
1  3  3  0
2  4  0  0
## a..b
1  2  3  0
3  4  0  0

```

Horizontal concatenation of two matrices can be done using the standard ellipsis operator, which results in the formation of a new matrix rather than a reference.

5.4 Vector operations

A vector is a matrix that consists of one row or column. Accordingly, standard arithmetic operations and most functions defined for matrices are applicable to vectors. To simplify the creation of a column vector, the function **V**(elements) is provided.

```

## v1 = Mat:V {1,2,3} -- column vector
## v1
1
2
3
## v2 = Mat{{4,5,6}} -- row vector
## v2
4  5  6
## v2 * v1
32
## v1 * v2
4  5  6
8  10 12
12 15 18

```

Some inconvenience is that to access an element, you need to specify 2 indices, as for a regular matrix. The operator “()” partially solves this problem, but it only allows you to read the value. To simplify working with vectors, another reference object is introduced, which is formed by the function **vec()**. For three-element vectors, this also makes it possible to use indices *x*, *y*, *z*.

```

## v1(3)    -- reading value
3
## p = v1:vec()
## q = v2:vec()
## p[3]     -- reading value
3
## p[1] = 0 -- assigning a new value
## p
0  2  3
## v1      -- the original matrix also changes
0

```

```

2
3
## #p
3
## q.z      -- access by "name"
6

```

Unlike the previous objects, this type is not a matrix, but it does contain a number of specific methods, such as vector (**cross**), external (**outer**) and scalar (**dot**) products, the formation of a skew-symmetric matrix (**skew**), vector normalization **normalize()**, as well as the calculation of the norm (**norm**) for different metrics (l_1 , l_2 , l_∞).

```

## p:dot(q)
28
## p:cross(q)
-3
12
-8
## p:norm('l1')
5
## q:normalize()
## q
0.455  0.569  0.683
## q.data = p    -- copying vector elements
## q
0  2  3

```

5.5 Transformations

Gaussian transform (function **rref()**) is most often used to solve systems of linear equations. In this case, the matrix passed in the argument is modified.

```

## m = a..Mat:V{5,6}
## m
1  2  5
3  4  6
## m:rref()
1  0  -4
0  1  4.500

```

The LU decomposition of a square matrix is performed by the function **lu()**, which returns the lower and upper triangular matrices (L, U), as well as the permutation matrix P.

```

## L,U,P = a:lu()
## L*U == P*a
true
## L
1  0
0.333  1
## U

```

```

3      4
0  0.667

```

The function used for QR decomposition is **qr()**, and the number of rows must be no less than the number of columns. The result of the work is a matrix with orthonormal columns **Q** and the upper triangular **R**.

```

## Q,R = a:qr()
## Q*R
1.000  2.000
3.000  4.000
## Q
-0.316  -0.949
-0.949   0.316
## R
-3.162  -4.427
0      -0.632

```

The Cholesky decomposition is performed by the function **chol()**, which returns the lower triangular matrix of *nil* when the original matrix is not positive definite.

```

## m = Mat {{1,3},{3,10}}
## L = m:chol()
## L
1  0
3  1
## L*L:T()
1  3
3  10

```

The singular value decomposition is carried out by the function **svd()**, which returns a matrix with singular values on the main diagonal, as well as matrices of left and right singular vectors.

```

## U,B,V = a:svd()
## B
5.465      0
0  0.366
## U*B*V:T()
1.000  2.000
3.000  4.000

```

The matrix exponential can be found using the function **exp()**.

```

## a:exp()
51.98  74.75
112.1  164.1

```

The function **map(function, [row, column])** allows one to apply a function of one argument to each element of a matrix. Additionally, here you can set a condition tied to the row and column number. If you need to apply the function to several matrices, use **zip(function, ...)**.


```
## f1 = function (x) return 2*x end
## a:map(f1)
2 4
6 8
## f2 =function (x,y) return x*y end
## Mat:zip(f2, a, b)
3 0
0 0
```

Thus **zip()** allows you to perform element-wise matrix multiplication and similar operations.

5.6 Other operations

The Kronecker product and sum can be calculated using the methods **kron**(*matrix*) and **kronSum**(*matrix*) respectively.

```
## a:kron(a)
1 2 2 4
3 4 6 8
3 6 4 8
9 12 12 16
## a:kronSum(b)
4 0 2 0
0 1 0 2
3 0 7 0
0 3 0 4
```

Function **vectorize()** transforms a matrix into a column vector, unlike the function **reshape()** the elements grouped not by rows, but by columns. Usually it is used together with the Kronecker product.

```
## (a*b):vectorize()
3
9
0
0
## Mat:eye(2):kron(a) * b:vectorize()
3
9
0
0
```

6 complex: complex numbers

6.1 Introduction

The module *complex* contains functions necessary for working with complex numbers. The constructor has the form **Z**(*real_part*, *imaginary_part*), by default both arguments are zero. If the number is specified in polar (exponential) form $e^{i\alpha}$, you can use the constructor **E**(*a*). The imaginary unit can also be obtained using **i**([*factor*]) or constant **I**.

```
## Z(1, 2)
1+2i
## Z:i()
0+1i
## 2*Z:E(PI/4)
1.414+1.414i
```

Complex numbers are immutable, i.e. functions do not modify an existing object, but create a new one.

6.2 Basic operations

Complex numbers support basic arithmetic operations, the result is converted to an ordinary number if the imaginary part is zero. If you need to perform operations on the result that are specific to complex numbers, you can pass it through the constructor.

```
## a, b = Z(1,2), Z(1, -2)
## a + b
2
## Z(a + b):im()
0
## a ^ b
17.94-9.855i
```

Functions **re()** and **im()** return the real and imaginary parts of a number, respectively. The absolute value of a number can be found using the function **abs()**, and the argument (angle) for exponential notation using **arg()**. The complex conjugate of a number can be obtained with the function **conj()** or the bitwise negation operator “~”.

```
## a:abs()
2.2360679774998
## a:arg()
1.1071487177941
## ~a
1-2i
```

6.3 Functions

Almost all mathematical functions of the module *main* (except **atan2()**) are defined for **complex**, and the call **fn(x)** will be more versatile than **x:fn()**, since in the first case *x* can be any numeric type of so/\ata, while in the second - only a complex number.

```
## a:exp()
-1.131+2.472i
## exp(a)    -- required implementation of exp inside the function
-1.131+2.472i
## a:sin()
3.166+1.960i
```

When loading *complex* module redefinition of functions **sqrt()** and **log()** occurs for working with negative and complex numbers.

```
## sqrt(-1)
0+1i
## log(-1)
0+3.142i
```

7 polynomial

7.1 Introduction

Polynomials in the module *polynomial* are represented by a list of coefficients from the highest to the lowest degree. That is, the expression $x^2 + 2x + 1$ can be represented as **Poly** {1, 2, 1}, where **Poly()** is a constructor. However, in the resulting object the coefficients are arranged in such a way that the index of the element corresponds to its degree.

```
## a = Poly {1, 2, 1}
## a
1 2 1
## a[0] -- free coefficient
1
```

Sometimes it may be more convenient to use the “algebraic” notation of a polynomial as a sum of powers, especially when there are zero coefficients. For this purpose, you can use the object returned by the function **x()**. The function **str(symbol)** returns a string with the algebraic notation of the polynomial, by default the variable is denoted by *x*. The value of the polynomial for a given argument *t* can be obtained by the function **val(t)**, or simply the operator “()”. A copy of the polynomial is built by the function **copy()**.

```
## x = Poly:x()
## b = 2*x^3 + 3*x^2 + 1
## b
2 3 0 1
## b:str()
2*x^3+3*x^2+1
## b(2) -- value at x=2
29
```

7.2 Basic operations

Basic arithmetic operations can be performed with polynomials: addition, subtraction, multiplication, division, remainder from division, and raising to a natural power. Equality testing is also possible.

```
## p1 = Poly {1, 1}
## p2 = p1^2 + 1
## p2
1 2 2
## p1 + p2
1 3 3
## p2 - p1
1 1 1
## p2 / p1
```

```

1 1
## p2 % p1
1
## p2 / p1 == p1
true

```

The function **der()** allows you to calculate the derivative, and the integral is **int(p0)**, where the argument is the value of the free coefficient.

```

## p2:der():str()
2*x+2
## p2:int(7):str()
0.333*x^3+x^2+2*x+7

```

The roots of a polynomial are calculated by the function **roots()**, which returns a list. The real roots come first, then the imaginary ones. The inverse operation, constructing a polynomial from known roots, is performed by the function **R(roots)**.

```

## rs = p2:roots()
## rs
{-1-1i, -1+1i, }
## Poly:R(rs)
1 2 2

```

Function **char(matrix)** constructs the characteristic polynomial for a given matrix.

```

## m = Mat{{1,2},{3,4}}
## cp = Poly:char(m)
## cp:str()
x^2-5*x-2
## r = cp:roots()
## r[1] * r[2]
-2.0
## m:it()
-2

```

If the calculation results in only a free coefficient, it is converted to a regular number. To save the result as a polynomial, you can pass it to the constructor **Poly**, by analogy with matrices and complex numbers.

7.3 Approximation

7.3.1 Polynomials

The approximation functions can be divided into two groups: some of them follow the given points, and others find an approximation. A representative of the first group is **lagrange(xs, ys)**, which constructs the Lagrange polynomial, its arguments are lists x and y values at nodal points.

```

## p3 = Poly:lagrange({0,1,2}, {3,5,1})
## p3:str()

```

```
-3.0*x^2+5.0*x+3.0
## p3(1)
5.0
```

If for some point x_0 the value of the function and the first derivatives are known, the value in the neighborhood x_0 can be calculated using the Taylor polynomial, which is constructed by the function **taylor**(x_0, f, f', \dots).

```
## x0 = PI/4
## f, f1, f2 = sin(x0), cos(x0), -sin(x0) -- разложение sin
## p4 = Poly:taylor(x0, f, f1, f2)
## p4:str()
-0.353*x^2+1.26*x-0.066
## p4(PI/2)
1.0443776422176
```

Function **fit**($xs, ys, degree$) calculates the best approximation of the original set of points by a polynomial of a given degree using the least squares method.

```
## xs = D:range(0, 10)
## ys = xs:map(function(x) return p2(x)+0.5*math.random() end)
## Poly:fit(xs, ys, 2):str()
0.981*x^2+2.15*x+2.12
```

7.3.2 Splines

If you do not need to find a curve that passes through all the given points, you can split the sequence of nodes into pairs and connect them with curves separately. Function **lin**($xs, ys, [y_0, y_N]$) performs interpolation of the given dependence with straight lines. Out of range xs extrapolation occurs using the obtained functions by default, but constant values can be used if you set y_0 and y_N . The value at a given point is calculated in the same way as for a regular polynomial, but two numbers are returned, the first is the result, the second is the index of the polynomial in the list. When calculating, you can specify this index, then the calculation will be faster.

```
## pp1 = Poly:lin({0,1,2}, {3,4,1}, 0)
## pp1(1.5)
2.5
## pp1(-1)
0
```

Function **spline**(xs, ys) performs interpolation using cubic splines.

```
## pp2 = Poly:spline({0,1,2}, {3,4,1})
## pp2(1.5)
2.875
```

There are no arithmetic operations defined on splines, but functions **copy()**, **der()** and **int()** can be applied to them.

