AskLua: adding interactive help to other modules

Julio Manuel Fernández-Díaz

Profesor Titular, Department of Physics, University of Oviedo

February 2010

Table of contents

asklua: adding interactive help to other modules

Julio Manuel Fernández-Díaz

Profesor Titular, Department of Physics, University of Oviedo (Spain), February 2010

Abstract:

We present `AskLua`, a system for managing help: for on line use in the interactive interpreter, and for generating documentation in `html` and printed formats.

Module `ask`, provided by `AskLua`, is little intrusive and, although it occupies some memory, it can be deleted by the user at any time if he/she does not want to continue with the help on line.

The system is fairly integrated, in such a way that it is possible to easily add help for an existing module, even of binary type.

1 Introduction

When we learn a new subject it is suitable to have information, as complete as possible, about the topic. Often this is achieved through an expert (teacher). Nevertheless, we have not always access to one. For this it is necessary to be self-taught. In these occasions a good helping system, as interactive as possible, is very interesting.

Both Python and Matlab, for example, have interactive help on line. This allows the beginners a comfortable access to the system during the first phases of their learning.

Obviously tools like the mentioned languages will have more use with an easy access to the documentation. And this is very interesting for designers of Lua modules.

Lua has a quite minimalist approach, almost opposite to two mentioned. Therefore, it is practically a chimera the inclusion of a help system in the standard.

Lua does not impose policies, but it is very extensible. This allows us to easily include a system (module) of help, which is loaded if we wish.

Literate programming, very aimed to the programmer, does not convince me as a mechanism to solve our problem, since it is necessary to learn another *pseudo-language* and it really serves for another thing.

The system of documentation that accompanies `stdlua` with `@param`, `@returns`, etc., is not of my taste, since it consists of comments in the code. This is nice for checking and studying Lua code, but to obtain a system of documentation on line it is relatively complex. In addition, the documentation is also aimed to the programmer and not so much to the user of some module (for example, an engineer who wants to use Numlua).

A system very similar to `stdlua`, though more advanced is Luadoc, which allows the creation of documentation in `html` format, also starting from comments in the modules. The final format is friendly enough for managing the documentation in a browser, but it does not serve for on line help, which is what I initially wished.

In any case, both `stdlua` and Luadoc can be easily adapted to our purposes (below explained): it is enough to integrate the comments (by hand, before their use) in the corresponding help tables.

When I was ending this documentation I discovered in LuaRocks a module named LuaHelp (being December 13, 2009, the date of the first version). Initially I thought that my work had been useless since others have done the module that I was wishing. (Anyhow, new things are always learnt on programming.) I have to say that the announce of it in lua list goes unnoticed to me.

Once analyzed the topic more deeply, I have realized the differences (although LuaHelp might be adapted in many ways, obviously): the documentation that I propose is included in the own module, whereas the other one is included in auxiliary files placed in a concrete path. In addition, the method here presented allows to consult separately specific parts of the help: how a function in a module is used, examples of use, the list of functions of a module, etc. They are two different approaches with its advantages and disadvantages (that the user can verify).

The unique trouble is that we was compelled to change the name of my module. Now its name is `ask` (and I had preferred `help`). That way they can use both modules simultaneously. Anyhow, if the user wishes to rename `ask` and to call it `help` only renaming `ask.lua` to `help.lua` is sufficient. In this case it is interesting to change the notices that a module gives when loaded (see section 4.3 How we know that a module has help).

Another specific possibility is to load module `ask`, by referencing it with a variable `help` in the following way:

```
local help = require "ask"
```

Et voilà. We can choose both `ask` and `help` for using the present module (from this moment we cannot use LuaHelp, obviously).

2 The problem

It has two aspects:

1. On the one hand, for the *programmer* of a module who wants to add help. It will have to be easy to implement, with the minimal modifications to his/her module.

2. On the other hand, for the module *users*. The browsing of the documentation will have to be as complete as possible but of easy managing.

As programmers, we are interested in the ease of adding documentation, as complete as possible. Also, if possible several formats are welcome:

- on line in interactive Lua,
- on line in a browser, which requires `html` format (although some others exist, `html` is a standard),
- printed format.

