Package 'luajr'

May 8, 2026

Type Package

Title 'LuaJIT' Scripting

Version 0.2.2

Description An interface to 'LuaJIT' <https://luajit.org>, a just-in-time compiler for the 'Lua' scripting language <https://www.lua.org>. Allows users to run 'Lua' code from 'R'.

URL https://github.com/nicholasdavies/luajr, https://nicholasdavies.github.io/luajr/

BugReports https://github.com/nicholasdavies/luajr/issues

License MIT + file LICENSE

Encoding UTF-8

SystemRequirements GNU make

Suggests Rcpp, crayon, knitr, rmarkdown, testthat (>= 3.0.0)

RoxygenNote 7.3.2

VignetteBuilder knitr

Config/testthat/edition 3

NeedsCompilation yes

Author Mike Pall [aut, cph] (Author of the embedded LuaJIT compiler), Lua.org, PUC-Rio [cph] (Copyright holders over portions of Lua source code included in LuaJIT), Nicholas Davies [cre, ctb, cph] (Author of the R package wrapper, ORCID: <https://orcid.org/0000-0002-1740-1412>), Scott Lembcke, Howling Moon Software [ctb, cph] (Authors of the embedded debugger.lua debugger)

Maintainer Nicholas Davies <nicholas.davies@lshtm.ac.uk>

Repository CRAN

Date/Publication 2026-02-16 19:50:02 UTC

Contents

`lua` *Run Lua code*

Description

Runs the specified Lua code.

Usage

```
lua(code, filename = NULL, L = NULL)
```

Arguments

`code`	Lua code block to run.
`filename`	If non-`NULL`, name of file to run.
`L`	Lua state in which to run the code. `NULL` (default) uses the default Lua state for **luajr**.

Value

Lua value(s) returned by the code block converted to R object(s). Only a subset of all Lua types can be converted to R objects at present. If multiple values are returned, these are packaged in a `list`.

Examples

```
twelve <- lua("return 3*4")
print(twelve)
```

`lua_func` *Make a Lua function callable from R*

Description

Takes any Lua expression (as a character string) that evaluates to a function and provides an R function that can be called to invoke the Lua function. Instead of a character string, you can also provide an external pointer to a Lua function (see examples).

Usage

```
lua_func(func, argcode = "s", L = NULL)
```

Arguments

`func`	A character string with a Lua expression evaluating to a function, or an external pointer to a Lua function.
`argcode`	How to wrap R arguments for the Lua function.
`L`	Lua state in which to run the code. `NULL` (default) uses the default Lua state for **luajr**.

Details

The R types that can be passed to Lua are: `NULL`, logical vector, integer vector, numeric vector, string vector, list, external pointer, and raw.

The parameter `argcode` is a string with one character for each argument of the Lua function, recycled as needed (e.g. so that a single character would apply to all arguments regardless of how many there are).

In the following, the corresponding character of `argcode` for a specific argument is referred to as its arg code.

For `NULL` or any argument with length 0, the result in Lua is **nil** regardless of the corresponding arg code.

For logical, integer, double, and character vectors, if the corresponding arg code is `'s'` (simplify), then if the R vector has length one, it is supplied as a Lua primitive (boolean, number, number, or string, respectively), and if length > 1, as an array, i.e. a table with integer indices starting at 1. If the code is `'a'`, the vector is always supplied as an array, even if it only has length 1. If the arg code is the digit `'1'` through `'9'`, this is the same as `'s'`, but the vector is required to have that specific length, otherwise an error message is emitted.

Still focusing on the same vector types, if the arg code is `'r'`, then the vector is passed *by reference* to Lua, adopting the type `luajr.logical_r`, `luajr.integer_r`, `luajr.numeric_r`, or `luajr.character_r` as appropriate. If the arg code is `'v'`, the vector is passed *by value* to Lua, adopting the type `luajr.logical`, `luajr.integer`, `luajr.numeric`, or `luajr.character` as appropriate.

For a raw vector, only the `'s'` type is accepted and the result in Lua is a string (potentially with embedded nulls).

For lists, if the arg code is `'s'` (simplify), the list is passed as a Lua table. Any entries of the list with non-blank names are named in the table, while unnamed entries have the associated integer key in the table. Note that Lua does not preserve the order of entries in tables. This means that an R list with names will often go "out of order" when passed into Lua with `'s'` and then returned back to R. This is avoided with arg code `'r'` or `'v'`.

