LuaJIT Numerical Computing for Quants *Minimalist Efficiency*

Stefano Peluchetti
sp@scilua.org

LUA

- **Dynamic:** execution model and types
- **Minimalist:** concepts, standard library, memory footprint, implementation (~20KLOC)
- **Easy**: high-level, consistent, well documented
- **Full-featured:** functional + OO programming
- **Embeddable:** trivial in C and others

LUA - WHAT?

- Relatively unknown in the finance industry but with a proven record:
    - Adobe Lightroom
    - CloudFlare, OpenResty/Nginx
    - World of Warcraft

LUA - SAMPLE 1

```lua
local function european(iscall, K)
  return function(S)
    if iscall then
      return math.max(S - K, 0)
    else
      return math.max(K - S, 0)
    end
  end
end

local call, put = european(true, 1.0), european(false, 2.0)
assert(call(1.5) == put(1.5))
```

LUA - SAMPLE 2

```lua
local function count(t)
  local out = { }
  for i=1,#t do
    local v = t[i]
    out[v] = out[v] and out[v] + 1 or 1
  end
  return out
end

local counted = count({ 'a', 'b', 'a', 10 })
for k,v in pairs(counted) do
  print(k, v)
end
--> b    1
--> a    2
--> 10   1
```

LUAJIT

- **Fast:** combines a high-speed interpreter (in assembler) with a trace JIT compiler
  - interpreted: 4x Lua (already fast!)
  - compiled: C performance
  - startup, warmup and compile times in the milliseconds range

- **FFI:** directly call C functions and use C data

```lua
local ffi = require 'ffi'
ffi.cdef('int printf(const char* ftm, ...)')
ffi.C.printf('Hello %s!\n', 'quants')
--> Hello quants!
```

LUAJIT - CLEVER

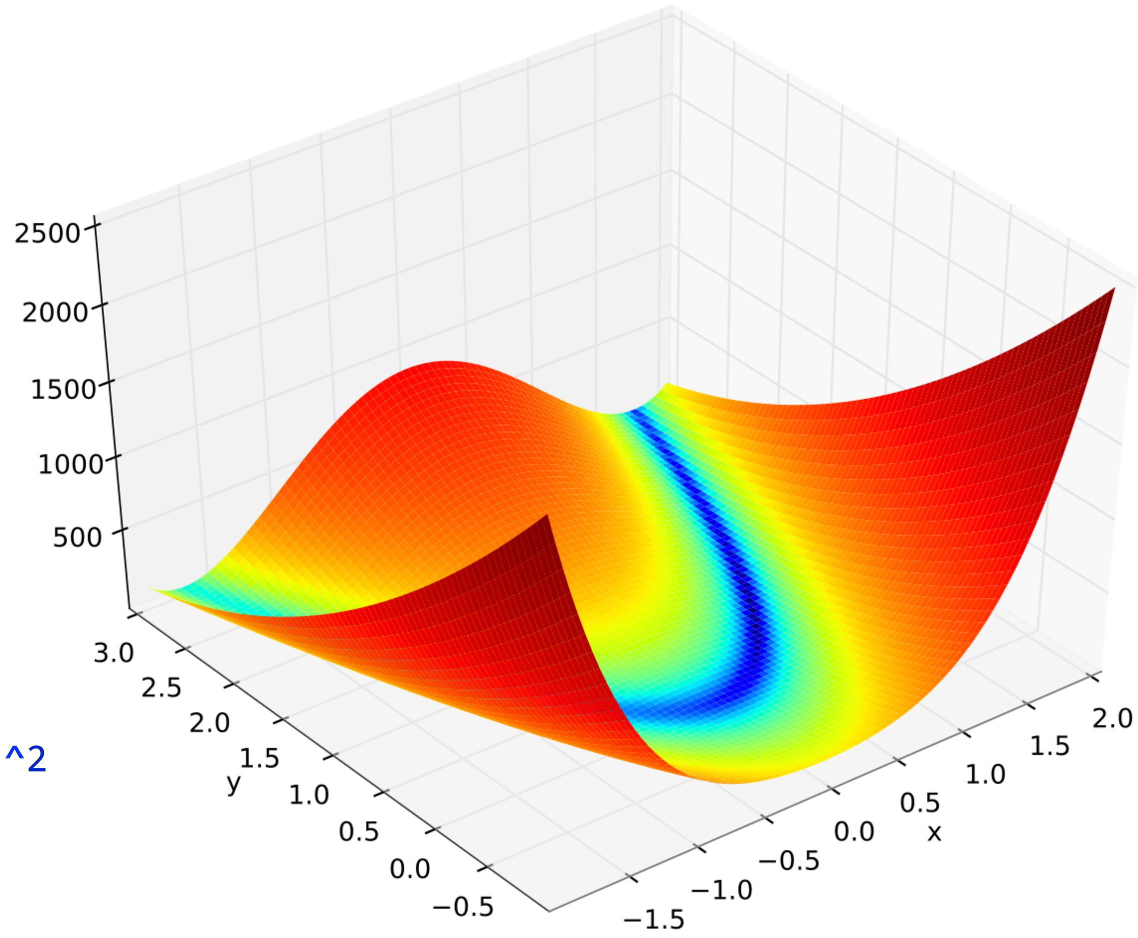
```
-- Allocation sinking:
local point
point = {
  new = function(self, x, y)
    return setmetatable({x=x, y=y}, self)
  end,
  __add = function(a, b)
    return point:new(a.x + b.x, a.y + b.y)
  end,
}
point.__index = point

local a, b = point:new(1.5, 2.5), point:new(3.25, 4.75)
for i=1,1e8 do
  a = (a + b) + b -- No allocations.
end

-- Many more: hoisting, CSE, … this loop is optimised away:
local x, n = 1,1e8
for i=1,n do
  x = math.abs(x)
end
```

