A Julia wrapper for Apophenia

Ben Klemens

22 December 2013

Last time I expounded on Julia (entry #172) in code-free generalities. This time, I'll give a specific example, in which I use Apophenia's library of statistical models from a Julia front-end.

Conceptually, the big theme of this column is that Julia makes a real effort to talk to C structures, which means work defining the C-to-Julia type mapping, but makes the function calls themselves almost trivial. Nontrivial C code in the present day manipulates nontrivial structures, not just flat homogeneous arrays, so being able to call a function that takes in a struct opens up a world of already-written code for us to wrap in Julia functions.

The demo is a rehash of the demo code from a previous simple logit (entry #171), which was two lines of C code, mostly because I had a lot going on each line. Stretching out the process of explicitly opening an on-disk database, reading in the data to a table named `dta`, querying out the desired parts to a matrix-like structure, and estimating the statistical models looks like this in Julia:

Get this and the code below from the github repo[1] with all the code associated with this blog. Provided you have Apophenia (and its dependencies) and Julia installed, run it with

```
julia -L 173-ademo.jl
```

which leaves you at the Julia command prompt post-script to interrogate `out1`, `out2`, `oest`, `data`, et cetera.

The `q` is an `apop_data` struct, which is defined below. In most languages, this counts as an opaque pointer, meaning that it is pointing to data that is not in a native data type, so these

[1] `https://github.com/b-k/modeling_examples`

```
require("apop.jl")

db_open("test.db")
text_to_db("173-data.txt", "dta")
q = query_to_data("select out, input from dta")
data = data_as_array(q)
db_close()


oest = apop_estimate(q, "apop_ols")
out1 = data_as_vector(oest.parameters)

lest = apop_estimate(q, "apop_logit")
out2 = data_as_array(lest.parameters)
```

---

opaque pointers are usable only to pass back to C. Here they are just semi-opaque. For example, the `apop_model` Julia objects `oest` and `lest` include a parameter set, which we can view as a Julia-native vector (for OLS, where the parameter set is a vector) or an array (for multinomial Logit, where the parameter set can be a matrix).

The two strings describing the models in the above script, `apop_ols` and `apop_logit`, are the names of objects in the C object library. More on how this works below.

apop.jl The `apop.jl` file provides enough to run the above demo, but not much more. You, the reader, are welcome to pick it up and expand it to meet your needs. I'm reluctant to go much further with it without talking to somebody better versed in the customs of Julian code-writers.

I'll give some discussion first, then show the code file.

The first third of the file is a element-by-element mapping of C data structures. If you compare it to the C header file[2] where these types are declared, you'll see how most of this is line-by-line translation, as `int` in C becomes `Int32` in Julia, `char*` in C becomes `Ptr{Uint8}`, et cetera. After declaring a Julia type named `apop_name`, I can refer to `Ptr{apop_name}`. This may seem obvious, but this much is already impossible in a lot of C-hosting languages.

Currently, the `name` element of the `apop_model` is not a pointer (`char*`), but a fixed array (`char[101]`). I had to declare a throwaway type for a bitfield of 808 bytes (which I gave a throwaway name[3]). The struct also has several pointer-to-functions, which have `Ptr{Void}` placeholders in the Julia struct.

Having gone through the all-but-automatable process of mapping C struct elements to Julia types, we can call functions that make use of these types.

The first function, `db_open` is a good and simple example of the process of calling a C function, using `ccall`. The first argument to `ccall` is a pair of strings: the function name in the C library, and the name of the C library itself. Julia knows how to call the POSIX-standard `dlopen@`[4]function to open the library you name and use `dlsym` to find the function you named therein. Again, this seems really trivial, but other systems sometimes make this a lot of work.

The `db_open` function uses standard C types, taking in a string and returning an int, but if you skim further down, you'll see that functions that take a `Ptr{apop_data}` or `Ptr{apop_model}` behave identically. This works naturally given that we told Julia enough about the C structs to send them to a function, and this is yet another example of something that a lot of C-wrapping languages can't do.

You can't send a plain `apop_data` across the Julia-C border, but C functions usually take in a pointer-to-struct anyway, so this is bothersome but not fatal.

Other noteworthy bits:

- The `unsafe_load` function reads the data at a pointer as the given struct. Thus, we take in a `Ptr{apop_data}` and use the struct info declared in the top third of the file to look at its constituents.
- The `pointer_to_array` function takes a simple C array and wraps it as a Julia-native array.

[2] `https://github.com/b-k/Apophenia/blob/master/types.m4.h`
[3] `https://en.wikipedia.org/wiki/Roland_808`
[4] `http://linux.die.net/man/3/dlopen`

- The `cglobal` function is how we pull a single object, like the `apop_ols` model, out of a C library. There's some awkwardness about having a variable for the object name; see the Julia documentation[5] for an explanation of the workaround.

OK, here's the 130 lines of code it took to set up Julia for reading a text file to database, pulling query results to a Julia-native data set, estimating any of the dozens of statistical models that ship with the Apophenia library[6], and viewing the resulting parameters as Julia arrays:

[5] `http://docs.julialang.org/en/release-0.2/manual/calling-c-and-fortran-code/`
[6] `http://apophenia.info/group__models.html`

```
require("GSL")

type apop_name
      title::Ptr{Uint8}
      vector::Ptr{Uint8}
      col::Ptr{Ptr{Uint8}}
      row::Ptr{Ptr{Uint8}}
      text::Ptr{Ptr{Uint8}}
      colct::Int32
      rowct::Int32
      textct::Int32
end

type apop_data
      vector::Ptr{gsl_vector}
      matrix::Ptr{gsl_matrix}
      names::Ptr{apop_name}
      text::Ptr{Ptr{Ptr{Uint8}}}
      textsize1::Int32
      textsize2::Int32
      weights::Ptr{gsl_vector}
      more::Ptr{apop_data}
      error::Uint8
end

bitstype 101*8 backbeat

type apop_model
      name::backbeat
      vsize::Int32
      msize1::Int32
      msize2::Int32
      dsize::Int32
      #apop_settings_type *settings;
      data::Ptr{apop_data}
      parameters::Ptr{apop_data}
      info::Ptr{apop_data}
      # void (*estimate)(apop_data * data, apop_model *params);
      estimate::Ptr{Void}
      #long double (*p)(apop_data *d, apop_model *params);
      p::Ptr{Void}
      #long double (*log_likelihood)(apop_data *d, apop_model *params);
      log_likelihood::Ptr{Void}
      #long double (*cdf)(apop_data *d, apop_model *params);
      cdf::Ptr{Void}
      # long double (*constraint)(apop_data *data, apop_model *params);
      constraint::Ptr{Void}
      # void (*draw)(double *out, gsl_rng* r, apop_model *params);
      draw::Ptr{Void}
      # void (*prep)(apop_data *data, apop_model *params);
      prep::Ptr{Void}
      # void (*print)(apop_model *params, FILE *out);
      settings::Ptr{Void}                                    4
      more::Ptr{Void}
      more_size::Int32
      error::Uint8
end

function db_open(dbname::String)
   val = ccall( (:apop_db_open, "libapophenia"),
                  Int32, (Ptr{Uint8},), bytestring(dbname))
   if val == 1
      error("db_open: failed to open", dbname)
```