```
## pp1 = Poly:lin({0,1,2}, {3,4,1}, 0)
```

```
## pp1:der():val(1.5)  
-3.0  
## pp1:int():val(1.5)  
3.625
```

8 numeric: numerical methods

8.1 Introduction

Module *numeric* is a collection of functions for the numerical solution of such problems as finding the root of an equation, integrating a function, etc. The operation of the module is regulated by variables:

- *numeric.TOL*- required accuracy of the solution;
- *numeric.SMALL*- minimum step of some algorithms;
- *numeric.NEWTON_MAX*- the maximum number of iterations of Newton's method;
- *numeric.INT_MAX*- maximum number of integration iterations.

When loading a module, a variable *INF* is added to the environment, which is a link to *math.huge*.

8.2 Functions

8.2.1 Root Search

Finding the root of a function on a given interval $[a, b]$ executes the method **solve**(*function*, *a*, *b*), which returns the found value or raises an error if the function values at the extreme points have the same sign. The search is performed by the chord method. For a differentiable function, the solution can be found by Newton's method **newton**(*function*, *x0*), in this case it is enough to specify only one initial value *x0*, preferably in the vicinity of the root.

```
## -- root sin(x) = 0, chord method
## Num:solve(sin, 0.5*PI, 1.5*PI)
3.1415926535898
## -- Newton's method
## Num:newton(sin, 0.8*PI)
3.1415927368139
```

Newton's method can also be used for a function that takes a vector as input and returns a scalar.

```
## Num.TOL = 1E-4
## -- argument - vector
## fn = function (v) return v(1)^2 + v(2)^2 end
## Num:newton(fn, Mat:V{1, 1})
4.09E-03
4.09E-03
```

8.2.2 Mathematical analysis

This section contains functions that are related to integral and differential calculus. The method **lim**(*function*, *argument_limit*, [*positive*]) allows you to estimate the

value of the limit of a function, the second argument is a number t , to which the independent variable tends. The optional third argument is *true* if $x \rightarrow t_+$, and *false* when $x \rightarrow t_-$. The limit can be infinite (*math.huge*). If the assessment cannot be completed, a warning is displayed.

```
## Num:lim(function (x) return sin(x)/x end, 0)
0.999999999999983
## Num:lim(exp, -INF)
0.0
## Num:lim(sqrt, 0) -- defaults to 0-
Warning: limit not found
-in
## Num:lim(sqrt, 0, true) -- now 0+
3.1622776601684e-05
```

The method **der**(*function*, *x*) estimates the value of the derivative at the point *x*.

```
## Num:der(sin, PI/4)
0.70709499613245
## Num:der(exp, 1.0)
2.7183271333827
```

Method **int**(*function*, *a*, *b*) is used to find a definite integral on an interval $[a, b]$. In some cases, the method works for infinite limits or situations when the function is not defined at the extreme points.

```
## Num:int(sin, 0, PI)
2.0000165910479
## fn = function (x) return 1/((1+x)*sqrt(x)) end
## Num:int(fn, 0, INF) -- pi
3.1394196640198
## fn = function (x) return exp(-x*x) end
## Num:int(fn, -INF, INF) -- square root of pi
1.772391289196
```

8.3 Ordinary differential equations

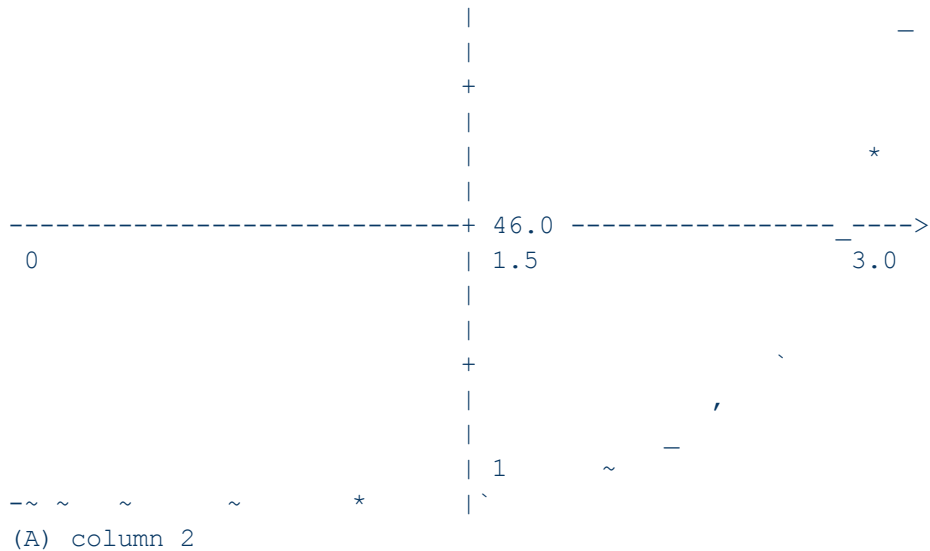
For the numerical integration of ordinary differential equations, a method **ode**(*function*, *interval* *t*, *initial* *y*, [*options*]) has been implemented, which takes as input the function, the integration interval and the value at the initial point of this interval. The options will be discussed in more detail below. The method returns a list of pairs, where the first element is the independent variable, the second is the value at the point.

```
# -- equation y' = t*y, t in [0, 3], y0 = 1
## ys = Num:ode(function (t,y) return t*y end, {0, 3}, 1)
## fig = Ap()
## fig:tplot(ys)
## fig
```

A

A

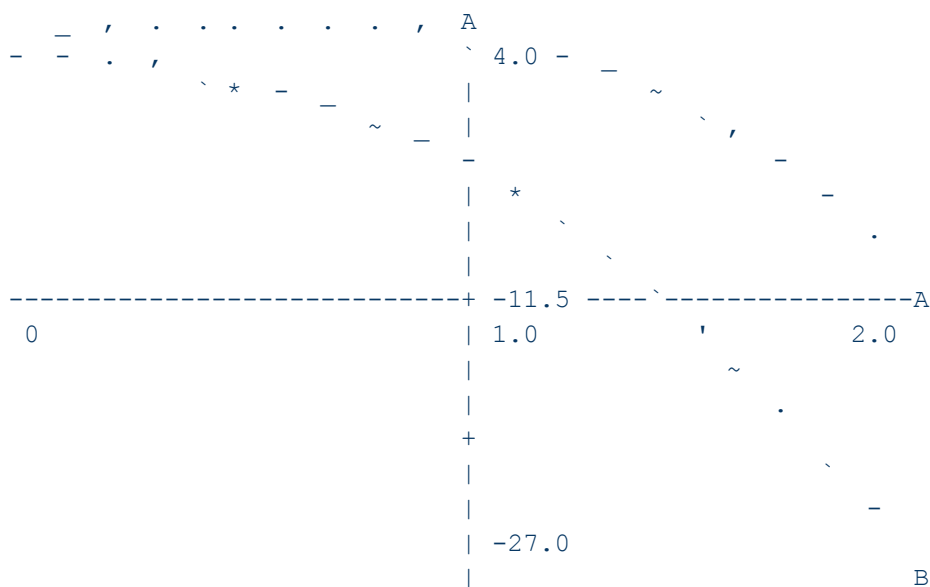
| 91.0



Function **ode()** by default changes the integration step to achieve the required accuracy. If you need to perform the calculation with a constant step, you can specify it in the options table, in the field *dt*. The calculation termination condition can be specified as a function that receives the current intermediate results as input and returns *true* to complete. This function must be equal to the field *exit*.

ode() can work with functions whose input and output are vectors, i.e. integrate a system of differential equations. As an example, consider a second-order equation $y'' - 2y' + 2y = 1$, which, taking into account the replacement $x_1 = y, x_2 = y'$ can be represented as a system $x_1' = x_2, x_2' = 1 + 2x_2 - 2x_1$.

```
## fun = function (t, x) return Mat:V{x(2), 1+2*x(2)-2*x(1)} end
## -- calculation with a constant step of 0.1
## xs = Num:ode(fun, {0, 2}, Mat:V{3,2}, {dt=0.1})
## -- for display we transform {t, Vec{x1, x2}} into {t, x1, x2}
## fig:tplot(xs:flat())
## fig
```



```

(B) column 3
## -- interrupt on request
## fun = function (t, y) return -y end
## cond = function (ys) return ys[#ys][2] < 0.1 end -- about zero
## ys = Num:ode(fun, {0, 100}, 1, {exit=cond})
## ys[#ys] -- last point
{ 2.35, 0.095415291087094, }

```

This example uses the function **flat()**, defined specifically for the ODE solution table. It replaces the vector in each row with a list of its elements, which is required for drawing via **tplot()**.

9 extremum: optimization methods

9.1 Introduction

The optimization problem for the function $f(x)$ is to find such x at which the function value reaches a minimum or maximum. Most often, minimization is performed, and to find the maximum, one can consider the function $-f(x)$. Module *extremum* contains implementations of algorithms for finding extrema, and also allows solving some related optimization problems.

9.2 Finding the minimum of a function

The extremum of a function of one variable can be found using the method **minimum1D**(*function*, *start*, *end*, *parameters*), which returns the minimum point and the function value at that point. The second and third arguments limit the search interval, the parameter table allows you to set additional search settings: *method* - algorithm (golden ratio by default, or 'Brent'), *b* - starting point of the search, *dfun* - derivative of the function, can be used in Brant's algorithm.

```
## fun = function (x) return x*x end
## xm, fm = Ex:minimum1D(fun, -10, 5)
## print( xm, fm )
3.9508921150952e-06    1.5609548505121e-11
## df = function (x) return 2*x end
## xm, fm = Ex:minimum1D(fun, -10, 5, {method='Brent', dfun=df})
## print( xm, fm )
0.0    0.0
```

For ease of use, the function **maximum1D**() is also defined with the same set of arguments.

Method **minimum**(*function*, *p*, *parameters*) searches for an extremum for a function that depends on a vector of numbers. The parameter table defines the type of search *method* ('Powel' by default, can be 'simplex'), and the derivative *dfun* for Powell's algorithm. The second argument *p* is the starting point or, in the case of a simplex, a matrix with search boundaries. As in the one-dimensional case, the result is the minimum point found and the value at this point, and the function **maximum**() is defined with the same parameters.

```
## foo = function (and) return (y(1)-1)^2 + (y(2)-2)^4 end
## p0 = Mat:V {5,4}
## xm, fm = Ex:minimum(foo, p0)
## xm
1
2
-- search area in the form of column vectors
## pp = Mat{{5,3}, {7,-9}, {-7,-4}}:T()
## xm, fm = Ex:minimum(foo, pp, {method='simplex'})
```

```
## xm
1.000
2.000
## fm
9.8929233647719e-16
```

9.3 Linear programming

Linear programming consists of minimizing the product Cx with additional restrictions: $A_{in}x \leq b_{in}$, $A_{and}x = b_{and}$, $A_lx \geq b_l$. To solve the linear programming problem, the simplex method is used, which is implemented in the function **linprog**(matrix, restrictions). The initial system of equations is given as a matrix C, the constraints are presented in a table with the following fields: Au, bu - upper boundary parameters, Ae, be - additional equality, Al, bl - lower boundary. Constraints are optional.

```
## -- C*x -> min, Au*x <= bu, Ae*x == be, Al*x >= bl
## C = Mat{{-0.4, -0.5}}
## Au = Mat{{0.3, 0.1}}; bu = Mat(2.7)
## Ae = Mat{{0.5, 0.5}}; be = Mat(6)
## Al = Mat{{0.6, 0.4}}; bl = Mat(6)
## xm, fm = Ex:linprog(C, {Au=Au, bu=bu, Ae=Ae, be=be, Al=Al, bl=bl})
## fm
-5.4
## xm
6.000
6.000
```

9.4 Nonlinear approximation of a function

Approximation of data by a nonlinear function requires the use of optimization methods, therefore it is also implemented in the module *extremum*. Method **fit**(function, parameters, x_points, y_points) takes the desired function as input $f(x, parameters)$, the parameters are a dictionary with initial values.

