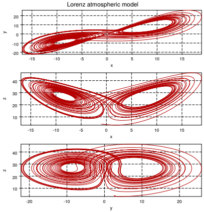
LGSL: Numerical algorithms for Lua

A Lua-ish interface to the GNU Scientific Library

Lesley De Cruz, Francesco Abbate, Benjamin von Ardenne

Lua devroom @ FOSDEM
ULB, Brussels, January 30, 2016



What is LGSL?

The LGSL module provides a friendly, **Lua-ish interface** to the **GNU Scientific Library (GSL)**.

It is based on the numerical modules of **GSL Shell**.

LGSL uses **FFI bindings** to the functions provided by the GSL shared library.

LGSL is a pure Lua(JIT) module: it requires **no compilation** (if the GSL library is present).

Thanks to LuaJIT and BLAS, LGSL is **blazingly fast***.

What is LGSL?

LGSL uses **FFI metatypes** to turn GSL primitives into featureful, garbage-collected Lua objects. For example:

- Matrices can be printed, inverted, multiplied, etc.

```
local matrix = require("lgsl.matrix")
local m1 = matrix.unit(2) - 1
local m2 = matrix.inv(m1)
print(m1*m2)
-- [ 1 0 ]
-- [ 0 1 ]
```

- Mix matrices and scalars, both complex and real.

```
print(m1*1i)
-- [    0 0-i ]
-- [ 0-i    0 ]
```

- Can be passed as arguments to GSL C functions such as

```
void gsl_matrix_set_identity (gsl_matrix * m);
```

What is LGSL?

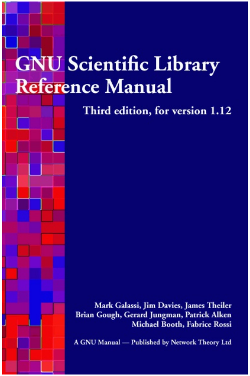
LGSL uses **FFI metatypes** to turn GSL primitives into featureful, garbage-collected Lua objects. For example:

- Complex number operators are fully supported, but have their own math library functions in the `lgsl.complex` module.

```
local complex = require("lgsl.complex")
print((1+1i)/(1-1i))
-- 0+1i
print(1i^1i)
-- 0.20787957635076+0i
print(complex.exp(1i*math.pi/4))
-- 0.70710678118655+0.70710678118655i
```

Why use the GNU Scientific Library?

- Well-written, ANSI C compliant code
- Well-tested by a comprehensive test suite and years of field-testing
- Well-documented, with an extensive Reference Manual (in print)
- Free as in Freedom (GPL): freely share your applications with others



According to the GSL website: “The interface was designed to be simple to link into very high-level languages, such as GNU Guile or Python.”

GNU Scientific Library: contents

LGSL implementation: Lua-ish interface / bare FFI bindings.

- Complex Numbers
- Special Functions
- Permutations
- BLAS Support
- Eigensystems
- Quadrature
- Quasi-Random Sequences
- Statistics
- N-Tuples
- Simulated Annealing
- Interpolation
- Chebyshev Approximation
- Discrete Hankel Transforms
- Minimization
- Physical Constants
- Discrete Wavelet Transforms
- Running Statistics
- Roots of Polynomials
- Vectors and Matrices
- Sorting
- Linear Algebra
- Fast Fourier Transforms
- Random Numbers
- Random Distributions
- Histograms
- Monte Carlo Integration
- Differential Equations
- Numerical Differentiation
- Series Acceleration
- Root-Finding
- Least-Squares Fitting
- IEEE Floating-Point
- Basis splines
- Sparse Matrices and Linear Algebra

GNU Scientific Library: contents

LGSL implementation: **Lua-ish interface** / bare FFI bindings.

