# Methodical Accelerator Design
## Next Generation Technologies
### 2nd CERN Developper Workshop

**Laurent Deniau**

**CERN-BE/ABP**

**30th May 2016**

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# Content


I. Road to MAD

II. Technologies overview for MAD

III. MAD performance overview

IV. Lua(JIT) technology

V. MAD technology

VI. MAD examples


Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# What is MAD?

- MAD: Methodical Accelerator Design *(~1000 users, ref. for CAS)*
  - ➡ design, studies and optimisation of beam physics for particle accelerators
  - ➡ all-in-one family of applications started in the 70's and 80's
  - ➡ MAD8 1984-2002 *(F77) developed for LEP*
  - ➡ MAD9 + Lib. Classic 1994-2004 *(C++) developed for HEP community*
  - ➡ **MAD-X** 2000- *(F77, F90, C, C++) for LHC (scripts: ~8 millions lines).*
  - ➡ **MAD**-*NG* 2015- *(C, Lua), developed for accelerator beam physics community.*

- MAD code specificities
  - ➡ user input scripts: highly dynamic, **injected deeply inside physics calculations**
  - ➡ complex scientific code: ~80% of the code is for **scientific computations** *(~160 KSLOC)*
  - ➡ ***prototypical language without delegation***: clone+update objects ➠ dictionary
  - ➡ ***deferred expressions***: lazy evaluation everywhere (!) for sharing and linking
  - ➡ « abstractions » make string lookup, expr. evaluation and memory management ***critical***
  - ➡ global workspace: everything is global, no scope *(deferred expr. ➠ lambda without parameters)*
  - ➡ context dependent syntax and semantic: command specific « parser » and behavior…
  - ➡ no function, weak macro system: unexpected behavior…

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# What should be MAD?

- MAD: Methodical Accelerator Design
  - ➡ design, studies and optimisation of beam physics for particle accelerators
  - ➡ all-in-one family of applications started in the 70's and 80's
  - ➡ MAD8 1984-2002 *(F77) developed for LEP*
  - ➡ ~~MAD9 + Lib. Classic 1994-2004 *(C++) developed for HEP community*~~
  - ➡ **MAD-X** 2000- *(~~F77, F90~~, C, C++) for LHC (scripts: ~~~8 millions lines~~).*
  - ➡ MAD-*NG* 2015- *(Lua, C), developed for accelerator beam physics community.*

- MAD**-NG** code specificities
  - ➡ user input scripts: highly dynamic, **injected deeply inside physics calculations**
  - ➡ complex scientific code: ~80% of the code is for **scientific computations** ~~*(~160 KSLOC)*~~
  - ➡ ***prototypical language with~~out~~ delegation***: clone+update objects ➠ dictionary
  - ➡ ***deferred expressions***: lazy evaluation ~~everywhere (!)~~ for sharing and linking
  - ➡ « **abstractions** » make string lookup, expr. evaluation and memory management ***critical***
  - ➡ ~~global workspace: everything is global, no scope *(deferred expr. ➠ lambda without parameters)*~~
  - ➡ ~~context dependent syntax and semantic: command specific « parser » and behavior…~~
  - ➡ ~~no function, weak macro system: unexpected behavior…~~