```
## -- function for approximation
## foo = function (x, t) return t.A*sin(t.B*x) + t.C end
## t0 = {A=2, B=0.5, C=-1}
## -- arrays of points
## xs, ys = {}, {}
## for i = 1, 40 do xs[i] = i*0.1; ys[i] = foo(xs[i], t0) end
## -- initial values
## init = {A=4, B=2, C=-2}
## -- approximation
## param, err = Ex:fit(foo, init, xs, ys)
## param
{ C = -1.0,
  B = 0.5,
  A = 2.0, }
```

9.5 Others optimization methods

9.5.1 Annealing method

The annealing method allows minimizing the “system energy” by gradually lowering its “temperature”. The algorithm is implemented as a function **annealing**(*exercise*), the argument of which is a table with the following fields:

- *energy* - function of estimating the energy of the system;
- *update* - a function that returns the modified state of the system for testing;
- *heat* - initial state of the system;
- *T* - initial temperature, optional;
- *alpha* - temperature update coefficient at each step ($T_{i+1} = \alpha T_i$, $0 < \alpha < 1$), optional;
- *loop* - number of attempts at each temperature, optional.

As an example, we will solve the problem of eight queens that should not attack each other. The update function will change the columns for two random queens. We will estimate the energy as the number of attacks of all figures on each other.

```
## -- initial state
## pos = {1, 2, 3, 4, 5, 6, 7, 8}
## -- update function
## tweak = function (x) \\  
..   local y, m, n = {}, nil, nil  
..   for i = 1, #x do y[i] = x[i] end -- make a copy  
..   repeat  
..     m, n = math.random(#y), math.random(#y)  
..     until m ~= n -- apply to different lines  
..     y[m], y[n] = y[n], y[m]  
..   return and  
.. end  
..  
## -- energy function  
## energy = function (x) \\  
..   local s = 0  
..   for i = 1, #x-1 do  
..     for j = i+1, #x do -- search on one diagonal  
..       if math.abs(j-i) == math.abs(x[j]-x[i]) then s = s+1 end  
..     end  
..   end  
..   return s  
.. end  
..  
## -- solution  
## xm, fm = Ex:annealing { \\  
..   energy=energy,  
..   update=tweak,  
..   init=pos,  
..   T=20,  
..   alpha=0.95,  
..   loop=2
```

```
.. }  
..  
## table.concat(xm, ' ') -- row numbers  
3 1 7 5 8 2 4 6
```

10 random: generators

10.1 Introduction

Module *random* contains various random number generators, as well as functions based on them. Calling the constructor **Rand()** returns a random number between 0 and 1 generated by the standard Lua generator for uniform distribution. Function **new()** creates a new uniformly distributed random number generator that is not associated with *math.random*. This generator works a little slower, but it allows you to get independent random numbers in different parts of the code. The function **seed(number)** allows you to set the generator core. The kernel is an integer and can be used to ensure repeatable results across different calls. **seed()** without an argument resets the core to a "random" value.

```
## Rand:seed(11)
## Rand()
0.44766006064378
## rnd = Rand:new()
## rnd()
0.84616688955534
## rnd1 = Rand:new():seed(3)
## rnd2 = Rand:new():seed(3)
## rnd1() == rnd2() -- identical sequences
true
## rnd() == rnd1() -- different sequences
false
```

All module functions can be called with the default generator (via **Rand**), as well as with custom generators.

10.2 Distributions of random numbers

The random module provides generators for the following random number distributions:

- **norm**(*m*, *p*) - Gaussian distribution;
- **rayleigh**(*p*) - Rayleigh distribution;
- **poisson**(*l*) - Poisson distribution;
- **cauchy**(*m*, *p*) - Cauchy distribution;
- **exp**(*l*) - exponential distribution;
- **gamma**(*a*, *b*) - gamma distribution;
- **logistic**(*m*, *p*) - logistic distribution;
- **binomial**(*p*, *N*) - binomial distribution;
- **int**(*from*, *to*) - uniform distribution of integers.

As an example, we will construct a histogram for a normal distribution and compare it with the theoretical value $\exp(-0.5(x - \mu)^2/\sigma^2)/(\sigma \sqrt{2\pi})$.


```

## rnd:bytes(10)
s+{/axj)*/
## t = {2, 1, 7, 9, 0, 4}
## Rand:choice(t)
9
## for i, v in Rand:ipairs(t) do print(i, v) end
4    9
3    7
5    0
1    2
2    1
6    4
## Rand:shuffle(t)
## t
{ 4, 1, 0, 9, 2, 7, }

```

11 bigint: long integers

11.1 Introduction

Module *bigint* was originally introduced to support arbitrary-length integers in various number systems, but over time the focus has shifted to integer functions. However, arbitrary length and number system conversion are still supported.

Object *bigint* created using a constructor **Int(value)**, the argument can be an integer, a string, or a table. Using numbers is the simplest way, but it provides a limited range of values. Using a string is more flexible. If the number is in decimal or starts with '0x' or '0b', the base of the number system can be omitted and the numbers are written sequentially without separators. In general, the base is placed at the end and separated by a colon, and the signs of the number are written in decimal notation and separated by a comma. Any punctuation mark (except a colon) can be used instead of a comma, and separators can be alternated for greater clarity. A table allows you to list the signs of a number and also specify the base of the number system *base* and a sign *sign*=±1. In this case, the k-th (from right to left) element must have index k, so the number is written in reverse order.

```
## a = Int(42)
## b = Int '1,2,3:8'
## b -- value is displayed in decimal form
83
## b1 = Int{3,2,1,base=8,sign=1} -- same as b
## b1
83
## big = Int '123`456`789`321' -- decimal, group digits
```

Standard arithmetic and Boolean operations are defined for integers. The function **float()** allows you to convert a value to a regular number, the sign of a number (1 or -1) can be obtained using **sign()**.

```
## a + b
125
## a - b
-41
## a * b
3486
## b / a
1
## b % a
41
## b ^ a
39928213565868246933433190168987688667882096577412654059296188512948149
4196023689
## ANS:float()
3.9928213565868e+80
## a > b
```

```

false
## b == b1
true
## a:sign()
1

```

During arithmetic calculations the object *bigint* can be simplified to a regular Lua number. To prevent this from happening, the result can be passed to the constructor **Int()**.

11.2 Functions

The modulus of a number is returned by the function **abs()**. A random number from 0 to a given value is generated by the method **random()**.

```

## c = Int(-50):random()
## c
-41
## c:abs()
41

```

The function **isPrime()** allows you to check if a number is prime. By default, iterating over the multipliers is used, if you add the argument “Fermat”, the Fermat method will be used. Function **factorize()** factorizes a number into prime factors.

```

## b:isPrime()
true
## a:isPrime('Fermat')
false
## a:factorize()
{ 2, 3, 7, }

```

The greatest common divisor is found by the function **gcd**(*n1*, *n2*, ...), the least common multiple is **lcm**(*n1*, *n2*, ...).

```

## Int:lcm(Int(2), Int(6), Int(7))
42
## Int:gcd(a, a*2, a/2)
21

```

11.3 Number systems

Function **digits(reason)** allows you to get a representation of a number in a given number system (decimal by default). The result is a table where *i*-th digit is in *i*-th position. This table can be passed to the constructor **Int()** to convert back to a number. The shift operators << and >> are defined for these objects, the function **group(length)** allows you to combine characters into groups for better readability when printing.

```

## d = (a * a):digits(6)
## d

```

```

12100:6
## d >> 2
121:6
## d:group(3)
12`100:6

```

11.4 Elements of Combinatorics

The factorial can be found by the function **F()**, to speed up the calculation of the ratio of factorials, a function **ratF(a, b)** is defined. The number of permutations from n by k defines the function **P(n, k, repetitions)**, the number of combinations is **C(n, k, repetitions)**, in both cases the optional third argument determines whether repetitions are taken into account. The subfactorial is calculated by the function **subF()**, double factorial is **FF()**.

```

## a:F()      -- a!
1405006117752879898543142606244511569936384000000000
## Int:ratF(a, a/2)
27500101936481280675682713600000
## Int:P(Int(10), Int(3))
720
## Int:C(Int(10), Int(3))
120
## Int(10):subF()  -- !10
1334961
## Int(10):FF()    -- 10!!
3840

```

12 rational: rational numbers

12.1 Introduction

Rational numbers are numbers that can be represented as a ratio of two integers. From the point of view of software implementation, they store both values: the numerator and the denominator. You can create a rational number using the constructor **Rat**(numerator, denominator), if the second argument is omitted, it is considered equal to 1. These objects are “immutable”.

For rational numbers, the basic arithmetic operations are defined: addition, subtraction, multiplication, division, raising to an integer power.

```
## use 'rational'
## a = Rat(1,2)
## b = Rat(2,3)
## a + b
7/6
## b - a
1/6
## a * b
1/3
## a / b
3/4
## a^3
1/8
```

You can also perform comparisons and equality checks. However, keep in mind that in Lua the expression “number == object” always returns *false*.

```
## a == b
false
## a < b
true
## eq(a, 0.5)  -- a == 0.5
true
```

To display a fraction in mixed form, you must set the MIXED flag.

```
## Rat.MIXED = true
## a + b
1 1/6
```

12.2 Functions

There are some specific functions for rational numbers: **num()** returns numerator, denominator returns **denom()**, **float()** converts object to floating point number.

```
## a:num()
1.0
## a:denom()
```

```

2.0
## a:float()
0.5

```

The inverse operation, i.e. converting a floating point number to a rational number with a given precision (by default 0.001) performs the function **from**(*number*, *precision*).

```

## Rat:from(math.pi, 1E-5)
355/113
## 355 / 113
3.141592920354

```

12.3 Continued fractions

so/\ata can transform continued fractions of the form $a_0 + 1/(a_1 + 1/(a_2 + 1/...))$ into rational numbers and back. In this case, the continued fraction is represented as a table of positive coefficients, where the number (a_0) corresponds to index 0. The function **toCF**() generates this table of coefficients, and **fromCF**() performs the inverse transformation.

```

## c = b:toCF()
## c
{0+L1+L2} -- 1/(1+1/2)
## c[1]
1
## Rat:fromCF(c)
2/3

```

Sign *L* symbolizes “1/(...)” to avoid confusion with the simple sum of fractions.

13 quaternion: operations with quaternions

13.1 Introduction

Quaternions are an extension of the idea of complex numbers, they consist of four elements, one of which is real and the remaining three are imaginary. The module *quaternion* supports basic arithmetic operations and functions, but focuses on the use of the unit quaternion to represent orientation in space. In this sense, the imaginary part can be thought of as the direction of the axis of rotation, and the real part as a measure of rotation about this axis.

Quaternion constructor **Quat** has arguments that may be a list {*real w, imaginary x, imaginary y, imaginary z*}, dictionary {*w = real w, x = imaginary x, y = imaginary y, z = imaginary z*} or a combination of them, missing elements are replaced by zero. Among the arithmetic operations, division is not defined; instead, multiplication by the inverse quaternion can be used. Raising to a fractional power is specified only for unit quaternions. The function **normalized()** allows you to get a unit quaternion.

```
## a = Quat {1, 2, 3, 4}
## b = Quat {w=5, y=2,}
## a + b
6+2i+5j+4k
## a - b
-4+2i+1j+4k
## a * b
-1+2i+17j+24k
## a^3
-86-52i-78j-104k
## c = a:normalized() -- |c| is 1
## c^0.5 -- sqrt
0.769+0.237i+0.356j+0.475k
```

A quaternion is an immutable object. Individual components can be accessed using functions **x()**, **y()**, **z()**, **w()**. The result of arithmetic operations can be an ordinary number if at some stage the imaginary part is equal to zero. To be sure to get a quaternion, you can place the calculation result in the constructor.

13.2 Functions

The modulus of a quaternion can be found using the function **abs()**, conjugate value via **conj()**, the opposite is **-inv()**. Some standard functions are defined, such as **exp()**, **log()**.