In addition, the programmer is very interested in introducing the documentation only once, and the existence of an automatic system to produce several formats is nearly a must. This avoids many mistakes and the modifications, when necessary, have to be done only in one place.

From the user point of view, he/she will want that documentation is:

- *easy* to access, giving only the required information (not more, nor less);
- the *most complete* possible, covering many aspects, even secondary in occasions;
- *relational*, that is, it can allow us to go from one point to another in the documentation (if a relationship exists).

After many thoughts, many tests, and designing several versions of the help module, finally the system explained below arised.

3 The solution

The information of help in Lua can go only (more or less) in tables, with strings, or in files (as in LuaHelp). It has seemed to me that file managing would be cumbersome enough since files must be accessible and possibly where the module resides. It would be necessary to search in the *paths* of Lua and there might be more than one file with the name for that we are looking for (and not to correspond the help to what it is wished). Therefore I have preferred the other option.

Our module `ask` must be able to access to these tables (if they exist, but throwing a warning if no available information about a certain module exists), then giving the proper required information.

I will describe module `ask`.

3.1 Where information is placed

The first one is *where* is held the help information of a module. In Lua it is obvious that a table is the right place, with possible sub-tables, and finally text (strings). Besides, help for a module (v.g., `mininum`, the example attached to `ask`) must be associated to it in some way: Lua have to know the help like it knows about module functions (otherwise it cannot "help" us).

The help must be sub-divided. The final idea consists of including in the module (v.g., inside `module.lua`) a table `_H` with fields and sub-tables. Here is the structure:

```
_H = {
  _CHARSET = "iso-8985-15",   -- or typically "utf-8"
  _Name    = [[Name of the module]],

  _basic   = [[Basic help]],
  _usage   = [[Usage help]],
  _more    = [[Complementary information]],
  _seealso = [[Links to more information (internal or external)]],
  _example = [[An example of use]],
  _notes   = [[Notes (Copyright, etc.)]]
  _version = [[Version, name of the author and other subjects]],
}
```

(Note: `_Name` has a capital `"N"`.)

`_CHARSET` identifies what encoding has been used in the texts of help. For help in `html` format `ask` directly uses the value in this field.

For on line help, `ask` detects the operating system character encoding to convert from the set of characters indicated in `_CHARSET` to that one of the system (this way help is correctly visualized on screen). At the moment this system only is implemented with UFT-8 and iso-8859: If `ask` does not detect UTF-8 then it thinks `iso-8859` is the enconding in the operating system.

This table is included in any point of the file `module.lua` (although its logical place is near the beginning). None of the fields is mandatory: if the programmer does not gives an example the field `_example` is not specified.

As we see, in order that the minimal interference exists with other variables and functions of the module, I have put `"_"` in front of the table name. In this way, we will avoid collisions of names: it is rare to use names of this type (except for local variables). In any case, the user of `ask` can change in the source code `_H` into another name with a text editor (since the source code is

provided) and to perform accordingly.

Every function of the module (v.g., `about`, `base` and `doc` in the module `ask`) can have its associate help. For example, for the first function, it is:

```
_H.about = {
  _basic   = [[Basic help]],
  _usage   = [[Usage help]],
  _more    = [[Complementary information]],
  _seealso = [[Links to more information (internal or external)]],
  _example = [[An example of use]],
}
```

In some cases we have tables of functions. For example, if `tabfun` is a table with functions, namely `f1` y `f2`, we could put:

```
_H.tabfun = { ... }

_H.tabfun.f1 = {
  _basic   = [[ ... ]],
  _usage   = [[ ... ]],
  _example = [[ ... ]],
}

_H.tabfun.f2 = { ... }
```

Obviously if no information exists about some function, then we do not have to specify it (as `_more` and `_seealso` in previous `f1`).

Initially the format of the string going between `[[ ]]` was pure text. With it, for the help on line it is enough. Nevertheless, it was my desire that the information can also be printed. Then, a more advanced format is better. It leaded me to the format markdown, which in addition has an implementation in pure Lua, markdown.lua, done by Niklas Frykholm. (Note: the version appearing in luaforge.net is obsolete, but the one in Luarocks is correct.) Actually, this document has been prepared in markdown format.

This format is not much intrusive. This is important because I did not want to filter help on line, but I wanted it untouched in the strings of help.