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# Objectives of MAD-NG

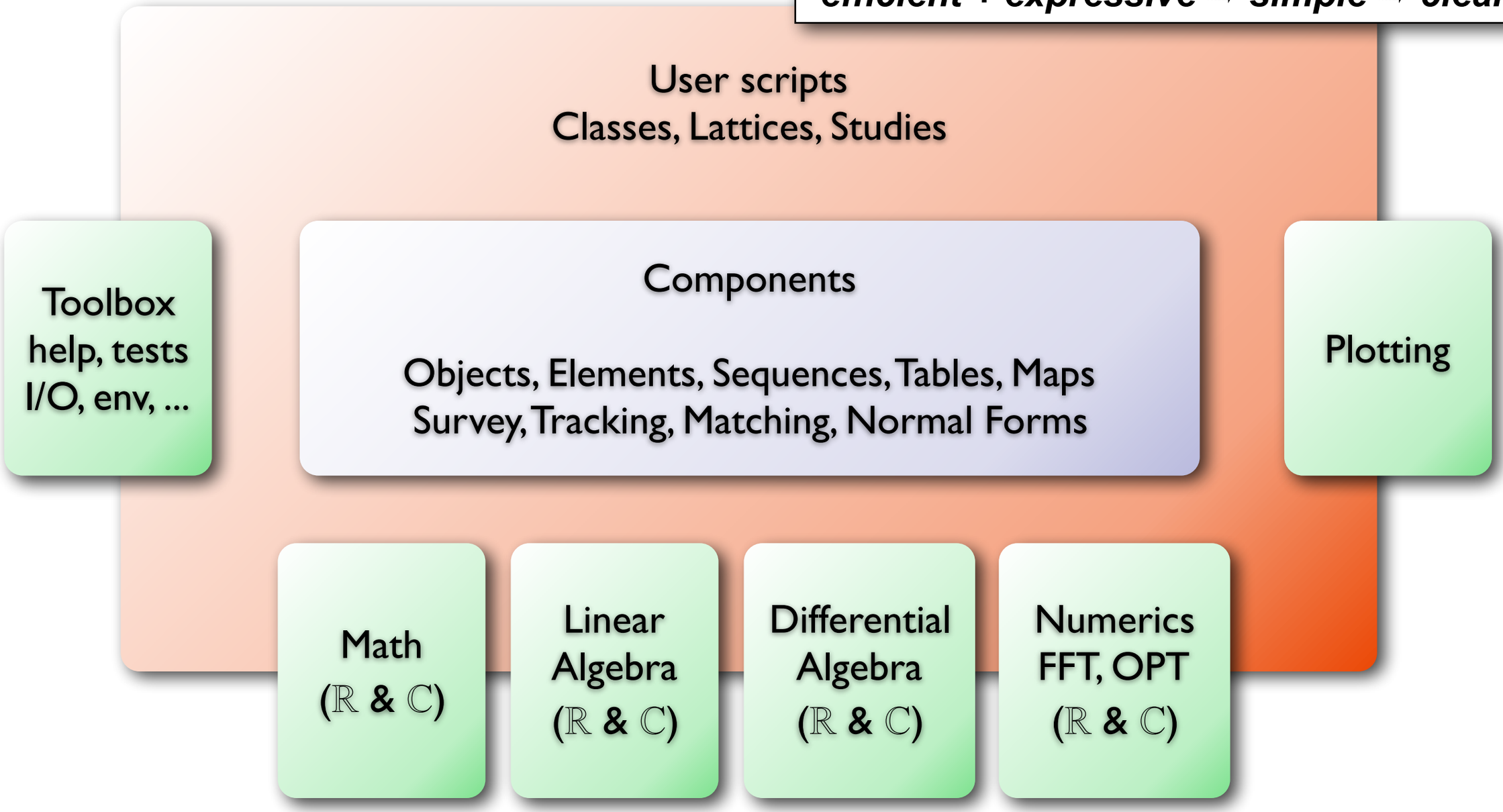
- Long term design and **easy to use**
  - ➡ Safe, correct and accurate ➠ closed to changes, open to extensions and compositions
  - ➡ Flexible ➠ easy scripting language to implement new features (avoid backdoor)
  - ➡ Extensible ➠ good architecture to extend existing features (avoid special cases)
  - ➡ Efficient ➠ performance of MAD-X or better
- Robust, consistent and **easy to support**
  - ➡ Expressive ➠ smaller code, easier to read, understand and document
  - ➡ Partitioned ➠ less coupling, less dependencies easier to maintain
  - ➡ Functional ➠ less globals, options, attributes and side effects (avoid headaches)
  - ➡ License GPLv3 ➠ all modifications (improvements) must be submitted to the authors
- Portable and **easy to maintain**
  - ➡ Simple technologies ➠ simple to build, test, install and run (**LNX,OSX,WIN**)
  - ➡ Portable technologies ➠ same code gives same results everywhere (**LNX,OSX,WIN**)
  - ➡ Flexible technologies ➠ reduce updates and releases, **self-contained**