SCILUA

- General purpose scientific computing framework
- C performance + Lua ease of use
- Focus on state-of-the-art algorithms
    - optimisation: adaptive Differential Evolution
    - MCMC: adaptive Hamiltonian Dynamics MC
- Syntax extensions for simpler Linear Algebra

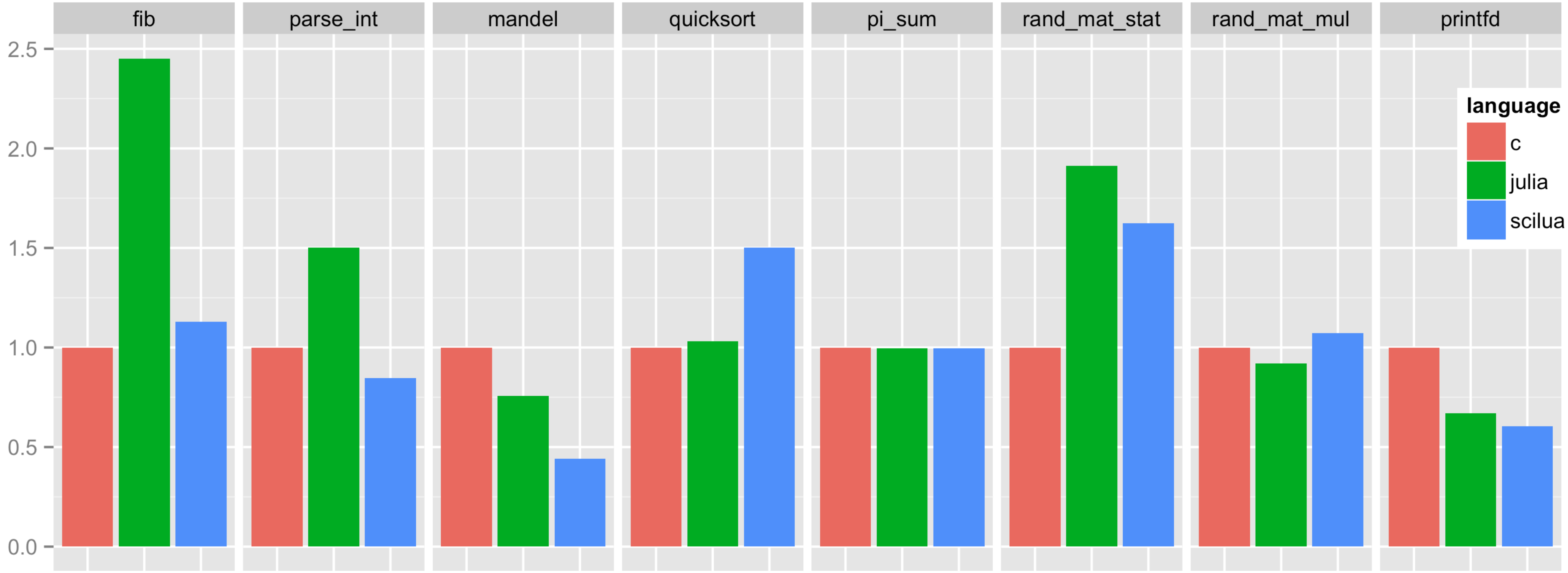
SCILUA - SAMPLE

```lua
-- Load required modules:
local alg  = require 'sci.alg'
local fmin = require 'sci.fmin'

local N = 6 -- Number of dimensions.

-- Wide initial range:
local xl, xu = alg.vec(N), alg.vec(N)
for i=1,N do
  xl[i], xu[i] = -100, 100
end

-- Rosenbrock function:
local function f(x)
  local sum = 0
  for i=1,N-1 do
    sum = sum + (1 - x[i])^2 + 100*(x[i+1] - x[i]^2)^2
  end
  return sum
end

-- Differential Evolution:
local xm, fm = fmin.de(f, {
  xl   = xl,
  xu   = xu,
  stop = 1e-6,
})
```



```
--> argmin               : +1.000000,+1.000000,+1.000000,+1.000000,+1.000000,+1.000000
--> f(argmin)            : +9.99e-15
--> CPU seconds fmin.de : +0.009713
--> CPU seconds total    : +0.026021
```