If a list is passed in with the arg code `'r'` or `'v'`, the list is passed to Lua as type `luajr.list`, and all vector elements of the list are passed by reference or by value, respectively.

For external pointers, the arg code is ignored and the external pointer is passed to Lua as type **userdata**.

When the function is called and Lua values are returned from the function, the Lua return values are converted to R values as follows.

If nothing is returned, the function returns `invisible()` (i.e. `NULL`).

If multiple arguments are returned, a list with all arguments is returned.

Reference types (e.g. `luajr.logical_r`) and vector types (e.g. `luajr.logical`) are returned to R as such. A `luajr.list` is returned as an R list. Reference and list types respect R attributes set within Lua code.

A **table** is returned as a list. In the list, any table entries with a number key come first (with indices 1 to n, i.e. the original number key's value is discarded), followed by any table entries with a string key (named accordingly). This may well scramble the order of keys, so beware. Note in particular that Lua does not guarantee that it will traverse a table in ascending order of keys. Entries with non-number, non-string keys are discarded. It is probably best to avoid returning a **table** with anything other than string keys, or to use `luajr.list`.

A Lua string with embedded nulls is returned as an R raw type.

A function is returned as an external pointer, which itself can be converted into a function that can be called from R, by passing it through `lua_func()` as the `func` argument.

Value

An R function which can be called to invoke the Lua function.

Examples

```
# use with a character string
squared <- lua_func("function(x) return x^2 end")
print(squared(7))

# use with an external pointer to a Lua function
times2ptr <- lua("return function(x) return 2 * x end")
print(times2ptr)
times2 <- lua_func(times2ptr)
print(times2(14))
```

lua_mode *Debugger, profiler, and JIT options*

Description

Run Lua code with the debugger or profiler activated, and control whether the LuaJIT just-in-time compiler is on.

Usage

```
lua_mode(expr, debug, profile, jit)
```

Arguments

`expr`	An expression to run with the associated settings. If `expr` is present, the settings apply only while `expr` is being evaluated. If `expr` is missing, the settings apply until they are changed by another call to `lua_mode()`.
`debug`	Control the debugger: `"step"` / `"on"` / `TRUE` to step through each line; `"error"` to trigger the debugger on a Lua error; `"off"` / `FALSE` to switch the debugger off.
`profile`	Control the profiler: `"on"` / `TRUE` to use the profiler's default settings; a specially formatted string (see below) to control the profiler's precision and sampling interval; `"off"` / `FALSE` to switch the profiler off.
`jit`	Control LuaJIT's just-in-time compiler: `"on"` / `TRUE` to use the JIT, `"off"` / `FALSE` to use the LuaJIT interpreter only.

Value

When called with no arguments, returns the current settings. When called with `expr`, calls the value returned by `expr`. Otherwise, returns nothing.

Details

This function is experimental. Its interface and behaviour may change in subsequent versions of luajr.

`lua_mode()` works in one of three ways, depending on which parameters are provided.

When called with no arguments, `lua_mode()` returns the current `debug`, `profile`, and `jit` settings.

When called without an `expr` argument, but with at least one of `debug`, `profile`, or `jit`, the specified settings apply for any subsequent executions of Lua code until the next call to `lua_mode()`.

When called with an `expr` argument, the specified settings for `debug`, `profile`, and `jit` are applied temporarily just for the evaluation of `expr` in the calling frame.

Note that if you provide some but not all of the `debug`, `profile`, and `jit` arguments, the "missing" settings are retained at their current values, not reset to some default or "off" state. In other words, you can temporarily change one setting without affecting the others.

The debugger

The debug setting allows you to run Lua code in debug mode, using Scott Lembcke's debugger.lua.

Use debug = "step" (or TRUE or "on") to step through each line of the code; use debug = "error" to trigger the debugger on any Lua error; and turn off the debugger with debug = "off" (or FALSE).