*Productivity must be about x2-5 higher than for previous MADs*
*Application must be a drag-n-drop standalone binary*
➠ *importance to use simple, flexible, efficient and portable* ***technologies***

# Schematic layout

- Red area should be entirely written in the scripting language
  - ➡ Implement >80% of the features with **full** “monitored” access to physics and algebra!
  - ➡ Importance of the scripting language!! ➠ **clear, simple, expressive and efficient**

***efficient + expressive ➠ simple ➠ clear***



# Content

I. Road to MAD

**II. Technologies overview for MAD**

III. MAD performance overview

IV. Lua(JIT) technology

V. MAD technology

VI. MAD examples

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# Technologies overview (static: Parser+VM+JIT)

- *Simple* for ***user***, *simple* for ***developper***, *simple* for ***support***, *efficient* enough for ***physics***.
- Candidates: C, C++, Fortran, Java.
  - ➡ C: baseline, simple, stable, fast, well supported, well known, portable.
    - ± low level, – operator overloading, ± poor standard library (pros & cons!), ± GC.
  - ➡ C++: (*my favorite*) powerful for generative prog., complex for dynamic prog.
    - what about portability, productivity, dependencies and performance
    - rely on **SFINAE, CRTP, template expressions, type erasure, visitors, observers, alloc**.
    - support & technology: GCC (good asm), LLVM (good inlining), Intel (best, stable?).
    - **BE-ABP attempts** switched to Python, Octave, Mathematica…
  - ➡ Fortran: simple, fast, moderately expressive, not so supported, not so portable.
    - not so appealing, + operator overloading, not dead (std 2003, ~~2008, 2015~~)…
  - ➡ Java: simple, ~fast, GC, not so expressive (math), well supported, portable (?).
    - – heavy runtime, – operator overloading, + interface (dyn.), ± rich SDK (pros & cons!)

*none of them are good in all aspects*

# Technologies overview (dynamic: Embedded+DSL)

- *Simple* for ***user***, *simple* for ***developper***, *simple* for ***support***, *efficient* enough for ***physics***.
- Candidates: Python, Lua, Javascript, Julia, (Ruby, Perl, Tcl, R, Root).
  - ➡ Python: simple, too slow for physics, well supported & known, moderately portable.
    - large community, many libraries, “easy” connection to C & C++, not easy to embed.
    - powerful « glue » language, very dynamic, **most used & known in BE-ABP**.
    - JIT: PyPy(CFFI) is running after LuaJIT(FFI)… integration? portability? future?
  - ➡ Lua: **the simplest**, faster than Python, well supported, very portable.
    - « smaller » community (0.1), many libraries, easy connection to C, easy to embed.
    - JIT: one of the best ever, compete with C and C++ and beat Java in most cases.
    - JIT: provide the best FFI interface ever, tracing JIT, include a profiler and a tracer.
  - ➡ Javascript: simple, faster than Python (V8), well supported & known, very portable, not suitable for scientific computing.
  - ➡ Julia: rising scripting language (dynamic+static) in scientific computing, not yet mature, rely on LLVM technologies (JIT), support multi-methods (!), heavy FFI, future?

