Tutorial: Using external C libraries with the LuaTeX FFI

Henri Menke

Abstract

The recent 1.0.3 release of LuaTeX introduced an FFI library (Foreign Function Interface) with the same interface as the one included by default in the LuaJIT interpreter. This allows for interfacing with any external library which adheres to the C calling convention for functions, which is pretty much everything. In this tutorial I will present how to interface with the GNU Scientific Library (GSL) to calculate integrals numerically. As a showcase I will plot a complete Fermi-Dirac integral using the `pgfplots` package. To understand this article, the reader should have good knowledge of the Lua and C programming languages and a basic understanding of the C memory model.

1 The FFI library

Lua is known for its rich C API which allows interfacing with system libraries in a straightforward fashion. The workflow for that is always the same: Write a function in C which fetches the arguments from the stack of the Lua interpreter and converts them into fixed C types. Using the fixed-type variables call the library function and receive a result, either as a return value or as an output argument. The result has to be converted back to a Lua type and pushed onto the stack of the Lua interpreter. Then hand the control back to the interpreter.

As we can already see, this recipe involves writing a lot of code, most of which is pure boilerplate. Wouldn't it be great if there was something which would just do all the type conversion work for us? And indeed there is, the FFI [3, 5, 8]. The concept of a Foreign Function Interface is not exclusive to Lua and also exists in other languages, e.g. with the `ctypes` library for Python.

Different FFIs have different ways of binding library functions. The Lua FFI chooses to parse plain C declarations. The advantage of this is that when interfacing with C libraries, you can copy and paste function prototypes from corresponding header files. Of course, the disadvantage is that for non-C libraries you have to come up with those prototypes yourself, which is not always an easy task. The FORTRAN language, for example, does not use the C-style *call by value* convention but always uses *call by reference*; that is to say, all types from a C function prototype would have to be converted to pointer types.

Thanks to Hans Hagen for very useful discussions.

2 The GNU Scientific Library

The GNU Scientific Library (GSL) [2] is a software library for scientific computing, implementing a broad range of algorithms. A complete list of algorithms is far too long to be presented here, and beyond the scope of this tutorial. We will only deal with the numerical integration routines here.

The numerical integration routines in the GSL are based on algorithms from the QUADPACK [9] package for adaptive Gauss-Legendre integration. In essence, each of the functions computes the integral

$$I = \int_a^b f(x)w(x)\,dx \qquad (1)$$

where $w(x)$ is a weight function. We will be focussing only on the case where the weight function $w(x) = 1$. Since an integral cannot be solved exactly by a computer, the user has to provide error bounds to indicate convergence.

3 Interfacing with the GSL

The first thing to do when we want to interface with an external library is load the FFI Lua module and use it to load the shared library of interest into memory.

```
local ffi = require("ffi")
local gsl = ffi.load("gsl")
```

3.1 C declarations

Next we have to add all the C declarations which are important for us. Let us first have a look over the code and then discuss why I wrote things the way they are.

```
ffi.cdef[[
typedef double(*gsl_cb)(double x, void *);

typedef struct {
  gsl_cb F;
  void *params;
} gsl_function;

typedef void gsl_integration_workspace;

gsl_integration_workspace *
 gsl_integration_workspace_alloc(size_t n);

void gsl_integration_workspace_free(
  gsl_integration_workspace * w);

int gsl_integration_qagiu(
  gsl_function *f,
  double a, double epsabs, double epsrel,
  size_t limit,
  gsl_integration_workspace *workspace,
  double *result, double *abserr);
]]
```

The first declaration introduces a new type, which I call `gsl_cb`, which stands for GSL callback. It is a pointer to a function which takes a floating point number and a void pointer and returns another floating point number. In reality, this function pointer will point to a Lua function representing the integrand, i.e. $f(x)$ in Eq. 1. We can ignore the unnamed second argument (`void *`) here because this is only relevant for the C interface of the GSL but we still have to declare it.

The next declaration is another type declaration, this time with the name `gsl_function`. It is a structure containing two values; the first is the function pointer to the integrand `F`, the second a pointer to some memory where parameters could be located. In our case we will not use the `params` field but we nevertheless have to declare it. What is very important is that the order of the fields in the structure is *exactly the same* as in the C header file. Otherwise the memory alignment of the field will be off and a segmentation fault will occur.

The last type declaration is for the identifier `gsl_integration_workspace`, which I simply make it an alias for `void`. Looking in the C header file of the GSL, we find that `gsl_integration_workspace` is defined as a structure with several fields, so why do we not declare those fields? The reason is simple: we don't care. As you will see we do not access any fields of `gsl_integration_workspace` from the Lua level and the GSL library already knows what the fields are. Therefore I decided to make `gsl_integration_workspace` *opaque*.