```
## a:abs()
5.4772255750517
## a:conj()
1-2i-3j-4k
## a * b:inv()
```



```

0.379+0.621i+0.448j+0.552k
## d = a:log()
## d:exp()
1.000+2i+3.000j+4k

```

13.3 Orientation

The unit quaternion is one way to represent the orientation of a body in space, along with rotation matrices, Euler angles, and rotation about a given axis. The following functions are defined to transform these forms into quaternions and back: **toRot()** and **fromRot(matrix)**, **toRPY()** and **fromRPY(roll, pitch, yaw)**, **toAA()** and **fromAA(angle, axis)**. The function **rotate(vector)** allows you to rotate a vector using a quaternion. You can interpolate a value between two unit quaternions using **slurp(Q1, Q2, part)**, where the third argument is a number from 0 to 1.

```

## Quat:slerp(c, b:normalized(), 0.5)
0.670+0.220i+0.555j+0.441k
## m = c:toRot()
## m
-0.667    0.133    0.733
 0.667   -0.333    0.667
 0.333    0.933    0.133
## Quat:fromRot(m)
0.183+0.365i+0.548j+0.730k
## r, p, y = c:toRPY()
## Quat:fromRPY(r, p, y)
0.183+0.365i+0.548j+0.730k
## angle, axis = c:toAA()
## axis
{ 0.3713906763541, 0.55708601453116, 0.74278135270821, }
## Quat:fromAA(angle, axis)
0.183+0.365i+0.548j+0.730k
## c:rotate {1, 0, 0}
{ -0.666666666666667, 0.666666666666667, 0.333333333333333, }

```

14 special: functions

14.1 Introduction

Special functions usually include functions that are common in mathematical physics and are not expressed explicitly through elementary functions. Some of them are implemented in the module *special*.

14.2 List of functions

14.2.1 Gamma functions

The gamma function can be calculated using **gamma**(*x*), the logarithm is calculated through **gammaln**(*x*). The incomplete (lower) gamma function is called via **gammp**(*degree*, *x*), and the conjugate function to it (upper) through **gammq**(*degree*, *x*).

```
## Spec:gamma(-1.5)
2.3632718012074
## Spec:old(10)
12.801827480082
## Spec:gammp(3, 1.5)
0.19115316863236
## Spec:gammq(3, 1.5)
0.80884683136764
```

Associated with incomplete gamma functions are the error function **erf**(*x*) and an additional error function **erfc**(*x*).

```
## Spec:erf(0.5)
0.52049990772324
## Spec:erfc(0.5)
0.47950009227676
```

14.2.2 Beta functions

The value of the beta function can be found using **beta**(*z1*, *z2*). The natural logarithm is returned by the function **betaln**(*z1*, *z2*). The incomplete beta function is calculated using **betainc**(*x*, *z1*, *z2*), Where $x \in [0, 1]$ - upper limit of integration.

```
## Spec:beta(3, 7)
0.0039682539682521
## Spec: pay(3, 7)
-5.5294290875119
## Spec:betainc(0.5, 3, 7)
0.91015624999996
```

14.2.3 Bessel functions

Bessel functions of the first (J) and second (Y) kind are calculated using **besselj**(degree, x) and **bessely**(degree, x). The corresponding modified functions of the first (I) and second (K) kind are defined as **besseli**(degree, x) and **besselk**(degree, x).

```
## Spec:besselj(2, 5)
0.046565118685798
## Spec:bessely(2, 5)
0.36766287898699
## Spec:besseli(2, 5)
17.505615012117
## Spec:besselk(2, 5)
0.0053089437352436
```

14.2.4 Other functions

The integral exponential function can be calculated using **expint**(degree, x). The Dawson integral is defined by **dawson**(x).

```
## Spec:expint(2, 3.5)
0.0058018938267817
## Spec:dawson(2.5)
0.22308368068315
```

15 graph: graph operations

15.1 Introduction

Module *graph* is designed to work with directed and undirected graphs without loops and multiple edges. Nodes can be arbitrary Lua objects, while edge weights must be numbers. An empty graph is created by the constructor **Graph()**. Using additional parameters presented in the form of a table, you can determine the type and method of graph generation. The list of nodes is returned by the method **nodes()**, and the edges with **edges()**.

```
## g1 = Graph()
## g1
Graph {}
## g1:nodes()
{ }
## g1:edges()
{ }
## g2 = Graph {dir=true, C=3} -- oriented ring
## g2:nodes()
{ n1, n2, n3, }
## g2:edges()
{ n1 -> n3, n2 -> n1, n3 -> n2, }
## g2
Digraph {n1,n2,n3}
## #g2 -- same as g2:size(), number of nodes
3
```

15.2 A generation

As mentioned above, some specific graph types can be generated based on constructor parameters:

- O is a graph without edges;
- K is a complete graph;
- C - ring;
- P - chain.

For each type, you can specify the number of nodes or a table of names. By default, nodes are named 'n1', 'n2', etc. If necessary, the common part of the name can be replaced using the parameter *name*.

To create a directed graph, you need to set the flag *dir*, which can be combined with other parameters.

```
## a = Graph {K=3}
## a:nodes()
{ n1, n2, n3, }
## a:edges()
{ n1 -- n2, n1 -- n3, n2 -- n3, }
## b = Graph {P=3, name='b'}
```

```
## b:nodes()
{ b1, b2, b3, }
## b:edges()
{ b2 -- b1, b3 -- b2, }
## c = Graph {O={'one', 'two', 'three'}} -- node names
## c:nodes()
{ one, two, three, }
```

The graph can be randomly filled with edges using functions **rand(number_of_edges)** and **randp(probability)**. In the first function, you need to specify the desired number of edges, in the second - the probability of adding an edge during the generation process. Both functions delete the found edges first.

```
## d = Graph {O=4}
## d:rand(4)
## d:edges()
{ n1 -- n2, n1 -- n3, n4 -- n2, n2 -- n3, }
## d:randp(0.5)
## d:edges()
{ n1 -- n4, n4 -- n2, n2 -- n3, }
```

15.3 Basic operations

15.3.1 Modification

The main function for adding elements is **add(node1, node2, weight)**. If only the first argument is specified, the node will be added. If both nodes are specified, the corresponding edge will be added, for which you can specify a weight using the third argument (default 1). The same function allows you to update the weight. Deleting a node or edge is performed by the function **remove(node1, node2)**. Similarly, one argument removes a node, two - an edge.

To work with a group of elements, use functions **addNodes(nodes)** and **addEdges(edges)**. The first method takes a list of nodes as input, the second - a list of edges, where each edge is represented by a table with the names of the corresponding nodes and (optionally) a weight.

```
## g = Graph()
## g:add 'a'
## g:add('b', 'c')
## g:add('c', 'd', 2)
## g
Graph {b,c,d,a}
## g:edges()
{ b -- c, c -- d, }
## g:remove('c','d')
## g:edges()
{ b -- c, }
## g:remove('b')
## g
Graph {c,d,a}
```

```

## g:addNodes { 'x', 'y', 'z' }
## g:nodes()
{ x, a, z, c, d, y, }
## g:addEdges { \
.. {'a', 'x'}, \
.. {'b', 'y'}, \
.. {'c', 'z', 3}}
## g:edges()
{ b -- y, a -- x, c -- z, }
## g:nodes()
{ b, x, a, z, c, d, y, }

```

You can find out the weight of the edge using the function **edge(nodes)**, and check the belonging of a node to a graph through **has(node)**. For a directed graph, the function **nin(node)** returns a list of nodes corresponding to the incoming edges, **nout(node)** - corresponding to the outgoing ones, for an undirected graph both functions return a list of adjacent nodes.

```

## g:has 'x'
true
## g:edge {'c', 'z'}
3
## g:nin 'a'
{ x, }

```

15.3.2 Export

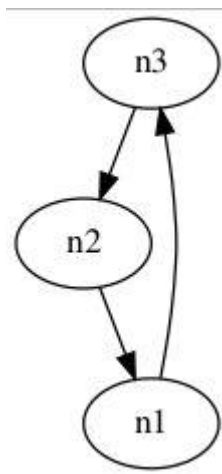
To visualize graphs, you can use the Graphviz program. If it is installed and the dot utility is configured to call from the console, you can save the graph to an svg file using the function **toSvg(file name)**.

```

## g2:toSvg 'example'

```

The call will generate a file *example.svg*.



A text description of the graph in dot utility notation can be generated by the function **dot([file name])**. If the file name is not specified, the text will be displayed on the screen.

```
## g2:dot()
digraph {
  n3 -> n2;
  n1 -> n3;
  n2 -> n1;
}
```

15.3.3 Other functions

Function **matrix()** returns the adjacency matrix of the graph and the list of nodes corresponding to the rows. A copy of the graph can be obtained using the method **copy()**. The function **concat(list)** allows you to combine several graphs into one, in the case of two objects, the concatenation operator can be used. The operation can only be applied to graphs of the same type (directed or undirected).

```
## m, ns = a:matrix()
## m
0  1  1
1  0  1
1  1  0
## ns
{ n3, n2, n1, }
## a1 = a:copy()
## a1 == a
true
## a == b
false
## Graph:concat{a,b}:edges()
{ b3 -- b2, b2 -- b1, n2 -- n3, n1 -- n2, n1 -- n3, }
## ab = a..b
## ab:edges()
{ b3 -- b2, b2 -- b1, n2 -- n3, n1 -- n2, n1 -- n3, }
```

The method **components()** allows decomposing a graph into individual connectivity components, which returns a list of found subgraphs.

```
## ab:components()
{ Graph {b3,b2,b1}, Graph {n3,n2,n1}, }
```

15.4 Algorithms

15.4.1 Checking properties

A number of methods are designed to test graph features: **isComplete()** - is it complete, **isConnected()** - is it connected, **isDirected()** - is it oriented, **isEuler()** - does it contain an Euler cycle, **isTree()** - is it a tree, **isWeighted()** - contains edges with weights different from 1. The functions return *true* or *false* depending on the result of the check.

```
## a:isConnected()
true
## a:isEuler()
```

```

true
## a:isTree()
false

```

15.4.2 Finding the way

To find a path between two nodes, use the function **search**(*start*, *thread*, *method*). The method is a string with the name of the algorithm: 'bfs' - breadth-first search, 'dfs' - depth-first search, 'dijkstra' - shortest path search using Dijkstra's algorithm.

```

## p = Graph {K=6}
## p:search('n1', 'n3', 'bfs')
{ n1, n3, }
## p:search('n1', 'n3', 'dfs')
{ n1, n5, n6, n3, }
-- random weights from 1 to 10
## for _, e in ipairs(p:edges()) do \
..   p:add(e[1], e[2], math.random(1, 10)) end
## p:search('n1', 'n4', 'dijkstra')
{ n1, n6, n4, }
## p:edge {'n1', 'n4'} -- direct path
8
## p:edge {'n1', 'n6'} + p:edge {'n6', 'n4'} -- found path
5

```


16 units: units of measurement

16.1 Introduction

Module *units* is designed to perform calculations using units of measurement and converting one unit to another based on the established rules. You can create an object using the constructor **U**(*value* , *unit_of_dimensions*). If the first argument is missing, it is considered equal to one. In the case of a dimensionless quantity, the second argument can be an empty string. Basic arithmetic operations and comparison rules are defined for objects. It should be remembered that you can only add, subtract, and compare quantities whose units of measurement are the same or can be reduced to each other.