*none of them are good in all aspects*

# Technologies overview (exotic)

- *Simple* for ***user***, *simple* for ***developper***, *simple* for ***support***, *efficient* *enough for* ***physics***.
- Candidates: D, Scala, Haskell, COS, (Go, Swift).
  - ➡ D (2) is between C++ and Java, not well known, not well supported, future?
  - ➡ Scala (2) is a much better “Java” (same VM), more expressive, more complex, operator overloading, functional & object oriented, traits, rich type system (parametric+variance +constraint+existential) ➠ not so known (EPFL!) and probably as complex as C++.
  - ➡ Haskell (2010): (*one of my favorite*) good in most points, powerful, rather exotic and complex for common users and developers, low performance if low expertise.
  - ➡ C Object System: (*one of my favorite*) good in most points, simple and powerful pure C solution but *single person technology*, no operator overloading, no community. Potential candidate for an « all in C » simple and efficient solution.

*exotic technologies should be envisaged in last resort*

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# Technologies overview (combinations)

- Python + C *(**motivation**: widely known/used at CERN and BE-ABP, huge community)*
  - ➡ Python is too slow for the physics ➠ 80% of the code in C / C++ ➠ resources…
  - ➡ Python is a high level user « glue » language ➠ callback ➠ sandboxing…
  - ➡ Python relies on many libraries ➠ problem of portability and integration ➠ packaging…
  - ➡ Python relies on many libraries ➠ maintenance of dependencies ➠ packaging…
  - ➡ JIT: is PyPy(CFFI) good enough? How to integrate everything? Libraries compatibility?

- LuaJIT + C *(**motivation**: simplicity, performance, integration, portability)*
  - ➡ Lua: simple, dynamic, can be generative, prototypical, functional, object oriented, …
  - ➡ LuaJIT is fast enough for everything ➠ 20% in C, no callback, easy sandboxing…
  - ➡ LuaJIT is less known/used (small community) ➠ fallback: PyPy(CFFI) + C
  - ➡ Integration of C is trivial with LuaJIT FFI ; very good productivity if combined with Lua.
  - ➡ C is good for low level, vectorized and parallel computations (SSE, OMP, MPI, GPU).

***LuaJIT + C** considered as the best compromise ➠ selected solution*

# Content

I. Road to MAD

II. Technologies overview for MAD

**III. MAD performance overview**

IV. Lua(JIT) technology

V. MAD technology

VI. MAD examples

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# Performances *(containers: dictionary)*

**« library » class**

*C++ GCC 4.8 : 21 sec*

```
template <typename T>
T find_duplicates(const T& inp)
{
  std::unordered_map<typename T::value_type, int> map;
  T res;

  for (auto &item : inp) map[item] += 1;
  for (auto &item : inp) {
    auto &count = map[item];
    if (count > 1)
      res.push_back(item), count=0;
  }
  return res;
}

int main()
{
  std::vector<std::string> inp, out;
  inp = {"b","a","c","c","e","a","c","d","c","d"};

  for (int i=1; i<10000000; i++)
    out = find_duplicates(inp);

  std::cout << "{";
  for (auto &a : out)
    std::cout << "\"" << a << "\",";
  std::cout << "}" << std::endl;
}
```

*C++ Intel 16 : 26.4 sec*

*C++ LLVM 3.7 : 26.7 sec*

**« native » type**

*LuaJIT 2.1 : **7.2** sec*

```
local function find_duplicates(inp)
  local res = {}
  for _,v in ipairs(inp) do
    res[v] = res[v] and res[v]+1 or 1
  end
  for _,v in ipairs(inp) do
    if res[v] > 1 then
      res[#res+1], res[v] = v, 0
    end
  end
  return res
end

local inp, out
inp = {'b','a','c','c','e','a','c','d','c','d'}

for i=1,1e7 do
  out = find_duplicates(inp)
end

io.write('{"',table.concat(out,'", "'),'"}\n')
```