3.2 How information is accessed

We have already the information in *its* place (Lua tables). When the module is loaded also these tables are. Then, from this moment they are accessible (evidently they occupy space; in 5 How to use the help interactively we will see how to free it if wished).

Now we must talk about *how* to access the information. For that, `ask` has three functions: `about`, `base` and `doc`.

| about |

This was also the first one in being designed: it provides the information on line. It is necessary to tell it *about* we want help and *which* of the diverse information piceces we wish (for example `basic`, for the basic information). Function `about` would be invoked, for example:

```
ask.about"/mininum.root^basic"
```

being `mininum` the name of the module, `root` the name of one of the functions inside the module, and `basic` the type of information we wish (I will explain below the inclusion of the initial `"/"`). A caret `"^"` separates what we wish from the type of information we want.

In another example we would have:

```
ask.about"/somemod.tabfun.f1^usage"
```

that is, we ask for `usage` help of the function `f1` inside the table `tabfun` of the module `somemod`.

Another important information for the user exists: the list of functions inside the module. The designer of the module has not to introduce it because `ask` obtains this directly. For displaying it in screen we have to use:

```
ask.about"/mininum^list"
```

Also it is possible to ask for all the information that exists about something, by means of:

```
ask.about"/mininum.root^all"
```

As we see, the user in this basic format of invocation of `about` has to type much. The following thing that I did was to admit abbreviations for the information type:

- `b` instead of `basic`, that provides basic information;
- `e` instead of `example`, that shows us an example;
- `l` instead of `list`, that shows a list of functions in the module (or in a table of functions);
- `m` instead of `more`, that shows additional information;
- `s` instead of `seealso`, that cites us links to more information (external or internal);
- `u` instead of `usage`, that shows us how to use the module or function;
- `v` instead of `version`, that shows us version information, name of the authors, etc.
- `n` instead of `notes`, that shows us notes as the Copyright, etc.
- `a` instead of `all`, that provides all the previous information.

In this way we have the equivalence:

```
ask.about"/mininum^list"  --><--  ask.about"/mininum^l"
```

The spaces typed in the argument of `ask.about` are removed before the search. Also `about` collapses multiple `"."` that could appear in the searching string. For example, we would have the equivalence:

```
ask.about "/ mininum . root ^ usage" --><-- ask.about"/mininum.root^u"
```

When information is not provided, by default it is supposed `basic`.

By using `nil` as argument of `ask.about` we obtain basic information about the help system. Then,

```
ask.about()  --><--  ask.about("/ask^basic")
```

are equivalent.

It is necessary to remember that `ask` provides the information the designer of the module has included in the proper help tables (except for `list` and `all`, obviously). That is, the information might be incorrect, for example, if data about the version is included in `_more`.

| base |

The reader can see we have still to type very much. Normally when one works with a module (let's say Numlua) we need help in an implicit way for it (and not for other module). For it, I have included the function `ask.base`. This function allows to implicitly assume an information searching prefix. For example, by putting:

```
ask.base"mininum"
-->  Changing help basis to "mininum"
```

from this moment the prefix `"/mininum"` is added to the search path (sometimes with an added point `"."`, when referring to a module or to a table of functions inside the module and we wish information about an specific function). Then, these are equivalent:

```
ask.about"/mininum^list"  --><--  ask.about"^l"
```

I have called this prefix `basis` (stored in a local variable).

Invoking `ask.base` using as parameter an empty string `""`, it shows us the current `basis`:

```
ask.base""
-->   Help basis is "mininum"
```

On the other hand, there has seemed suitable to me that:

```
ask.base(nil)
```

changes `basis` to `"ask"`.