The next three declarations are all function declarations which are straight copies from the header file: `gsl_integration_workspace_alloc` allocates enough memory to perform integration using `n` subintervals; `gsl_integration_workspace_free` releases that memory back to the system; and the third function declaration, `gsl_integration_qagiu`, is the actual integration routine. It computes the integral of the function `f` over the semi-infinite interval from `a` to ∞ with the desired absolute and relative error limits `epsabs` and `epsrel` using at most `limit` subintervals which have been previously allocated in `workspace`. The final approximation and the corresponding absolute error are returned in `result` and `abserr` [10].

3.2 Lua interface

Now that we've declared all of the library functions it is time that we integrate this with Lua. To this end we write a function which nicely encapsulates all the lower level structure. The function is named `gsl_qagiu` and takes as parameters a (Lua) function `f` (which takes one argument), the lower limit of the integral `a`, and three optional arguments, the absolute error `epsabs`, the relative error `epsrel`, and the maximum number of subintervals `N`.

```
local gsl_f  = ffi.new("gsl_function")
local result = ffi.new("double[1]")
local abserr = ffi.new("double[1]")

function gsl_qagiu(f,a,epsabs,epsrel,N)
  local N = N or 50
  local epsabs = epsabs or 1e-8
  local epsrel = epsrel or 1e-8

  gsl_f.F = ffi.cast("gsl_cb",f)
  gsl_f.params = nil

  local w =
     gsl.gsl_integration_workspace_alloc(N)

  gsl.gsl_integration_qagiu(gsl_f, a,
     epsabs, epsrel, N,
     w, result, abserr)

  gsl.gsl_integration_workspace_free(w)
  gsl_f.F:free()

  return result[0]
end
```

We start by defining some local variables outside the function for better performance. We instantiate a new value of type `gsl_function` and two arrays of length one using the `ffi.new` method.

After processing the optional arguments, we set the fields `F` and `params`. This is where it gets interesting. Recall that the type of the field `F` is a pointer to a function which takes two arguments. Even though the Lua function `f` only takes one argument we can use it directly, because of the way Lua deals with optional arguments. If the number of arguments is less than the number of parameters passed to the function call, all the additional parameters are simply dropped. The only problem that we have left is that this is a Lua function, not a C function. To this end we use `ffi.cast` to cast the Lua function into a C function. It can also be converted implicitly by simply assigning `f`, but then it is less clear what is going on. At this point it is very important that the types of the arguments and the return value match, otherwise we will run into memory problems. Because the field `params` is unused we simply set it to the null pointer by assigning `nil`. (We could probably leave it unset but that is bad practice. Always initialize your variables!)

The result and the absolute error of the integration are returned as output arguments from the GSL function, i.e. the variables have to have pointer type. The easiest way to create a pointer to a value is by creating an array of length one of the desired type, which we already did outside the function. Arrays can be implicitly cast into pointers but at the same time live on the stack, so we do not have to worry about heap allocation and deallocation.

Next we use the previously declared functions to first allocate a workspace structure of sufficient size, then call the integration function with all of our arguments, releasing the workspace memory back to the system. You might notice that not all of the variables in the call to the integration routine have been created using `ffi.new`. This is indeed not necessary because the FFI will try to convert Lua values to native C types for you implicitly. Roughly speaking, you only have to use `ffi.new` for non-fundamental types or arrays.

There is one last subtlety to take care of. The library function to which we passed the function pointer is allowed to store that pointer for later use. Therefore this pointer will not decrease its reference count after exiting the function and therefore can never be garbage collected. We are probably not going to call this function so many times that this memory leak will have a huge impact but it is certainly good practice to release resources on exit, so we indicate to the garbage collector that this pointer can be cleaned up by calling its `free()` method.

Finally we return the result which is stored in the first element of the array. Note that C uses zero-based indexing.

3.3 Usage in pgfplots

So far we have only been implementing some kind of abstract skeleton for numerical integration. Now it is definitely time to actually use it. To this end we will plot the following complete Fermi-Dirac integral:

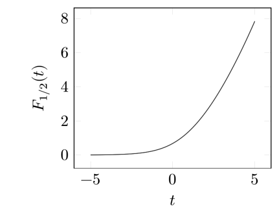
$$F_{1/2}(t) = \int_0^\infty \frac{x^{1/2}}{e^{x-t}+1} dx. \qquad (2)$$