```
## a = U(2, 'm')
## b = 1 * U('s') -- same as 1*U('s') or U(1, 's')
## a / b
2 m/s
## a * b
2 m*s
## 0.5 * a / b^2
1 m/s^2
## a > b
ERROR ./matlib/units.lua:255: Different units!
```

16.2 Conversion of units of measurement

16.2.1 List of rules

Conversion rules in the module *units* can be divided into implicit and explicit. The former include transformations based on fractional and multiple prefixes, which are defined in the table **U.prefix**. By default, these are standard prefixes used in science (kilo-, micro-, etc.). If, when comparing two units of measurement, the program detects that they differ in a prefix known to it, it tries to convert the quantities to a common dimension.

Explicit transformations are performed based on rules stored in the table **U.rules**. The rule is an expression of the type

$$1 \text{ 'A' } = x \text{ 'B' }$$

which is written in the form

$$\text{U.rules['A']} = \text{U}(x, \text{'B'})$$

In this case, the value 'A' must be a simple unit of measurement that does not contain arithmetic operations.

The list of rules and prefixes can be displayed using the function **print()**.

16.2.2 Functions

To determine a numerical value for given units of measurement, you need to refer to the object *units* as a function, passing a string with the required dimensions as an

argument. If the argument is omitted, the current value will be returned. The units of measurement can be obtained using the function **u()**.

```
## a = U(2, 'm/s')
## a() -- current value
2
## a:u() -- current units of measurement
m/s
## -- to get km/h we define 2 intermediate transformations
## U.rules['h'] = 60 * U'min' -- U(60, 'min')
## U.rules['min'] = 60 * U's' -- U(60, 's')
## a 'km/h' -- same as a('km/h')
7.2
## a 'm/min'
120.0
## a 'dm/ms'
0.02
```

Function **convert(units)** returns a new *units* object, and not just a numerical value. A copy can be created using the method **copy()**.

```
## b = a:convert('km/h')
## b
7,200 km/h
## c = b:copy()
## c == a
true
```

17 const: collection of constants

17.1 Introduction

Module `const` is a collection of constant values used in various fields of science (mainly in physics for now). The value can be obtained using dot notation, with the section name specified. As befits constants, these values are immutable.

```
## C.phy.G -- gravitational constant
6.672041e-11
## C.math.phi
1.6180339887499
## C.math.phi = 123
ERROR: ./matlib/const.lua:55: Constants are immutable!
## :set C.phy.mu0 as mu0 -- aliases can be defined
## mu0
1.2566370614359e-06
```

Physical quantities usually have units of measurement. The constants presented in this module can be converted into *units* objects, it is enough to add the suffix `'_U'` to the name.

```
## e = C.phy.e_U -- electron charge
## e
1.60E-19 C
## e 'nC' -- nC
1.602189246e-10
```

17.2 User constants

For permanent use, it is better to add constants to the corresponding tables of the file `matlib/const.lua`. But if you need to get an immutable object while working with the program, you can use the functions **`add(name, value, [units])`** and **`remove(name)`**. The first one adds the constant to the list, the second one removes it.

```
## C:add('aa', 4)
## C:add('bb', 5, 'm/s')
## C.aa
4
## C.bb
5
## C.bb_U
5 m/s
## C.aa = 2
ERROR: ./matlib/const.lua:55: Constants are immutable!
## C:remove('aa')
true
```

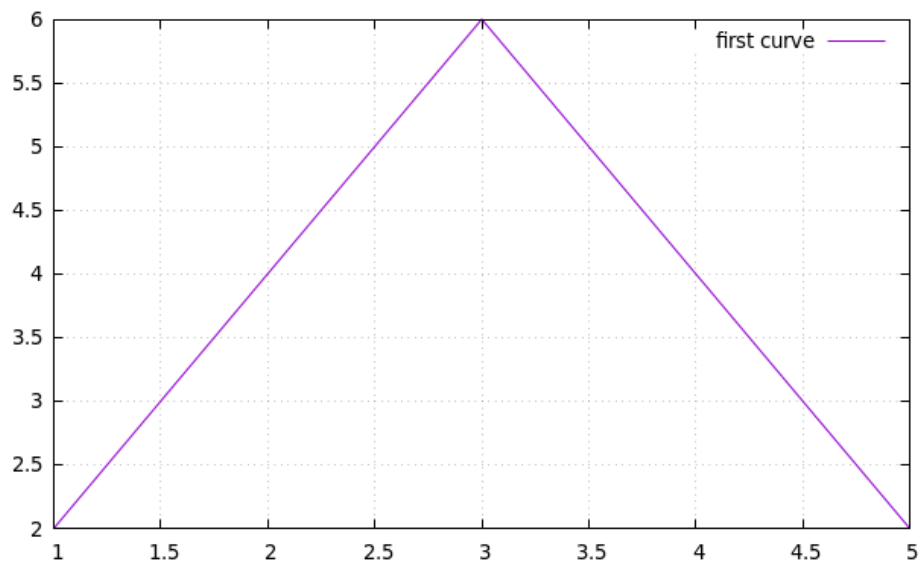
18 gnuplot: plotting graphs in GNUplot

18.1 Introduction

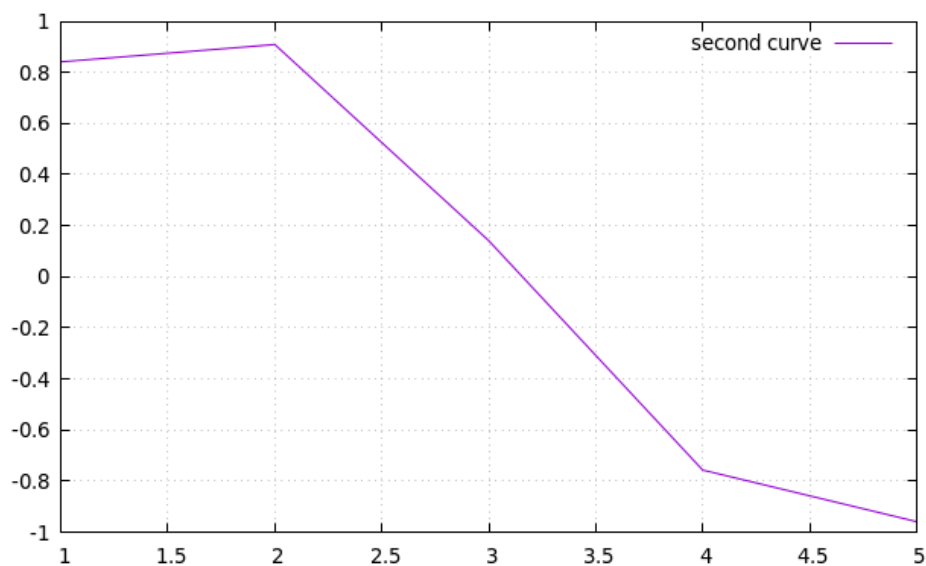
External graphical tools can be used to visualize data. One of them is Gnuplot. A module of the same name has been implemented to work with this program.

The canvas object is formed using the constructor **Gp()**, however, most functions can be called without being tied to a specific object.

```
## t1 = {1, 2, 3, 4, 5}  
## t2 = {2, 4, 6, 4, 2}  
## Gp:plot(t1, t2, 'first curve')
```



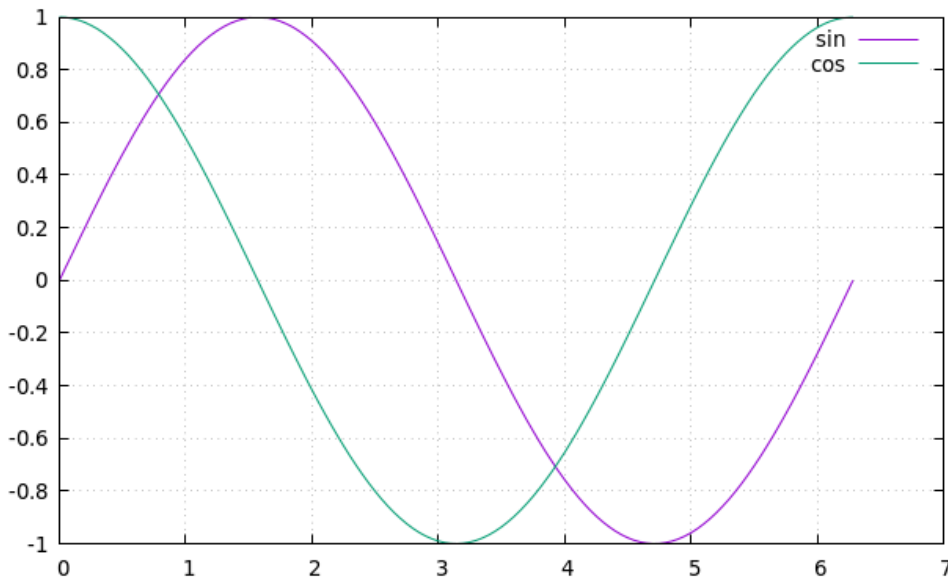
```
## Gp:plot(t1, sin, 'second curve')
```



18.2 Main functions

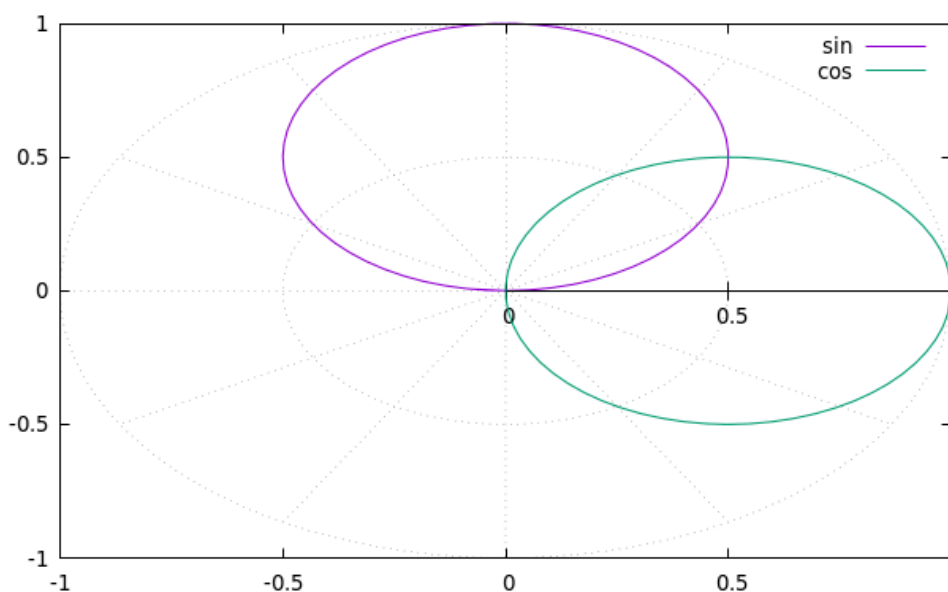
Function **plot()** was introduced in the previous section. Its arguments are sequences of the form: *list x [list y or function] [name]*, expressions in square brackets can be omitted.

```
## t3 = D:range(0, 2*PI, 0.05)
## Gp:plot(t3, sin, 'sin', t3, cos, 'cos')
```



The graph in polar coordinates is plotted by the function **polarplot()**, its signature is similar **plot()**.