SCILUA - PERFORMANCE



Time required (lower is better) to run the Julia benchmark suite. Timings are relative to the C implementation.
Linux x64 / gcc 5.2.1 / Julia 0.4.1 / SciLua 1.0-beta10

SCILUA - SYNTAX

```
-- LuaJIT with SciLua syntax extensions:
local function randmatstat(t)
    local n = 5
    local v, w = vec(t), vec(t)
    for i=1,t do
        local a, b, c, d = randn(n, n), randn(n, n), randn(n, n), randn(n, n)
        local P = join(a..b..c..d)
        local Q = join(a..b, c..d)
        v[i] = trace((P[]`**P[])^^4)
        w[i] = trace((Q[]`**Q[])^^4)
    end
    return sqrt(var(v))/mean(v), sqrt(var(w))/mean(w)
end

# Julia:
function randmatstat(t)
    n = 5
    v = zeros(t)
    w = zeros(t)
    for i=1:t
        a = randn(n,n)
        b = randn(n,n)
        c = randn(n,n)
        d = randn(n,n)
        P = [a b c d]
        Q = [a b; c d]
        v[i] = trace((P.'*P)^4)
        w[i] = trace((Q.'*Q)^4)
    end
    return (std(v)/mean(v), std(w)/mean(w))
end
```

SCILUA - MODULES

Module	Description
sci.alg	vector and matrix algebra
sci.diff	automatic differentiation
sci.dist	statistical distributions
sci.fmin	function minimization algorithms
sci.fmax	function maximization algorithms
sci.math	special mathematical functions

Module	Description
sci.mcmc	MCMC algorithms
sci.prng	pseudo random number generators
sci.qrng	quasi random number generators
sci.quad	quadrature algorithms
sci.root	root-finding algorithms
sci.stat	statistical functions

ECOSYSTEM - WHAT IS BAD?

- No-batteries-included
- Lua releases not 100% backward compatible
- Tendency to reinvent the wheel
- No “official” centralised package repository
    - de facto standard is LuaRocks, source-based

ULUA - A NEW DISTRIBUTION

- A Lua binary distribution for Windows, OSX and Linux based on LuaJIT (x86 and x64)
- Portable: unzip and run!
- LuaRocks integration: more than 300 imported packages automatically kept up to date
- Easy to use package manager

```
upkg available  # List all available packages.
upkg add sci    # Download and install the sci library.
upkg update     # Update the whole distribution.
```

THE OTHERS

	LuaJIT	Python	R	Julia	Matlab	C++
Speed	C	C/50	C/200	C	depends	C
Eco System	basic	good but messy	good but not in R	good	expensive	which?
Style	loop and vectorised	better be vectorised	better be vectorised	loop and vectorised	better be vectorised	loop
Maturity	senior	senior but splitted	senior but messy	medium and evolving	senior but evolving	super-senior (and complex)
Compile	JIT, fast	JIT, medium	not (mostly)	JIT, slow	not (mostly)	ahead
Size	0.5MB	80MB (PyPy)	120MB	250MB	5GB	compiled and linked #

SUMMARY - GOOD FOR?

- Payoff scripting
- Model calibration
- Distributed (stateless) computing
- Embedding in Excel AddIn
- Textual and numerical data manipulation

REFERENCES

- **Lua** (the language):
  - http://lua.org
- **LuaJIT** (the implementation):
  - http://luajit.org
- **ULua** (the distribution):
  - http://ulua.io
- **SciLua** (the numerics):
  - http://scilua.org

Q&A