***output: { "a","c","d",}***



Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# Performances *(abstraction of small size types: complex)*

| | « library » class | « native » type |
|---|---|---|
| « interpreted » | MAD (LJ2.1) : **5.3** sec | Julia 0.4 : 35 sec |
| | Python 3.5 : 2 hr or 522 sec | |
| | PyPy 4.0 : **10.2** or **7.6** sec | |
| « compiled » | C++ GCC 4.8 : **6.8** sec | C GCC 4.8 : **5.7** sec |
| | C++ Intel 16 : **5.7** sec | C Intel 16 : **5.3** sec |
| | C++ LLVM 3.7 : 20.4 sec | |

**OK within x2**

```
function f(z)
  return z^2 + (1+1i)*z - 1
end

local z = 0.1
for i in ipairs(1..1e9) do
  z = f(z)
end
print(z)
```

```
def f(z):
  return z*z + (1+1j)*z - 1

z = 0.1
for i in range(1000000000):
  z = f(z)

print(z)
```

```
function f(z)
  return z*z + (1+1im)*z - 1
end

local z = 0.1
for i=1:1000000000
  z = f(z)
end
println(z)
```

```
inline std::complex<double>
f(const std::complex<double> &z)
{
  return z*z + (1.0+I)*z - 1.0;
}

int main()
{
  std::complex<double> z = 0.1;
  for (int i=1; i<1000000000; i++)
    z = f(z);
  std::cout << z << std::endl;
  return 0;
}
```

```
static inline complex double
f(complex double z)
{
  return z*z + (1+I)*z - 1;
}

int main(void)
{
  complex double z = 0.1;
  for (int i=1; i<1000000000; i++)
    z = f(z);
  printf("%+g%+gi\n", creal(z),cimag(z));
  return 0;
}
```

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# Performances *(abstraction of medium size types: matrix)*

```
I = matrix.eye(n)

def f(z):
  return z*z + (1+1j)*z - I

z = I*0.1
for i in range(10000000):
  z = f(z)

print(z)
```

*PyPy 4.0 + NumPy : [4x4] 162 sec*
*[6x6] 188 sec*
*[8x8] 220 sec*

***« library » class***

*C++ + Eigen 3.3 : [4x4] **6.0** sec*
*[6x6] **9.4** sec*
*[8x8] **15.1** sec*

```
local I = matrix(n):eye()

function f(z)
  return z^2 + (1+1i)*z - I
end

local z = I*0.1
for i in ipairs(1..1e7) do
  z = f(z)
end
print(z)
```

*will be better x2-4 soon!*

*MAD (LJ2.1) : [4x4] **9.2** sec*
*[6x6] **16.5** sec*
*[8x8] **29.7** sec*

```
static CMatrix I;

inline CMatrix f(const CMatrix &z)
{
  return z*z + Cpx(1,1)*z - I;
}

int main()
{
  I = CMatrix::Identity(nr, nc);
  CMatrix z = I*0.1;

  for (int i=0; i < 10000000; i++)
    z = f(z);

  std::cout << z << std::endl;
}
```

```
local I, r = matrix(n):eye(), cmatrix(n)

function f(z)
  z:mul(z, r)
  return z:mul(1+1i,z):sub(I,z):add(r,z)
end

local z = cmatrix(n):eye(0.1)
for i in ipairs(1..1e7) do
  z = f(z)
end
print(z)
```

*MAD (LJ2.1) : [4x4] **2.3** sec*
***no** allocation [6x6] **6.5** sec*
*[8x8] **16.4** sec*

*C++ + Eigen 3.3 : [4x4] **1.5** sec*
***no** allocation [6x6] **3.9** sec*
*[8x8] **13.9** sec*

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# Performances *(abstraction and algorithms: solver)*

```
m = np.matrix([[…],…,[…]])  # [8x8]
a = m[:nr,:nc].T            # *1j
b = m[:nc,:nr]

for i in range(1000000):
  c = solve(b,a)    # lstsq(b,a)[0]

print(c.T)
```