```
## Gp:polarplot(t3, sin, 'sin', t3, cos, 'cos')
```

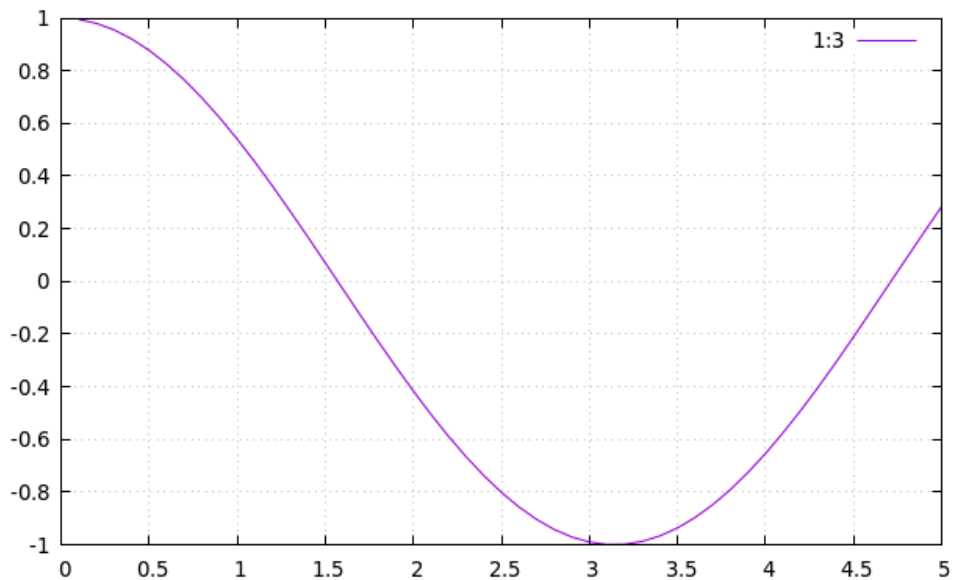


The function **tplot(array, [index x, index y1, index y2...])** is used to visualize tabular data. If column indices are not explicitly specified, all data is displayed, with the first column being considered the independent variable. A similar function **tpolar()** plots tabular data in polar coordinates.

```

## arr = {}
## for i = 1, 50 do \\
..   local x = 0.1*i
..   arr[i] = {x, sin(x), cos(x)}
.. end
..
## Gp:tplot(arr, 1, 3)

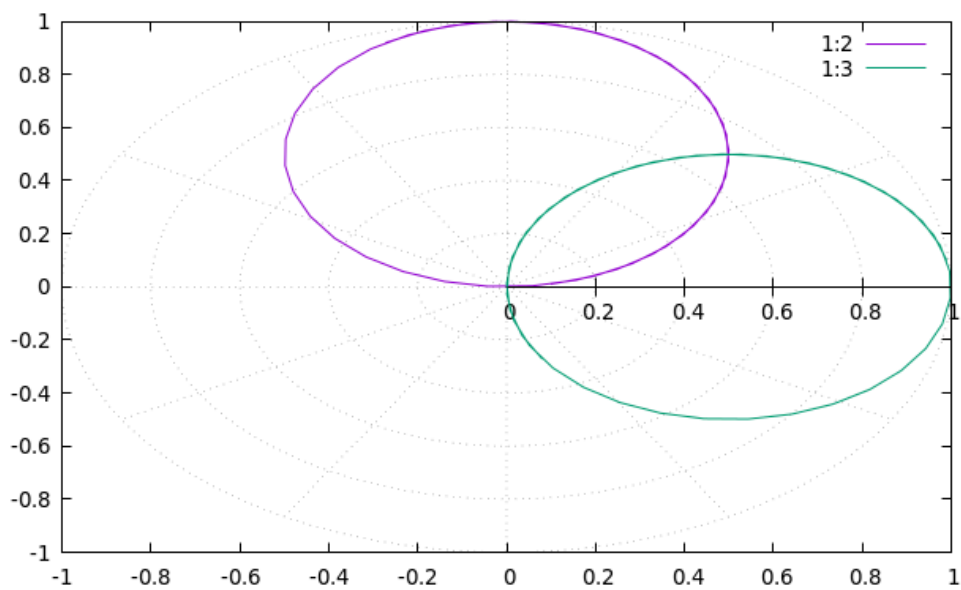
```



```

## Gp:tpolar(arr)

```

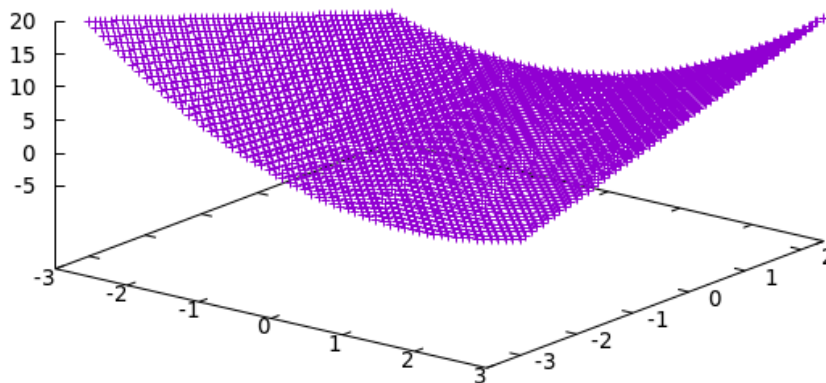


The function **surf plot**(list x list y function of (x, y)) is used to construct the surface. Surface points can be specified in a table, in which case you can use **tsurf**(table) for visualization.

```

## function fun(x,y) return x*x + y*x end
## t4 = D:range(-3, 3, 0.1)
## Gp:surfplot(t4, t4, fun, '3D')

```



18.3 State management

The functions described above use the default settings. If you need more control over the graph, you can customize the object generated with **Gp()**. Main parameters:

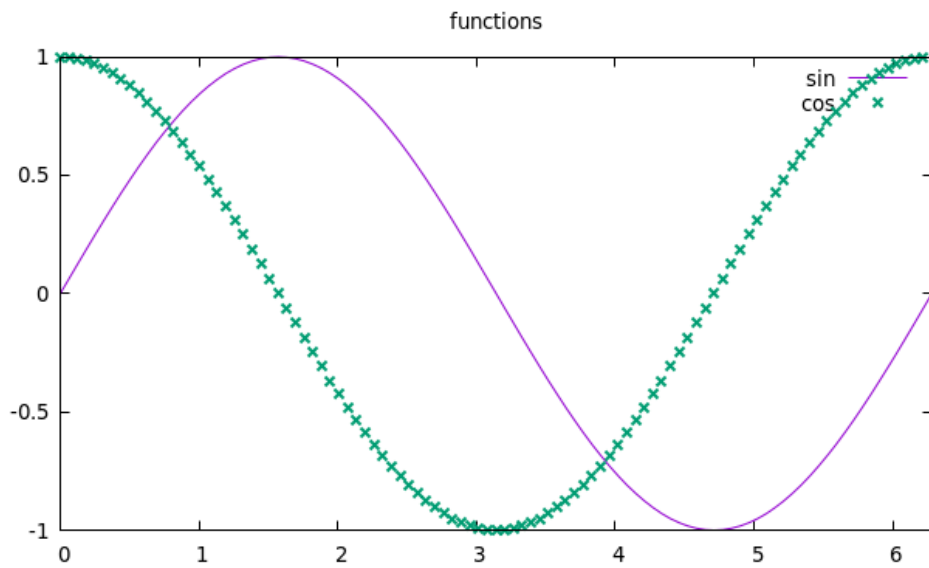
- *title* - title;
- *xlabel*, *ylabel* - names of axes;
- *legend* - show/hide legend;
- *terminal*, *output* - setting up data output;
- *raw* - execute a raw Gnuplot command.

The following can be specified for the line:

- *title* - Name;
- *with* - display method;
- *linetype*, *linestyle*, *linewidth* - line parameters.

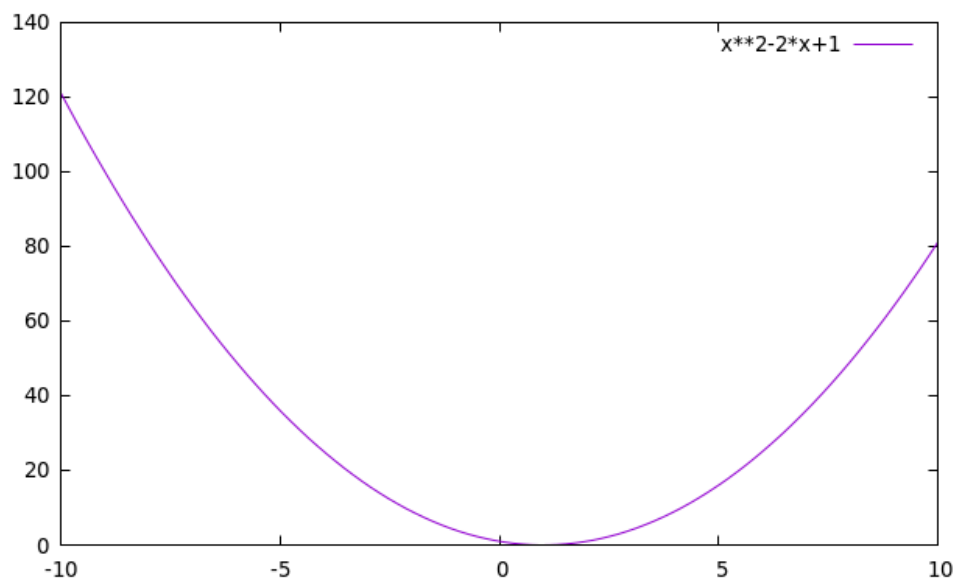
The function **add(data)** allows you to add data, its argument is a table with line type settings. The result is displayed by the function **show()**. You can copy an object with all its settings using the function **copy()**.

```
## fig = Gp()
## fig.xrange = {0, 2*PI}
## fig:add {sin, title='sin', with='lines'}
## fig:add {cos, title='cos', lw=2}
## fig.title = 'functions'
## fig:show()
```



If you are familiar with Gnuplot syntax, you can specify the command explicitly.

```
## fig = Gp()
## fig.raw = [[plot x**2-2*x+1; set xlabel "X"; set ylabel "Y"]]
## fig.show()
```



19 symbolic: Elements of Computer Algebra

19.1 Introduction

Module *symbolic* allows you to manipulate symbolic mathematical expressions. Although the capabilities of this module cannot be compared with such programs as, for example, Maxima, it allows you to perform some simplifications and calculate derivatives.

A symbolic variable or constant can be created using a constructor **Sym**(*variable*). The argument of the constructor can be one or more expressions separated by commas.

```
## a = Sym 'a'
## b = Sym(42)
## a * b
42*a
## p, q = Sym('x-y, x+y')
## p / q
(x-y)/(x+y)
## p + q
2*x
```

19.2 Basic operations

For symbolic variables, the operations of addition, subtraction, multiplication, division, and raising to a power are defined. An expression for certain values of variables can be calculated using the method **eval**(*meanings*), whose argument is a table of pairs *name* = *value*. Method **eval**() tries to simplify the given expression and can be called without arguments. If the result is a constant, you can convert it to a regular number using **val**()

```
## qe = q:eval {x=2, y=3}
## help(qe)
<symbolic>
5
## help(qe:val())
<number>
5
## p:eval {x=1}
1-y
```

The symbolic function can be defined with **def**(*name*, *arguments*, *body*), where the *arguments* is a list of names or symbols, and the *body* is an expression that is evaluated and returned when called. Arguments can be variables or names.

```
## foo = Sym:def('foo', {'x', 'y'}, \
.. Sym('x^2 + y^2')) -- returns
## foo
foo(x,y): x^2+y^2
## foo(p, q)
```

```
(x-y)^2+(x+y)^2
```

Standard and user-defined functions can be obtained from the table **fn**. You can check whether a symbol is a function using the method **isFn()**.

```
## Sym.fn.foo
foo(x,y): x^2+y^2
## foo:isFn()
true
```

The method **expand()** allows you to open the brackets. To calculate the derivative with respect to a given variable use **diff(variable)**, the argument can be a character object or a string.