To trigger the debugger from a specific place within your Lua code, you can also call luajr.dbg() from your Lua code. Within Lua, you can also use luajr.dbg(CONDITION) to trigger debugging only if CONDITION evaluates to false or nil. (In this way, luajr.dbg(CONDITION) is sort of like an assert(CONDITION) call that triggers the debugger when the assert fails.)

debugger.lua is more fully documented at its github repo page, but briefly, you enter commands of one character at the debugger.lua> prompt. Use n to step to the next line, q to quit, and h to show a help page with all the rest of the commands.

The profiler

The profile setting allows you to profile your Lua code run, generating information useful for optimising its execution speed.

Use profile = "on" (or TRUE) to turn on the profiler with default settings (namely, profile at the line level and sample at 10-millisecond intervals).

Instead of "on", you can pass a string containing any of these options:

- f: enable profiling to the function level.
- l: enable profiling to the line level.
- i<integer>: set the sampling interval, in milliseconds (default: 10ms).
- d<integer>: set the maximum stack depth (default: 200).
- z<real>: set the maximum profile size, in megabytes (default: 128 Mb).

For example, the default options correspond to the string "li10d200z128" or just "l".

If the internal buffer gets full (surpasses the limit in megabytes set by the z option), the profiler will stop recording profiles but Lua code will continue executing as normal. The truncation will be reported as a warning by lua_profile(). The default size, 128 Mb, is sufficient for about an hour of profiling at 10ms intervals, assuming the average call stack depth is 10. If you really need to profile something that takes more than an hour to run, it probably makes more sense to lengthen the sampling interval rather than increase the maximum profile size. Each Lua state has its own buffer, so if you are profiling code across multiple Lua states, this limit applies separately to each one of them.

You must use lua_profile() to recover the generated profiling data.

JIT options

The jit setting allows you to turn LuaJIT's just-in-time compiler off (with jit = "off" or FALSE). The default is for the JIT compiler to be "on" (alias TRUE).

Lua code will generally run more slowly with the JIT off, although there have been issues reported with LuaJIT running more slowly with the JIT on for processors using ARM64 architecture, which includes Apple Silicon CPUs.

See Also

lua_profile() for extracting the generated profiling data.

Examples

```
## Not run:
# Debugger in "one-shot" mode
lua_mode(debug = "on",
    sum <- lua("
        local s = 0
        for i = 1,10 do
            s = s + i
        end
        return s
    ")
)

# Profiler in "switch on / switch off" mode
lua_mode(profile = TRUE)
pointless_computation = lua_func(
"function()
    local s = startval
    for i = 1,10^8 do
        s = math.sin(s)
        s = math.exp(s^2)
        s = s + 1
    end
    return s
end")
lua("startval = 100")
pointless_computation()
lua_mode(profile = FALSE)
lua_profile()

# Turn off JIT and turn it on again
lua_mode(jit = "off")
lua_mode(jit = "on")

## End(Not run)
```

---

lua_module *Load Lua modules*

Description

lua_module() can be used in an R project or package to declare a Lua module in an external file. You can then use lua_import() to access the functions within the module, or provide access to those functions to your package users. The object returned by lua_module() can also be used to set and get other (non-function) values stored in the Lua module table.

Usage

```
lua_module(filename = NULL, package = NULL)

lua_import(module, name, argcode)
```

Arguments

filename	Name of file from which to load the module. If this is a character vector, the elements are concatenated together with file.path().
package	If non-NULL, the file will be sought within this package.
module	Module previously loaded with lua_module().
name	Name of the function to import (character string).
argcode	How to wrap R arguments for the Lua function; see documentation for lua_func().

Value

lua_module() returns an environment with class "luajr_module".

Typical usage

```
# To load a Lua module containing myfunc(x,y)
mymod <- lua_module("Lua/mymodule.lua", package = "mypackage")
func <- function(x, y) lua_import(mymod, "myfunc", "s")
```

Module files

Module files should have the file extension .lua and be placed somewhere in your project directory. If you are writing a package, the best practice is probably to place these in the subdirectory inst/Lua of your package.