The on line help system is already operative, but I liked no much the necessity of writing `ask.about` whenever I wanted help (altough, as read `ask.about"/mininum.root^usage"` is nearly equivalent to *ask about mininum.root usage*). The solution was to include the chunk of code:

```
__call = function (t, s, ...)
   if s == nil then
      about(nil)
   elseif type(s) == "string" then
      about(s)
   else
      return t[s](...)
   end
end

setmetatable(_G.ask, ask)
```

at the end of `ask.lua`. The last line allows us to use directly `ask` as a global variable after doing `require "ask"`. The previous chunk of code allows the function `about` is automatically invoked when `ask` receives a string as argument. Then

```
ask.about"/mininum.root^usage" --><-- ask"root^u"
```

are equivalent (with `basis = "mininum"`). The simplification is manifest, and it is near impossible to type less when asking help. Actually, slightly less we can type if the module `ask` is invoked in the way: `h = require "ask"`; from this moment we can put `h"root^u"`. Nevertheless, it seems to me that `ask` is not much writing, is more descriptive, and it is possible that `h` has another use inside our program.

| doc |

With the previous function I had already the module with the wished functionality. But it is a pity to have the information only on line and not to be able to print it on a decent format. Therefore, I have designed the function `doc`. The first idea was to adapt a bit the text output. But a bit more of expense when writing the help, using the markdown format, allows us an output in `html` format suitable enough. It was easy, since a module exists, markdown.lua (that obviously must be installed), which can be invoked from `ask`.

The `html` version of help is complementary to the on line one. It allows a more *global* vision of a given module, surfing the information by browsing among the different functions inside it. Hence, it is interesting enough.

In this point system `ask` seems Luadoc: both generate `html` format. Nevertheless, the starting points are very different.

It is also possible to obtain the help in printed format, printing from a browser. Probably it is better to use a program like html2ps by Jan Kärrman, that generates a Postscript (PS) format (and then PDF, also the latter with active hyperlinks).

The function `ask.doc` has been re-designed several times, and I am still not much happy with the result (though it works correctly, without being very elegant). It generates `html`. In this case, partial information of functions is not needed inside a module. Therefore, the generated `html` file contains all the information corresponding to the module (for it, `basis` must contain the name of the module and not of one of the functions inside). For example:

```
ask.doc""
```

would create `html` help for `basis` (v.g., `"mininum"` previously), with the name `"mininum.html"`, and:

```
ask.doc"ask"
```

would generate `html` help for `ask`.

The `html` file generated by `ask.doc` contains first generic information about the module (the content of `_H._basic`, `_H._usage`, etc.), then the help of all functions (in alphabetical order), to finish with `_H._version` and `_H._notes`.

I apologize since my programming is a little "dirty" in some parts of the module. My other occupations have not allowed me more time for improving it, and as one of my professors (León Garzón) said "the *best* is enemy of the *good*".

4 How modules are adapted to add them help

4.1 Why markdown format

The markdown format is simple enough in order that a module programmer learns it in an hour. It is not necessary for the help on line (even some people can think it is something annoying). I believe that it does not introduce much *noise* in the on line help, and nevertheless it allows a good presentation in `html` (and in PDF after other conversions).

It is possible to look up in the indicated address, but there also exists a two pages PDF document with a simple help of the system Markdown Syntax Cheat Sheet (the truth, it is necessary little more than this simple help to learn markdown).

Thinking a bit more about the help in `html` and printed formats we could introduce even references to images. It is evident that in the on line help (always text mode) it is impossible to visualize an image, but its reference can be given in markdown format (not much intrusive). It would be in the following way:

```
![Alternative text](/path/to/image.jpg "Optional title")
```

(Instead of JPG format we can also use PNG or GIF.)

The markdown format also allows to show links to other documents (internal or external ones) in a simple way. In the on line help this is not relevant but in `html` format is. For example, `_seealso` can be designed including links in the way:

```
internal:          [label](#reference)

external:          [label](http://what.we.want)
```

4.2 Preparing help data

The general idea is simple: doing it in an incremental way. A complete documentation can be a long and tedious task. Therefore, it would be convenient to begin with `_basic` help, then `_usage`, since they are both primary aspects. Since it is usual to put those as comments in the head of functions, what we have to do is to convert the comments already written in a piece of `_H`.

Later, other parts that still remain can be added. An example (commented if wished) is convenient for the user could try it (even copying and pasting code).

The logical place of introducing this information is before the corresponding function (as usual), and in case of modules near the beginning of the file.

If one wants to prepare the help for a compiled module we have to do the following.

If the binary module is ours, and we can recompile it, we can internally change the name of the table if we wish, renaming also the compiled file. Then one creates a Lua source file, with the included help, with the old name and extension `.lua`) that loads the compiled file.