What we will do now is call the `gsl_qagiu` routine with the integrand as the first argument and the lower limit as the second argument. Because we want to obtain the result of the integration in TeX we do not return the result of the integration but feed it back to TeX using `tex.sprint`.

```
function F_one_half(t)
  tex.sprint(gsl_qagiu(function(x)
      return math.sqrt(x)/(math.exp(x-t)+1)
    end, 0))
end
```

The last thing to do is plot this function using `pgfplots`. In the following I use ConTeXt syntax but the TeX and LaTeX syntax is very similar. It should be noted though, that for FFI to work in LaTeX, the `--shell-escape` option has to be enabled, because these operations are considered unsafe. First of all we need to tell Ti*k*Z about the Lua function. We do this using `declare function` and simply calling the Lua function with the argument. (A LaTeX user would use `\directlua` instead of `\ctxlua`.) There is still a slight problem. The `pgfplots` package uses its own representation for floating point numbers, called `fpu` [1], which is not compatible with Lua. There are ways to work around this (see the Appendix), but the simplest solution for the moment is simply turning off the `fpu` for this plot.

```
\starttikzpicture
 [declare function={
  F_one_half(\t) = \ctxlua{F_one_half(\t)};
 }]
 \startaxis[
  use fpu=false, % very important!
  width=6cm,
  no marks,
  samples=101,
  xlabel={$t$},
  ylabel={$F_{1/2}(t)$},
 ]
  \addplot{F_one_half(x)};
 \stopaxis
\stoptikzpicture
```



4 Conclusion

The availability of FFI in LuaTeX takes document processing to a completely new level. The possibility to interface with native C libraries allows for tasks which were previously intractable, such as the numerical integration in this tutorial. This article was inspired by a question asked on Stack Exchange, where a minimal working example of the techniques presented here can be found [7].

Another example would be the conversion of an image from SVG format to PDF without the generation of intermediate files, as I demonstrated in [6] using the Cairo and Rsvg-2 libraries.

Finally, Aditya Mahajan published an article on his ConTeXt blog on how to interface the Julia programming language with LuaTeX via the FFI [4].

5 Appendix

During the preparation of this manuscript I was made aware, by Aditya Mahajan, that the approach of turning off the `fpu` is not always a viable workaround; it can fail, for instance when trying to plot in logscale. Therefore one has to convert the function argument from `fpu` float to Lua number and the result from

Lua number to `fpu` float. Fortunately PGF provides macros to facilitate this conversion. Using those one can declare the function from the main text as follows:

```
\pgfmathdeclarefunction{F_one_half}{1}{%
  \pgfmathfloatparsenumber{%
  \ctxlua{
    F_one_half(\pgfmathfloatvalueof{#1})
  }%
 }%
}
```

One does not necessarily have to rely on the macro level here. As of version 3, the PGF package comes with a Lua backend for function evaluations which provides parser functions for `fpu` types. With this, one could adapt the Lua function from the main text as follows:

```
local plf = require"pgf.luamath.functions"

function F_one_half(t)
  local t = plf.tonumber(t)
  local result = gsl_qagiu(function(x)
     return math.sqrt(x)/(math.exp(x-t)+1)
    end, 0))
  tex.sprint(plf.toTeXstring(result))
end
```

References

[1] Christian Feuersänger. Floating point unit library. `https://ctan.org/pkg/pgf`, 2015.

[2] M. Galassi et al. *GNU Scientific Library Reference Manual*. Network Theory Ltd., third edition, 2009.

[3] Hans Hagen, Luigi Scarso, and Taco Hoekwater. LuaTeX 1.0.3 announcement. `https://tug.org/pipermail/luatex/2017-February/006345.html`, 2017.

[4] Aditya Mahajan. Interfacing LuaTeX with Julia. `https://adityam.github.io/context-blog/post/interfacing-with-julia/`, 2017.

[5] James R. McKaskill. LuaFFI. `https://github.com/jmckaskill/luaffi`, 2010–2013.

[6] Henri Menke. Answer to 'How to include SVG diagrams in LaTeX?' `https://tex.stackexchange.com/a/408014`, 2017.

[7] Henri Menke. Answer to 'Plot complete Fermi-Dirac integral in Lualatex'. `https://tex.stackexchange.com/a/403794`, 2017.

[8] Mike Pall. LuaJIT: FFI library. `http://luajit.org/ext_ffi.html`, 2005–2017.

[9] R. Piessens, E. de Doncker-Kapenga, C.W. Überhuber, and D.K. Kahaner. *Quadpack: A Subroutine Package for Automatic Integration*. Springer, 1983.

[10] The GSL Team. GNU Scientific Library: Numerical integration. `https://www.gnu.org/software/gsl/doc/html/integration.html`, 1996–2017.

◇ Henri Menke
9016 Dunedin
New Zealand
`henrimenke (at) gmail dot com`