Package 'luajr'

June 2, 2026

Type Package

Title 'LuaJIT' Scripting

Version 0.3.0

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

Depends R (>= 4.0.0)

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

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

RoxygenNote 7.3.2

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 2026-06-02 18:40:02 UTC

Contents

lua 2
lua_func 3
lua_mode 5
lua_module 7
lua_open 10
lua_profile 11
lua_reset 12
lua_shell 13

Index **14**

`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 Lua 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 = ".", 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

Any R type can be passed to a Lua function. The R types that have special support in **luajr** are: NULL, logical vector, integer vector, numeric vector, character vector, list, function, environment, and external pointer.

The parameter argcode is a string that specifies type handling for each argument of the Lua function. The last type is repeated when there are more arguments than specified types.

To receive a luajr.logical type in Lua, use the argcode logical or L. Similarly, luajr.integer is specified with integer or I; luajr.numeric is specified with numeric or N; and luajr.character is specified with character or C. luajr.list is specified with list, vector, or V.

To receive a luajr.rfunction type, use rfunction or F. To receive a luajr.environment type, use environment or E. To receive an R.sexp type, use sexp or S.

To pass any R type as "closest available" type from the above, use the argcode auto or ".".

Argcodes should be separated by commas. If you are using a "single-character" argcode like . or S, there does not need to be a comma following it.

You can also receive R values as Lua native types. Specifically, the argcode function corresponds to a Lua function; boolean is a Lua boolean; number is a Lua number; string is a Lua string and table is a Lua table. pointer or P passes an external pointer (EXTPTRSXP) as a Lua light userdata. The argcode native selects the closest available native type.

Additionally, the prefix $ specifies that the type should be passed as a Lua native type. So $auto is equivalent to native and $C is equivalent to string.

Note that native scalar argcodes (e.g. $L, $I, $N, $C, boolean, number, string) do not preserve R's NA values, because Lua has no NA concept: an R NA of any type is passed as Lua nil, which a

non-strict argcode silently treats as missing. Plain NaN values (e.g. from 0/0) do survive as Lua NaN because they are distinct from R's NA_real_. To preserve typed NAs on a round-trip, pass values through a luajr vector type (e.g. luajr.numeric, luajr.logical) instead of the native scalar form, and read individual elements in Lua via v(i) (which returns a length-1 vector of the same type rather than a bare scalar).

The prefix ! specifies strict type handling (i.e., error if the type cannot be converted). ! also yields an error on passing NULL if the argcode specifies a Lua native type (otherwise, NULL gets passed as Lua nil).

The prefix & can be used with logical, integer, numeric, character, or generic (list) vectors and specifies that the value should be passed **by reference**. Vectors passed by reference can have their elements mutated in Lua code, but they cannot be resized.

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.

Vector types (e.g. luajr.logical) are returned to R as the corresponding R vector. A luajr.list is returned as an R list. Vector 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", ".")
```

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$L` after the package is loaded, probably by using `.onLoad()`. An assignment to `module$L` only works when done before any module functions or variables are used; after the first use of any module value, changing `module$L` has no effect.

Importing functions

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

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

where `mymod` is the previously-declared module object, `"funcname"` is the function name within the Lua module, and `"."` 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", "$.")
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 as a Lua native type (e.g. with arg code `"$."`). You can change this behaviour with the `as` argument. For example, `module[i, as = "table"] <- 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", "$.")
greet("Janet")
greet("Nick")
```

---

lua_open *Create a new Lua state*

Description

Creates a new 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_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

.onLoad(), *9*

arg code, *9, 10*

file.path(), *8*

lua, 2
Lua state, *2, 3*, *9*, *12, 13*
lua(), *10*
lua_func, 3
lua_func(), *8*, *10*
lua_import *(*lua_module*)*, 7
lua_import(), *7*, *9*
lua_mode, 5
lua_mode(), *5*, *11*
lua_module, 7
lua_module(), *7–9*
lua_open, 10
lua_open(), *10*, *12*
lua_profile, 11
lua_profile(), *6, 7*
lua_reset, 12
lua_shell, 13
lua_shell(), *10*

system.file(), *9*