A second possibility is to name in a different way the compiled module and the one that we are going to load with `require`. For example, we have a module compiled in binary format of name `mymodule.so` (or `mymodule.dll`). We will create a Lua file with name `mymodule1.lua` of the form:

```
-- file mymodule1.lua
require "mymodule"
mymodule._H = { ... }
```

To load `mymodule` we must use:

```
require "mymodule1"
```

and we have available the functions `mymodule.<function>`, and also the help `mymodule._H`, since this moment.

Another third possibility is to utilize the same name, using the fact that Lua first looks for source files in `LUA_PATH` and then binary ones in `LUA_CPATH`. Then, we would create a file `mymodule.lua` in some place in `LUA_PATH`, in the way:

```
-- file mymodule.lua
module(..., package.seeall)

local p = package.path
package.path = ""
package.loaded["mymodule"] = nil

require"mymodule"
package.path = p

mymodule._H = [[ ... help ... ]]
```

In this way we can still use:

```
require "mymodule"
```

as before, but now with included help.

Anyhow, this third system might fail if the binary module `mymodule.so` (or `mymodule.dll`) loads in turn modules in Lua source code because during its load the search in `package.path` is deactivated.

4.3 How we know that a module has help

At the end of the Lua file containing the module it is possible to include the following chunk of code to detect if Lua has been launched in interactive mode:

```
-- checks if Lua calling was interactive;
-- it does not work in all cases, but it does in the normal ones
local interactive = true
if _G.arg then
   for _, v in pairs(_G.arg) do
     interactive = false
     if v == "-i" then
       interactive = true
       break
     end
   end
end
```

This system of detection does not always work well (and it allows to generate the `html` documentation with a simple line in the operating system shell; we will see it in the following section).

It would be interesting that in interactive mode the Lua interpreter creates a variable indicating it. Or at least it could use the existent variable `_PROMPT` for it: at present, it is created only if the user defines it at interpreter launch time, otherwise it is `nil`. It would be enough to define `_PROMPT` in interactive mode to be `">"` if the user did not declare another value, and leave it `nil` in non-interactive mode.

After detecting interactivity we have the following chunk:

```
if interactive then
   -- reusing interactive
   interactive = pcall(require, "ask")                          --**
   if interactive then
      io.stderr:write('Module "mininum" loaded. ')
      io.stderr:write('To obtain help invoke ask"mininum".\n')  --**
      io.stderr:write('Documentation occupies memory. ')
      io.stderr:write('For freeing it let execute:\n')
```

```
      io.stderr:write('\n    mininum._H = nil\n\n')

      ask.base"mininum"
   else
      io.stderr:write('Module "mininum" loaded. It has help\n')
      io.stderr:write('but module "ask" is not accesible.\n')   --**
      io.stderr:write('Help removed.\n')
   end
end

if not interactive then
   -- deleting _H
   _H = nil
   _G.ask = nil
   collectgarbage()
end
```

This frees not necessary memory in non-interactive mode, and in addition it shows on screen some messages in interactive mode.

If we want to change the name of `ask.lua` to another one we will have also to change the lines indicated by `--**` above in a similar way.

Apart from this, in the future there is possible the inclusion of an external utility to adapt the format that accompanies Luadoc (and also the one that accompanies LuaHelp) for its use in the module `ask`. This allows to depart not from void for modules whose documentation was already prepared in another way.

5 How to use the help interactively

As before we have exposed, a module that takes help will have to show when loaded with `require` in order to the user knows that help is available.

The last loaded module that holds help will change `basis` to the name of the module. The user will have to bear in mind this fact. Anyhow, at any time the user will be able to change `basis` or he/she will be punctually able to use the system with the complete path (that begins with `"/"`).

If one wants to delete the help from memory it is easy. Let's suppose that we have the module `mininum` loaded with:

```
require "mininum"
```

then, it is enough to type:

```
mininum._H = nil   -- repeat this with each module
collectgarbage()
```

After that, if we want to completely delete the help system, we will write:

```
_G.ask = nil
collectgarbage()
```

From this moment on, the memory is freed of the helping system and the help information of module `mininum` (although, obviously its functions are still loaded).