- **Complex Numbers**
- **Special Functions**
- Permutations
- BLAS Support
- **Eigensystems**
- **Quadrature**
- Quasi-Random Sequences
- Statistics
- N-Tuples
- Simulated Annealing
- Interpolation
- Chebyshev Approximation
- Discrete Hankel Transforms
- Minimization
- Physical Constants
- Discrete Wavelet Transforms
- Running Statistics
- Roots of Polynomials
- **Vectors and Matrices**
- **Sorting**
- **Linear Algebra**
- **Fast Fourier Transforms**
- **Random Numbers**
- **Random Distributions**
- Histograms
- **Monte Carlo Integration**
- **Differential Equations**
- Numerical Differentiation
- Series Acceleration
- Root-Finding **(under review)**
- **Least-Squares Fitting**
- IEEE Floating-Point
- **Basis splines**
- Sparse Matrices and Linear Algebra

Why use LGSL? Example: Monte Carlo integration

C implementation with GSL:

```
#include <gsl/gsl_rng.h>
#include <gsl/gsl_monte_vegas.h>
#include <stdlib.h>
#include <gsl/gsl_math.h>
int main(void) {
  double res, err;
  int dim = 9;
  double xl[9] = { 0.,0.,0.,0.,0.,0.,0.,0.,0.};
  double xu[9] = { 2.,2.,2.,2.,2.,2.,2.,2.,2.};
  gsl_monte_function G = { &f, dim, 0 };
  size_t calls = 1e6*dim;
  gsl_rng_env_setup();
  gsl_rng *r = gsl_rng_alloc (gsl_rng_taus2);
  gsl_rng_set(r, 30776);
  gsl_monte_vegas_state *s = gsl_monte_vegas_alloc(dim);
  gsl_monte_vegas_integrate(&G, xl, xu, dim, 1e4, r, s, &res, &err);
  int i=0;
  do  {
    gsl_monte_vegas_integrate(&G, xl, xu, dim, calls/5, r, s,  &res, &err);
    i=i+1;
  }
  while(fabs(gsl_monte_vegas_chisq(s) - 1.0) > 0.5);
  printf("Result = % .10f\n", result);
  gsl_monte_vegas_free(s);
  gsl_rng_free(r);
  return 0;
}
```

Why use LGSL? Example: Monte Carlo integration

Lua implementation with LGSL:

```
local vegas = require("lgsl.vegas")
local matrix = require("lgsl.matrix")
math.randomseed(30776)
local n = 9
local calls = 1e6*n
local a = matrix.new(n,1)
local b = a + 2
local res = vegas.integ(f,a,b,calls)
print("Result = ", res.result)
```

* But what about callbacks? (Lua $\rightarrow$ C $\rightarrow$ Lua)

Thanks to the LuaJIT FFI, a **C function** can take a **Lua function** as a callback argument.

Unlike other calls to C functions via the LuaJIT FFI, these callbacks cannot be inlined/optimized.

Simply using FFI bindings for *e.g.* quadrature algorithms, ODE integrators, root finders... would carry a very high **performance penalty**!

Solution: re-implement the algorithms in pure **Lua**.

* But what about callbacks? (Lua $\rightarrow$ C $\rightarrow$ Lua)

Functions reimplemented in pure Lua:

- Quadrature
- Sorting
- Monte Carlo Integration
- Differential Equations
- Root-Finding (under review)

Re-implementing numerical routines in pure Lua

A naive implementation is already very fast!

Keeping in mind the guidelines (`http://wiki.luajit.org`)

- Locals, locals everywhere
- Cache often-used functions (but not FFI C functions)
- Minimize the number of live variables
- Prefer numeric `for` over `pairs/ipairs`
- Avoid unbiased branches
- Avoid nested loops or loops with low iteration counts

However, we would like to compete with C.

The last guideline (no short loops) is hard to combine with flexible code.