The module file itself should follow standard practice for Lua modules. In other words, the module file should return a Lua table containing the module's functions. A relatively minimal example would be:

```
local mymodule = {}
mymodule.fave_name = "Nick"

function mymodule.greet(name)
    print("Hello, " .. name .. "!")
    if name == mymodule.fave_name then
        print("Incidentally, that's a great name. Nice one.")
    end
end

return mymodule
```

Loading the module

Before you import functions from your module, you need to create a module object using `lua_module()`. Supply the file name as the `filename` argument to `lua_module()`. If you are developing a package, also supply your package name as the `package` argument. If `package` is `NULL`, `lua_module()` will look for the file relative to the current working directory. If `package` is non-`NULL`, `lua_module()` will look for the file relative to the installed package directory (using `system.file()`). So, if you are developing a package and you have put your module file in `inst/Lua/mymodule.lua` as recommended above, supply `"Lua/mymodule.lua"` as the filename.

The module returned by `lua_module()` is not actually loaded until the first time that you import a function from the module. If you want the module to be loaded into a specific Lua state in your R project, then assign that state to the module's state right after declaring it:

```
mymod <- lua_module("path/to/file.lua", package = "mypackage")
mymod$L <- my_state
```

If you are creating a package and you want to load your module into a specific Lua state, you will need to create that state and assign it to `module$L` after the package is loaded, probably by using `.onLoad()`.

Importing functions

To import a function from a module, declare it like this:

```
myfunc <- function(x, y) lua_import(mymod, "funcname", "s")
```

where `mymod` is the previously-declared module object, `"funcname"` is the function name within the Lua module, and `"s"` is whatever arg code you want to use. Note that `lua_import()` must be used as the only statement in your function body and you should **not** enclose it in braces (`{}`). The arguments of `myfunc` will be passed to the imported function in the same order as they are declared in the function signature. You can give default values to the function arguments.

With the example above, the first time you call `myfunc()`, it will make sure the module is properly loaded and then call the Lua function. It will also overwrite the existing body of `myfunc()` with a direct call to the Lua function so that subsequent calls to `myfunc()` execute as quickly as possible.

In some cases, you may want to do some processing or checking of function arguments in R before calling the Lua function. You can do that with a "two-step" process like this:

```
greet0 <- function(name) lua_import(mymod, "greet", "s")
greet <- function(name) {
    if (!is.character(name)) {
        stop("greet expects a character string.")
    }
    greet0(name)
}
```

In a package, you can document and export a function that uses `lua_import()` just like any other function.

Setting and getting

Lua modules can contain more than just functions; they can also hold other values, as shown in the example module above (under "Module files"). In this example, the module also contains a string called `fave_name` which alters the behaviour of the `greet` function.

You can get a value from a module by using e.g. `module["fave_name"]` and set it using e.g. `module["fave_name"] <- "Janet"`. You must use single brackets `[]` and not double brackets `[[]]` or the dollar sign `$` for this, and you cannot change a function at the top level of the module. If your module contains a table `x` which contains a value `y`, you can get or set `y` by using multiple indices, e.g. `module["x", "y"]` or `module["x", "y"] <- 1`. Using empty brackets, e.g. `module[]`, will return all the contents of the module, but you cannot set the entire contents of the module with e.g. `module[] = foo`.

By default, when setting a module value using `module[i] <- value`, the value is passed to Lua "by simplify" (e.g. with arg code `"s"`). You can change this behaviour with the `as` argument. For example, `module[i, as = "a"] <- 2` will set element `i` of the module to a Lua table `{2}` instead of the plain value `2`.

Examples

```
module <- lua_module(c("Lua", "example.lua"), package = "luajr")
greet <- function(name) lua_import(module, "greet", "s")
greet("Janet")
greet("Nick")
```

---

lua_open *Create a new Lua state*

Description

Creates a new, empty Lua state and returns an external pointer wrapping that state.

Usage

```
lua_open()
```

Details

All Lua code is executed within a given Lua state. A Lua state is similar to the global environment in R, in that it is where all variables and functions are defined. **luajr** automatically maintains a "default" Lua state, so most users of **luajr** will not need to use `lua_open()`.

However, if for whatever reason you want to maintain multiple different Lua states at a time, each with their own independent global variables and functions, `lua_open()` can be used to create a new Lua state which can then be passed to `lua()`, `lua_func()` and `lua_shell()` via the `L` parameter. These functions will then operate within that Lua state instead of the default one. The default Lua state can be specified explicitly with `L = NULL`.

Note that there is currently no way (provided by **luajr**) of saving a Lua state to disk so that the state can be restarted later. Also, there is no `lua_close` in **luajr** because Lua states are closed automatically when they are garbage collected in R.