*« library » class*

*PyPy 4.0 + NumPy :* real
*[4x4] 29 sec*
*[6x6] 31 sec*
*[8x8] 32 sec*

*PyPy 4.0 + NumPy :* complex
*[4x4] 39 sec*
*[6x6] 43 sec*
*[8x8] 47 sec*

*PyPy 4.0 + NumPy :* real
*[3x4] 95 sec*
*[5x6] 101 sec*
*[7x8] 120 sec*

*PyPy 4.0 + NumPy :* complex
*[3x4] 251 sec*
*[5x6] 269 sec*
*[7x8] 291 sec*

**Price of heavy all-in-one data structure for medium size problems…**

```
local m = matrix{{…},…,{…}} -- [8x8]
local a = m:sub(1..nr,1..nc) -- *1i
local b = m:sub(1..nc,1..nr):t()
local c

for i=1,1e6 do
  c = a/b
end
print(c)
```

*MAD (LJ2.1) :* real
*[4x4] **1.1** sec*
*[6x6] **2.1** sec*
*[8x8] **3.6** sec*

*MAD (LJ2.1) :* complex
*[4x4] **1.5** sec*
*[6x6] **3.1** sec*
*[8x8] **5.4** sec*

*MAD (LJ2.1) :* real
*[3x4] **2.5** sec*
*[5x6] **4.5** sec*
*[7x8] **7.4** sec*

*MAD (LJ2.1) :* complex
*[3x4] **4.0** sec*
*[5x6] **7.6** sec*
*[7x8] **12.6** sec*

```
int main()
{
  CMatrix m(8,8);
  m << … ; // [8x8]                          // *I;
  CMatrix a = m.block(0,0,nr,nc).transpose();
  CMatrix b = m.block(0,0,nc,nr);
  CMatrix c;

  for (int i=0; i < 1000000; i++)
    c = b.fullPivHouseholderQr().solve(a);

  std::cout << c.transpose() << std::endl;
}
```

*C++ + Eigen 3.3 :* real
*[4x4] **3.5** sec*
*[6x6] **6.7** sec*
*[8x8] **8.0** sec*

*C++ + Eigen 3.3 :* complex
*[4x4] **4.4** sec*
*[6x6] **7.7** sec*
*[8x8] **12.3** sec*

*C++ + Eigen 3.3 :* real
*[3x4] **3.3** sec*
*[5x6] **5.2** sec*
*[7x8] **7.3** sec*

*C++ + Eigen 3.3 :* complex
*[3x4] **3.8** sec*
*[5x6] **6.8** sec*
*[7x8] **11.1**sec*

# MAD matrix module *(description)*

*MAD online help for matrix module*

```
NAME
  matrix (vector, cvector, cmatrix)

SYNOPSIS
  local matrix = require 'matrix'
  local m1 = matrix(3)                        -- column matrix = matrix(3,1)
  local m2 = matrix(2,3)
  local m3 = matrix {{1,2},{3,4},{5,6}}
  local m4 = matrix {1,2,3,4,5,6}             -- column matrix = {{1},{2},...}
  local m5 = matrix {{1,2,3,4,5,6}}           -- row matrix
  local m6 = m1:transpose()                   -- row matrix
  local I6 = matrix(6):ones()                 -- 6x6 identity

DESCRIPTION
  The module matrix implements the operators and math functions on matrices:
    (minus) -, +, -, *, /, %, ^, ==, #, [], ..,
    unm, add, sub, mul, div, mod, pow, emul, ediv,
    rows, cols, size, sizes, tsizes, get, set, get0, set0,
    zeros, ones, eye, fill, copy, same, tsame,
    get_row, get_col, get_diag, get_sub,
    set_row, set_col, set_diag, set_sub,
    transpose, t, trans, ctrans, conjugate,
    real, imag, conj, unit, norm, angle, trace, tr,
    dot, inner, cross, mixed, outer,
    abs, arg, exp, log, pow, sqrt, proj,
    sin, cos, tan, sinh, cosh, tanh,
    asin, acos, atan, asinh, acosh, atanh,
    solve, svd, eigen,
    fft, ifft, rfft, irfft, conv, corr, covar,
    foldl, foldr, foreach, map, map2, maps,
    concat, reshape, tostring, totable, fromtable,
    check_bounds.

REMARK:
  By default, check_bounds is true.

RETURN VALUES
  The constructor of matrices.
```