```
## (p*q):expand()
x^2-y^2
## Sym('x^2*sin(x/2)'):diff('x')
0.5*x^2*cos(x/2)+2.0*x*sin(x/2)
```

If the expression is rational, you can extract the numerator and denominator using **ratNum()** and **ratDenom()** respectively. The method **struct()** allows you to see the internal representation of a symbolic object.

```
## r = p / q
## r
(x-y)/(x+y)
## r:ratNum()
x-y
## r:ratDenom()
x+y
```

20 qubit: quantum computing emulation

20.1 Introduction

Module *qubit* allows you to emulate qubits and gates (quantum valves), on the basis of which quantum computing algorithms can be implemented. However, it should be noted that these calculations are performed on a regular computer, so the computation time will grow exponentially with an increase in the number of qubits.

20.2 Qubits

A qubit can be defined as a sum of states in the basis “|0>” and “|1>”. The constructor **Qb(status_string)** is used for this purpose, which is multiplied by a complex (or real) coefficient. From a computational point of view, the state is described by a vector whose modulus is equal to one. Given the arbitrary nature of the initial coefficients, the result can be normalized by the function **normalize()**. When recording a state, the symbols “|>” can be omitted.

The probability of a particular state is returned by the method **prob()**. Function **meas(qubit)** removes uncertainty, and the “quantum” system goes into one of the possible states. It is possible to “measure” the state of a single qubit, while the uncertainty of the remaining elements remains.

```
## q = 0.4*Qb'|0>' + 0.6*Qb'|1>'
## a:normalize()
## q
(0.555)|0> + (0.832)|1>
## q:prob '|0>'
0.30769230769231 -- square of amplitude
## q:prob '|1>'
0.69230769230769
## q:meas() -- measurement
|1>
## q:prob '1'
1
## q:prob '0'
0
```

If you pass the number of qubits to the constructor instead of a state string, the system will be returned in a random initial state. To obtain a projection onto a given state, the product operator is used.

```
## q2 = Qb(1)
## q2
(0.715-0.369i)|0> + (0.193-0.562i)|1>
## k = 1 / sqrt(2)
## state = {[ '+'] = k*Qb'0'+k*Qb'1', ['-'] = k*Qb'0'-k*Qb'1'}
## state['+'] * q2
0.642-0.658i
```

```
## state['-'] * q2
0.369+0.136i
## q2 * q2
1.0
```

A system of several qubits can be specified using a constructor, or obtained by combining simpler systems using the function **combine**(Q1, Q2, ...), the concatenation operation is defined for two elements.

```
## Qb'00' + Qb'10' - Qb'01' -- normalization needed
(1)|00> + (-1)|01> + (1)|10>
## state['+'] .. state['-']
(0.500)|00> + (-0.500)|01> + (0.500)|10> + (-0.500)|11>
```

A copy of a qubit can be created using the method **copy()**. Function **matrix()** returns a representation in the form of a matrix (vector).

```
## q3 = q2:copy()
## q3:matrix()
0.715-0.369i
0.193-0.562i
```

20.3 Gates

Gates are transformations applied to a quantum system. In this sense, they can be thought of as functions (or more precisely, operators) that receive a system of qubits as input. From a computational point of view, gates form a matrix whose determinant is equal to one.

To initialize the gates, you need to call the function **gates(inputs)**, and specify the required number of inputs. The sequence of transformations is then written from left to right, using standard or user-defined elements. Pauli gates (**X()**, **Y()**, **Z()**), Hadamard **H()**, and also **S()** and **T()**, apply only to those inputs whose numbers are specified in the argument list, indexing the inputs from 0. If the list is empty, these gates apply to all inputs. Gate **SWAP**(n1, n2) swaps the specified outputs, **CNOT**(n1, n2) applies the NOT operation to the input n1 if n2 is in state |1>. The inverse transformation can be obtained using the method **inverse()**.

```
## g1 = Qb:gates(3):X(0,2):Y(1):Z()
## g1
|x2> X - Z
|x1> - Y Z
|x0> X - Z
## g1 = g1:CNOT(1,2):SWAP(0,1) -- the gate can be "appended"
## g1
|x2> X - Z o -
|x1> - Y Z x \
|x0> X - Z - /
## g1(Qb'001') -- apply to state |001>
|100>
## g2 = g1:inverse() -- inverse transformation
```

```
## g2
|x2> - o Z - X
|x1> \ x Z Y -
|x0> / - Z - X
## g2(Qb'100')
|001>
```

The user can create his own gates in two ways. If the identity transformation matrix is known, the method **fromMatrix(matrix)** can be used. If the truth table is known, it can be passed to the function **fromTable(table)**; each input state must correspond to a unique output state, and vice versa. The method **isUnitary()** allows you to check whether a gate is unitary.

```
## t = {      \\
.. {'00', '01'},
.. {'01', '00'},
.. {'10', '11'},
.. {'11', '10'}}
..
## g3 = Qb:gates(2):fromTable(t)
## g3
|x1> U
|x0>U
## g3:isUnitary()
true
## g3(Qb'01')
|00>
## m = Mat {      \\
.. {1, 0, 0, 0},
.. {0, 0, 1, 0},
.. {0, 1, 0, 0},
.. {0, 0, 0, 1}}
..
## m:det()
-1
## g4 = Qb:gates(2):fromMatrix(m)
## g4(Qb'01')
|10>
```

Gates also have multiplication and exponentiation operations. The first allows you to sequentially connect two transformations, the second - to repeat the gate a specified number of times. The matrix representation can be obtained using the method **matrix()**.

```
## g5 = g3 * g4^3
## g5
|x1> U U U U
|x0> U U U U
## g5:matrix()
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0
```

20.4 An example

As an example, consider Grover's algorithm, which solves the problem of finding an argument for which the function returns 1 (for all other valid values 0 is returned).

As an oracle, we take a three-qubit function that outputs one for the state '101'. This condition corresponds to a matrix in which all diagonal elements are 1 except for row 6 (binary 101 is 5, but indexing in Lua starts with 1, so 6). Grover's diffusion gate can also be defined in matrix form, here all diagonal elements are -1 except for the first. Let's combine the oracle with the diffusion gate.

```
## -- matrices
## mat_fn = Mat:eye(8)
## mat_fn[6][6] = -1
## mat_df = -Mat:eye(8)
## mat_df[1][1] = 1
## -- oracle + diffusion gate
## oracle_diffuse = Qb:gates(3) \\
.. :fromMatrix(mat_fn)
.. :H()
.. :fromMatrix(mat_df)
.. :H()
..
```

An equiprobable superposition of the input states can be obtained using the Hadamard gate. The call to the gate written above must be repeated at least twice ($\sqrt{2^3} \frac{\pi}{4} \approx 2$). After that, you can feed the system the initial state '000' and perform the measurement. It should be remembered that the algorithm is probabilistic and can sometimes return an incorrect result.

```
## grov = Qb:gates(3):H() * oracle_diffuse^2
## grov(Qb'000'):meas()
|101>
```

21 lens: paraxial optics

21.1 Introduction

Paraxial rays are those that travel at a small angle to the optical axis (usually up to 5 degrees). For such rays $\sin(\alpha) \approx \tan(\alpha) \approx \alpha$, which allows us to linearize the system and describe the beam transformation using matrices.

The module *lens* is based on the methods described in the book “Introduction to Matrix Optics”, A. Gerrard, J. M. Birch. Each transformation is described by the identity matrix $\begin{Bmatrix} A & B \\ C & D \end{Bmatrix}$, the transformation of the system as a whole is determined by the product of these matrices. The state of the beam is determined by its displacement relative to the optical axis y and optical angle $V = \alpha \cdot n$, equal to the product of the geometric angle and the refractive index of the medium.

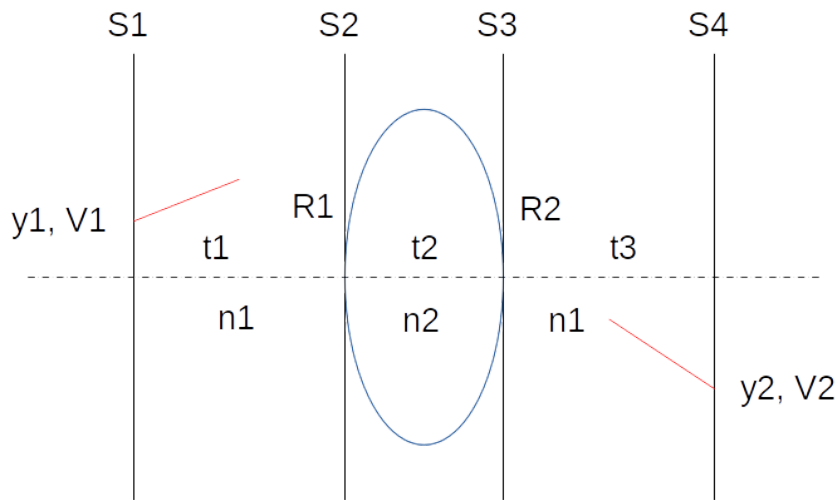
21.2 Optical systems

21.2.1 Description

To describe an optical system in terms of matrix optics, it is necessary to specify reference planes and determine the transformations between them. Elementary transformations include displacement **T**(distance, n), refraction **R**(n_{input} , radius, n_{output}) and reflection **M**(radius, n) which take into account the refractive index of the medium n . Matrices for the afocal system are also defined with **afocal**(increase) and a thin lens with **thin**(focal_length). An arbitrary matrix can be created using the constructor **Lens**(A, B, C, D). The sequence of transformations can be specified by sequentially calling the corresponding methods, or by concatenating individual elements.

As an example, consider the system shown in the following figure. The space between the reference planes S1 and S2 is filled with a homogeneous substance with a refractive index of n_1 , similarly for the space between S3 and S4. Let the optical system between S2 and S3 be a lens, then the transformations between these reference planes will include the refraction of a beam at the first surface, propagation in a medium with a refractive index of n_2 , and subsequent refraction at the second surface. Let $n_1 = 1$, $n_2 = 1.56$, $t_2 = 10$ mm, $R_1 = 200$ mm, $R_2 = -150$ mm (the surface is concave from the point of view of the “beam”). Then for the lens we can write

```
## L1 = Lens: R(1, 200, 1.56) : T(10, 1.56) : R(1.56, -150, 1)
## L1
A: 0.982    B: 6.410
C: -0.006   D: 0.976
```



Since the same refractive indices are found in adjacent elements, the following rule was implemented to simplify the notation: if a refractive index is specified in the constructor of an element, it can be “inherited” by the next element when called through a colon. Then the system under consideration can be represented as

```
## L2 = Lens:R(1, 200, 1.56) \\
..   :T(10)
..   :R(-150, 1)
..
## L2 == L1
true
```

Another simplification is the possibility of omitting n if it is equal to 1, i.e. the system is in the air. However, in the case of the element **R()** this feature should be used with caution.

Let $t1 = t2 = 20$ mm, $y1 = 2$ mm, $V1 = 0.05$. The transformation of the ray between planes S1 and S4 can be found as follows.

```
## sys = Lens:T(20) .. L1 .. Lens:T(20)
## y2, V2 = sys(2, 0.05)
## print(y2, V2)
3.85475555555556    0.0294044444444444
```

21.2.2 Analysis

The conversion of matrix elements to zero determines the following states:

- $D = 0$ - the input plane (S1) corresponds to the front focal plane;
- $B = 0$ - the input (S1) and output (S4) planes are conjugate (object and image planes) with magnification A ;
- $C = 0$ - afocal system;
- $A = 0$ - the output plane (S4) corresponds to the rear focal plane.

Function **solve**(function, index , x0) performs a parameter x search, at which the given matrix element becomes equal to zero. The argument here is a function that returns a description of the optical system taking into account the parameter x .