Re-implementing numerical routines in pure Lua

To speed things up even more:

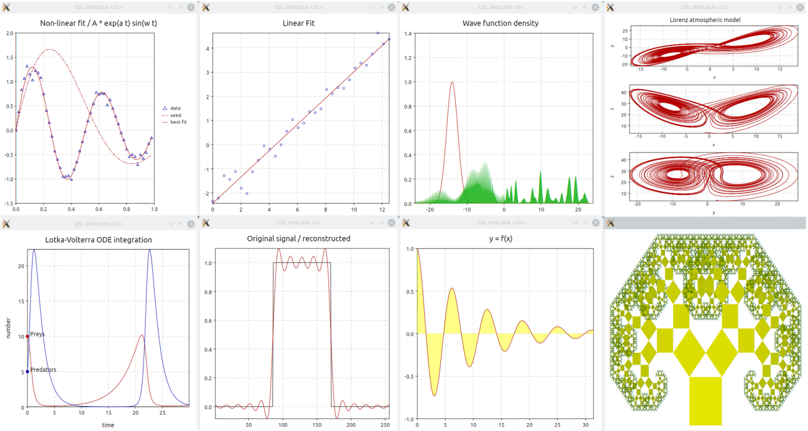
- Unroll loops with low iteration counts.
  LGSL uses Rici Lake's **template parser** for automatic loop unrolling. Example:

  ```
  -- k1 step of the 4th order Runge-Kutta
  -- ODE integration algorithm
  # for i = 0, N-1 do
  y_$(i) = y_$(i) + h / 6 * k_$(i)
  ytmp_$(i) = y0_$(i) + 0.5 * h * k_$(i)
  # end
  ```

  For very large ODE systems: `odevec` (under development)
- Use FFI arrays instead of Lua tables.

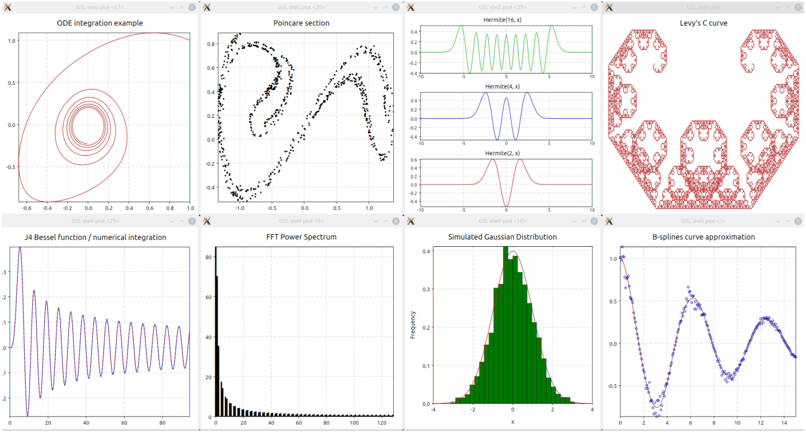
Visualisation

LGSL and **graph-toolkit** play well together.



Visualisation

LGSL and **graph-toolkit** play well together.



Installation (on Linux, e.g. Debian-derived)

LuaJIT

```
> git clone http://luajit.org/git/luajit-2.0.git
> cd luajit-2.0 && make && sudo make install
```

LuaRocks

```
> wget
http://luarocks.org/releases/luarocks-2.3.0.tar.gz
> tar xvzf luarocks-2.3.0.tar.gz
> cd luarocks-2.3.0
> ./configure && sudo make bootstrap
```

Recommended: graph-toolkit

```
> sudo apt-get install libagg-dev libfreetype6-dev
libx11-dev
> luarocks install --server=http://luarocks.org/dev
graph-toolkit
```

Installation (on Linux, e.g. Debian-derived)

GSL and LGSL

```
> sudo apt-get install libgsl0ldbl
> luarocks install lgsl
```

Documentation: `http://ladc.github.io/lgsl/`

GitHub: `http://www.github.com/ladc/lgsl/`

Pull requests welcome!