***Implemented in only 1000 lines of pure Lua + 1500 lines of C.***

**With High Order Functions at full speed, there is no need for many more features program what you need**

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# Content

I. Road to MAD

II. Technologies overview for MAD

III. MAD performance overview

**IV. Lua(JIT) technology**

V. MAD technology

VI. MAD examples

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# Lua overview (http://www.lua.org)


about
news
get started
download
documentation
community
contact
site map
português

Lua 5.3.2
released

*Programando em Lua*
published

Lua Workshop 2016
to be held in San Francisco

PUC RIO

***Learn Lua in 15 minutes***
***http://tylerneylon.com/a/learn-lua/***

**Lua is a powerful, fast, lightweight, embeddable scripting language.**

Lua combines simple procedural syntax with powerful data description constructs based on **associative arrays** and **extensible semantics**. Lua is **dynamically typed**, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it **ideal for configuration, scripting, and rapid prototyping**.

Lua has been used in many industrial applications (e.g., Adobe's Photoshop Lightroom), with an emphasis on embedded systems (e.g., the Ginga middleware for digital TV in Brazil) and games (e.g., World of Warcraft and Angry Birds). Lua is currently the leading scripting language in games. **Lua has a solid reference manual and there are several books about it.** Several versions of Lua have been released and used in real applications since its **creation in 1993**. Lua featured in HOPL III, the Third ACM SIGPLAN History of Programming Languages Conference, in June 2007. Lua won the Front Line Award 2011 from the Game Developers Magazine.

***Reference manual is 28 p. — 51 p. with C API***

# LuaJIT overview (http://www.luajit.org)

LuaJIT is a **Just-In-Time Compiler** (JIT) for the Lua programming language. Lua is a powerful, dynamic and light-weight programming language. It may be embedded or used as a general-purpose, stand-alone language.

LuaJIT is Copyright © 2005-2015 Mike Pall, released under the MIT open source license.

## Compatibility

| | | | | |
|---|---|---|---|---|
| Windows | Linux | BSD | OSX | POSIX |
| Embedded | Android | iOS | | |
| PS3 | PS4 | PS Vita | Xbox 360 | |
| GCC | CLANG LLVM | MSVC | | |
| x86 | x64 | ARM | PPC | e500 | MIPS |
| Lua 5.1 API+ABI | + JIT | + BitOp | + FFI | Drop-in DLL/.so | |

## Overview

| | | | | | |
|---|---|---|---|---|---|
| 3x - 100x | 115 KB VM | 90 KB JIT | 63 KLOC C | 24 KLOC ASM | 11 KLOC Lua |

LuaJIT has been successfully used as a **scripting middleware** in games, appliances, network and graphics apps, numerical simulations, trading platforms and many other specialty applications. It scales from embedded devices, smartphones, desktops up to server farms. It combines high flexibility with high performance and an unmatched **low memory footprint**.

LuaJIT has been in continuous development since 2005. It's widely considered to be **one of the fastest dynamic language implementations**. It has outperformed other dynamic languages on many cross-language benchmarks since its first release — often by a substantial margin.
For **LuaJIT 2.0**, the whole VM has been rewritten from the ground up and relentlessly optimised for performance. It combines a **high-speed interpreter**, written in assembler, with a **state-of-the-art JIT compiler**.