```
## -- determine the distance to the image plane t3 (B = 0)
## -- if the distance to the object t1 = 250 mm
## fn = function (d) return Lens:T(250) .. L1 .. Lens:T(d) end
## Lens:solve(fn, 'B', 100) -- initial approximation 100 mm
393.31465172138
## -- distance to the rear focal plane (A = 0)
## Lens:solve(fn, 'A', 100)
151.87162948081
```

The cardinal points of an optical system can be found using the function **cardinal()**, which returns a table with the positions of focal, principal and nodal points.

```
## t = L1:cardinal()
## t.F2 -- back focus
151.87162948081
## t
From the input plane
F1 at -150.94638891826
H1 and 3.700962250185
N1 at 3.7009622501851
From the output plane
F2 at 151.87162948081
H2 at -2.7757216876388
N2 at -2.7757216876388
```

21.2.3 Other functions

The function **copy()** allows you to create a copy of the system. The transformations of an optical system can be inverted using the method **inv()**. The values of individual elements can be obtained using indices (A, B, C, D), and the matrix as a whole - through the method **matrix()**.

```
## L1.C
-0.0064663247863248
## L1:matrix():det()
1.0
## isys = sys:inv()
## print(isys(y2, v2))
2.0    0.05
```

21.3 Laser radiation

Single-mode laser radiation is usually described by a Gaussian function. In the module *lens* several functions for working with Gaussian (laser) radiation are defined. Since the result is dimensional units (lengths, angles), all arguments must be of the same dimension. For convenience of transformations, you can use the module *units*.

Function **gParam**(constriction, wavelength) returns the far-field divergence angle and the near-field length, **gSize**(waist, wavelength, distance) defines the radius of curvature wave front R and beam size w at a given distance from the constriction. Method **beam**(R , w , wavelength) calculates R and w at the output of the system. If the optical system describes a resonator, then the method **emit**(wavelength) will return R for output radiation, and in the case of a stable resonator also the beam radius w , constrictions w_0 , and its displacement to the left from the reference plane.

```
## lam = 1024 * U('nm') -- wavelength
## -- constriction radius 1 mm
## print( Lens:gParam(1, lam 'mm') ) -- divergence, near zone
0.0003259493234522      3067.9615757713
## d = 100 * U('m') -- distance to obstacle
## -- radius of curvature, radius of the ray
## print( Lens:gSize(1, lam 'mm', d 'mm') )
100094.1238823      32.610268545191
## -- system output parameters, curvature radius 1000 mm
## print( L1:beam(1000, 1, lam 'mm') )
-180.0620907336      0.98846374677498
## -- resonator
## air_rod = Lens: T(50) : T(30, 1.56)
## mirror = Lens:M(-300)
## cavity = mirror .. air_rod .. mirror .. \
.. air_rod -- generally in reverse order
## print( cavity:emit(lam 'mm') )
300.0      0.29701991298779      0.22053244527224      134.61538461538
```

22 geodesy: coordinate transformations

22.1 Introduction

The *geodesy* module contains functions for solving a number of problems arising in geodesy, such as coordinate transformations or direct and inverse problems. In this case, the Earth is modeled as an ellipsoid with an equatorial radius a (major semi-axis) and polar radius b . Compression f is defined as $f = (a - b)/a$. Latitude, longitude and altitude are designated by B, L, H respectively, angular coordinates are expressed in degrees, distances in meters.

An ellipsoid for calculations can be obtained using the constructor **Geo(parameters)**. By default it uses WGS84 parameters, however, you can define your own values a and f . The coordinates of a point on the surface of an ellipsoid can be transformed into a rectangular geocentric system using the method **toXYZ(blh)**. The inverse transformation is performed by the function **toBLH(xyz)**.

```
## wgs = Geo()
## t1 = {B=22.2, L=33.3, H=10} -- angles in degrees
## xyz = wgs:toXYZ(t1)
## xyz
{
  Y = 3243716.0912491,
  Z = 2394935.8675327,
  X = 4938085.8106046,}
## wgs:toBLH(xyz)
{
  B = 22.2,
  H = 10.0,
  L = 33.3,}
```

22.2 Direct and inverse problems

Let a point in space (latitude and longitude) be given. A line of a certain length must be drawn relative to it in a direction specified by the azimuth. The direct problem is to find the position and azimuth of the end of the line. The inverse problem, accordingly, tries to find the length and azimuths for a segment whose ends are given.

To solve these problems, functions have been implemented **solveDir(start, azimuth, length)**, which returns the end point and azimuth, and **solveInv(beginning, end)**, returning distance and azimuths.

```
## -- angle 30 degrees, length 10 km
## t2, a2 = wgs:solveDir(t1, 30, 1e4)
## a2
30.018361900328
## t2
{
  L = 33.348504398613,
```

```

    B = 22.278201086692,}
## print(wgs:solveInv(t1, t2)) -- length, azimuth 1, azimuth 2
9999.4126387104    29.994174586851    30.012532175206

```

22.3 Coordinate transformations

22.3.1 Different ellipsoids

To create a new ellipsoid, you need to specify its parameters in the constructor **Geo()**. To transform coordinates between different ellipsoids, it is necessary to determine the linear and angular displacement between the bases, as well as the scale factor using the function **into(ellipsoid 1, ellipsoid 2, linear offset, angular offset, scale)**. After this, methods for mutual transformation of geocentric and astronomical coordinates will be added to each ellipsoid.

```

## -- Russian system "Parameters of the Earth"
## pz = Geo {a=6378136, f=1/298.25784}
## pz:into(wgs, \
.. {-1.1, -0.3, -0.9}, \
.. {0, 0, math.rad(-0.2/3600)}, \
.. -0.12E-6)
## xyz_wgs2pz = wgs.xyzInto[pz]
## xyz_wgs2pz(xyz)
{
  Z = 2394937.054925,
  X = 4938090.6483711,
  Y = 3243711.9923913,}
## blh_pz2wgs = pz.blhInto[wgs]
## blh_pz2wgs(t1)
{
  B = 22.199996473721,
  L = 33.300003425074,
  H = 7.508917491904,}

```

22.3.2 Single ellipsoid

The universal transverse Mercator (UTM) projection is widely used to represent the Earth's development on a plane. Function **bl2utm(blh)** for given astronomical coordinates determines the coordinates of the northward shift N and eastward shift E, the number of the corresponding zone, the hemisphere (N/S), and the scale factor. The inverse transformation is performed by the function **utm2bl(nose)**, which, in addition to the displacement value (NE), must know the zone number and the hemisphere index.

```

## ne, s = wgs:bl2utm(t1)
## ne
{
  hs = N,
  zone = 36,
  E = 530922.7463073,
  N = 2454995.0126965,}
## -- vice versa

```

```
## wgs:utm2bl(ne)
{
  L = 33.3,
  H = 0,
  B = 22.2, }
```

An alternative form of position representation is geohash. In fact, it specifies not a separate coordinate, but an area on the map, the size of which is determined by the length of the hash (the shorter it is, the larger the area). The function **hashEncode**(*dot*, *hash length*) allows you to encode the position. The second argument is optional, can take values from 1 to 12, it is 6 by default. The inverse transformation is performed by **hashDecode**(*hash*), the result is the coordinates of the central point of the zone and the dimensions in latitude and longitude.

```
## hash = Geo:hashEncode(t1, 5)
## hash
sezwm
## p, rng = Geo:hashDecode(hash)
## p
{
  B = 22.21435546875,
  L = 33.28857421875, }
## rng
{ 0.0439453125, 0.0439453125, }
```

22.4 Other functions

The function **dms2deg**(*degrees*, *minutes*, *seconds*) allows you to convert degrees, minutes and seconds to a decimal fraction, while the second and third arguments can be omitted. The inverse transformation is performed by the function **deg2dms**(*degrees*). The acceleration of gravity on the surface of an ellipsoid, taking into account the latitude, can be determined using the function **grav**(*latitude*).

```
## Geo:dms2deg(10, 20) -- degrees and minutes
10.333333333333
## print(Geo:deg2dms(10.123456)) -- in degrees, minutes, seconds
10    7    24.441599999997
## Geo:grav(50.5)
9.8111495438143
```

23 Applications

23.1 Setting up the environment

In Linux, you usually don't need to explicitly specify the path to the Lua program, since it is installed using standard tools. If the program is, for example, compiled from source codes, the path can be specified in the *bashrc* file as a line

```
alias lua='path to lua file'
```

In the case of Windows, you may need to go to the Environment Variables menu in System Properties, open the edition for the Path variable, click the Add button and enter the path to the lua executable file. After that, the program should be available for calling from the terminal.

23.2 Localization of the program

To add support for a new language, you need to generate a localization file, make a translation for the necessary functions and add the file name to the program settings. For example, to add a translation into French, you need to go to the folder with *so* and execute the command

```
lua sonata.lua --lang fr
```

After that in the folder *locale* file *fr.lua* will appear, containing a commented list of messages in English. If you uncomment any message and translate it, it will replace the default string when the file is loaded. The more messages are translated, the higher the localization of the program. To read the file when loading, you need to add it to a variable: `SONATA_LOCALIZATION = "fr.lua"`.

Command *lang* can also be used to update the list of messages in an existing file. That is, when this command is called, new messages (in English) will be added to the file and irrelevant ones will be deleted.

23.3 Adding a module

To add a new module, you need to place the corresponding file in the folder *matlib* and add a link into the table *use*.

The easiest way to add a module is to generate a template. For example, you need to add a module for signal processing. To do this, go to the *so* folder *sonata* and run the command

```
lua sonata.lua --new signal Sn "Signal processing"
```

Here is the first argument after *new* is the file name, the second is an alias (used in unit tests), the third argument contains a short description. The alias and description can be omitted. The result will be a file *signal.lua* in the folder *matlib*.

The methods available to the user will be located in the table with the previously specified name (signal). It is recommended to use the camel case naming style, and to start “hidden” methods with an underscore.

Each module contains a section that begins with ‘--[[TEST_IT’ and ends ‘]]’. These are unit tests, which consist of a sequence of text blocks separated by blank lines. If you want to test the result of a function, you need to save it in a variable *ans* and add the expected result in a comment that starts with ‘-->’ for exact equality or ‘--.n>’ for floating point comparison, where n is the order of precision.

To add a function description for **help()**, you need to place the function in the *about* table with a value in the form of a table of three elements: the first is the function signature, the second is the description, the third (optional) is a tag for grouping messages. For example:

```
about[signal.lowPass] = {":lowPass(freq_d) --> S", "Make low pass filter",  
"filters"}
```

The following rule is used for the signature: if the string starts with punctuation marks, the module alias will be added to it in the description, i.e. it is a class method, not an object. The postfix ‘_d’ here is used to indicate the data type (digit) and is optional.

To make the module load similar to other program components, you need to add it to the table *use* in the files *sonata.lua* in the form of *filename* = *Alias*. After that, you can load the module by mentioning its alias in the code or using **use()**.

23.4 Program input/output parameters

Input/output commands allow you to extend the capabilities of *so/\ata* by using third-party modules and programs, which, however, may not be portable between different operating systems and computers.

Command **tcp** runs the program in TCP server mode, using the *luasocket* library. The port number is a mandatory parameter. The server supports work with several clients, each of which operates in its own environment, however, their servicing is performed sequentially.

Command **w** starts the program in the "window" mode. In this case, input in the current terminal is impossible, and output is carried out from another terminal with which the operator works. Data exchange occurs due to the use of a temporary file, the contents of which are printed using the "tail -F" program.