6 How `html` (and PDF) documentation is generated

In this type of documents it is important to indicate the set of characters used in the help. At these moments I use `iso-8859-15` and therefore, in the help the field is included:

```
_H._CHARSET = "iso-8859-15"  -- "utf-8" sometimes
```

If one want to use `utf-8` (more current these days) it is necessary to change it in identical way. If not done so, some characters as "ü", "ñ", "á", will appear incorrectly. Normally, as we present modules help in English, people names and similar are usually the only words affected by this matter.

A very simple way to obtain the documentation in `html` format would be the following one in interactive mode:

```
$ lua
> require "mininum"
> ask.doc""
```

This would create `mininum.html`. Nevertheless, due to a small problem in the detection of the interactivity (see the following section), the same documentation can be also obtained by means of:

```
lua -e "require'mininum'; ask.doc''"
```

Once file `mininum.html` is created it is possible to visualize it with a browser to consult the help. The given style sheet `default.css` places at the left part of the page a fixed menu with the links of the whole document, including the list of functions inside the module.

The style of the `html` output is customizable, through a style sheet with name `default.css` (hardcoded inside `ask.lua`). I recomend to do it only with knowledge of CSS, but it does not hurt to play with it.

When a lot of functions exist in the module it is possible that some links disappear at the bottom of the screen. In this case it is necessary to diminish the font size (or to visualize the document without style, though ugly, it is still operative). Occasionally the font size should be diminished to see the examples (the lines do not wrap in the `html` elements of type `<pre>`; the examples are converted to `<pre>` elements).

Apart from viewing the `html` document it is also possible to print it, in the browser (not advisable) or with a specific program. We recommend the use of html2ps, which converts `html` in PS, with a style sheet a bit simplified (see the documentation provided in the link). In this case, the menu of navigation is not printed.

Later, if wished, we can convert the PS file into PDF by means of `ps2pdf` that accompanies Ghostscript. This method keeps the links, and we can navigate also in the PDF file.

7 Weaknesses and possible improvements

Firstly, due to the fact that we modify some tables (internal ones in the module that is been modified) it is possible that some collateral effects appear (which I have still not discovered).

On the other hand, a module can be loaded in a local variable, v.g., in the way:

```
local m = require"mininum"
```

to use "m." as prefix for the functions of mininum and to write less. From this moment it is possible to access the help with mininum and with m. This does not matter very much in interactive mode (as it is possible to access the functions with m.fun and with mininum.fun).

Nonetheless, ask.doc"m" creates a file mininum.html with the correct name (and not with the name m.html).

Functions in the module that are going to have help must not begin with "_" since ask supposes they are a part of its system. We can begin a function with "_" but it cannot have help.

Sometimes the interactivity does not work well. For example:

```
lua -e "require'mininum'" somefile.lua
```

shows an initial message and in addition it does not erase the help. Then, when it executes somefile.lua the help tables are in memory (what can be counter-productive).

Nevertheless it does not matter too much, because it is enough to include:

```
require"mininum"
```

at the beginning of somefile.lua, to invoke lua without the option -e, and then the module unloads ask and the help of mininum (as soon as it detects that the mode is non-interactive) before executing the rest of the program.

In interactive mode, in the future it is possible to include http links for more complete help (for example for a description of an algorithm or to give a link to a scientific article), or to verify if there are new versions of a given module. When some of these links appear a browser might be started when the user was asking for help of this type (there would be necessary to include a new type, v.g., link with k as abbreviation).

Tests could also be added (for every function in the module; partly they might be the executable examples). For example, two fields would be included: _test with code and _testsolution with text. On having executed the first one it should obtain the second one (the module ask would verify that they are equal). I do not know if this is interesting. There would be necessary to study it more deeply.

The system as presented here neither is closed nor is the definitive solution. It can have much improvement. The author wishes this version of the module can serve as an incentive for other programmers think about the topic and develop even better systems (or complete this).

8 Conclusions

We have designed an interesting system to provide help for other modules. This is versatile, and it allows on line access, html doc generation for its review in a browser, and printed format (PS and PDF).

This system is not perfect, but operative. The author believes that it can be the seed of a more complete help system for Lua. The advanced programmers can do the modifications they wish for their personal use. Actually, AskLua is released in the public domain. It is awaiting improvements.