Value

External pointer wrapping the newly created Lua state.

Examples

```
L1 <- lua_open()
lua("a = 2")
lua("a = 4", L = L1)
lua("print(a)") # 2
lua("print(a)", L = L1) # 4
```

---

lua_parallel *Run Lua code in parallel*

Description

Runs a Lua function multiple times, with function runs divided among multiple threads.

Usage

```
lua_parallel(func, n, threads, pre = NA_character_)
```

Arguments

func	Lua expression evaluating to a function.
n	Number of function executions.
threads	Number of threads to create, or a list of existing Lua states (e.g. as created by lua_open()), all different, one for each thread.
pre	Lua code block to run once for each thread at creation.

Details

This function is experimental. Its interface and behaviour are likely to change in subsequent versions of luajr.

lua_parallel() works as follows. A number threads of new Lua states is created with the standard Lua libraries and the luajr module opened in each (i.e. as though the states were created using lua_open()). Then, a thread is launched for each state. Within each thread, the code in pre is run in the corresponding Lua state. Then, func(i) is called for each i in 1:n, with the calls spread across the states. Finally, the Lua states are closed and the results are returned in a list. The list elements are returned in the correct order, i.e. the ordering of the returned list does not depend on the actual execution order of each call to func.

Instead of an integer, threads can be a list of Lua states, e.g. NULL for the default Lua state or a state returned by lua_open(). This saves the time needed to open the new states, which takes a few milliseconds.

Value

List of n values returned from the Lua function func.

Safety and performance

Note that func has to be thread-safe. All pure Lua code and built-in Lua library functions are thread-safe, except for certain functions in the built-in **os** and **io** libraries (search for "thread safe" in the Lua 5.2 reference manual).

Additionally, use of luajr reference types is **not** thread-safe because these use R to allocate and manage memory, and R is not thread-safe. This means that you cannot safely use luajr.logical_r, luajr.integer_r, luajr.numeric_r, luajr.character_r, or other reference types within func. luajr.list and luajr.dataframe are fine, provided the list entries / dataframe columns are value types.

There is overhead associated with creating new Lua states and with gathering all the function results in an R list. It is advisable to check whether running your Lua code in parallel actually gives a substantial speed increase.

Examples

```
lua_parallel("function(i) return i end", n = 4, threads = 2)
```

---

lua_profile *Get profiling data*

Description

After running Lua code with the profiler active (using lua_mode()), use this function to get the profiling data that has been collected.

Usage

```
lua_profile(flush = TRUE)
```

Arguments

flush	If TRUE, clears the internal profile data buffer (default); if FALSE, doesn't. (Set to FALSE if you want to 'peek' at the profiling data collected so far, but you want to collect more data to add to this later.)

Details

This function is experimental. Its interface and behaviour may change in subsequent versions of luajr.

Value

A data.frame.

See Also

lua_mode() for generating the profiling data.

Examples

```
## Not run:
lua_mode(profile = TRUE)
pointless_computation = lua_func(
"function()
    local s = startval
    for i = 1,10^8 do
        s = math.sin(s)
        s = math.exp(s^2)
        s = s + 1
    end
    return s
end")
lua("startval = 100")
pointless_computation()
lua_mode(profile = FALSE)

prof = lua_profile()

## End(Not run)
```

---

lua_reset *Reset the default Lua state*

Description

Clears out all variables from the default Lua state, freeing up the associated memory.

Usage

```
lua_reset()
```

Details

This resets the default Lua state only. To reset a non-default Lua state L returned by lua_open(), just do L <- lua_open() again. The memory previously used will be cleaned up at the next garbage collection.

Value

None.

Examples

```
lua("a = 2")
lua_reset()
lua("print(a)") # nil
```

`lua_shell` *Run an interactive Lua shell*

Description

When in interactive mode, provides a basic read-eval-print loop with LuaJIT.

Usage

```
lua_shell(L = NULL)
```

Arguments

`L`	Lua state in which to run the code. `NULL` (default) uses the default Lua state for **luajr**.

Details

Enter an empty line to return to R.

As a convenience, lines starting with an equals sign have the `"="` replaced with `"return "`, so that e.g. entering `=x` will show the value of `x` as returned to R.

Value

None.

Index