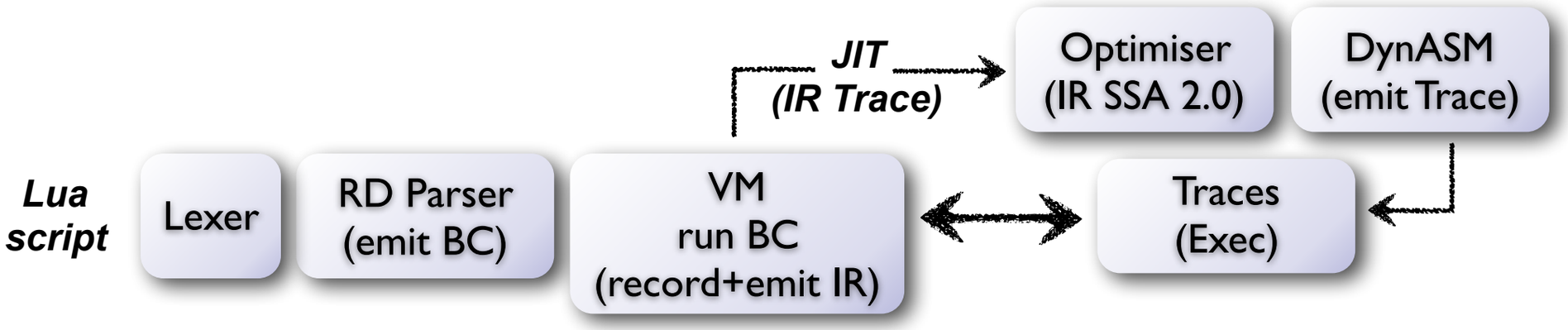
An innovative **trace compiler** is integrated with advanced, SSA-based optimisations and highly tuned code generation backends. A substantial reduction of the overhead associated with dynamic languages allows it **to break into the performance range traditionally reserved for offline, static language compilers.**

***From M. Pall website, author of LuaJIT***

# LuaJIT details

- Entirely hand-written in C + ASM + Lua (no dependencies)
  - ➡ Implements Lua 5.1 + extensions + FFI (+ CData) + DynASM.
  - ➡ NaN tagging technology (double-as-int well optimised).
  - ➡ Highly optimised memory footprint (embedded systems, game boxes, smart phones).
  - ➡ Many smart bit-level tricks (makes code difficult to follow).
  - ➡ Many context dependent states (difficult to extend safely the code).
    - Not easy to understand and extend (took me 1-2 weeks to make all MAD extensions).
  - ➡ VM is very fast, uses ASM for dispatch, records IR for type-stable traces.
    - CPS dispatch in C should give same performance and simplify IR recording.
  - ➡ FFI includes a minimal C parser (C struct definable and usable as-is in Lua).
    - CData accessed at the speed of C struct directly from Lua.
    - C functions called at the speed of C directly from Lua.

*Is Lua(JIT) more used than Python? use ≠ users!*



# Content

I. Road to MAD

II. Technologies overview for MAD

III. MAD performance overview

IV. Lua(JIT) technology

**V. MAD technology**

VI. MAD examples

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# Prototypical Language

- Prototypical = Prototype + Delegation ➠ more general than Class + Inheritance

**Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems**

**Henry Lieberman**

A traditional philosophical controversy between representing general concepts as abstract sets or classes and representing concepts as concrete prototypes is reflected in a **controversy between two mechanisms for sharing behaviour between objects** in object oriented programming languages. **Inheritance** splits the object world into classes, which encode behaviour shared among a group of instances, which represent individual members of these sets. The class/instance distinction is not needed if the alternative of using prototypes is adopted. A prototype represents the default behaviour for a concept, and new objects can re-use part of the knowledge stored in the prototype by saying how the new object differs from the prototype. The prototype approach seems to hold some advantages for representing default knowledge, and incrementally and dynamically modifying concepts. **Delegation** is the mechanism for implementing this in object oriented languages. After checking its idiosyncratic behaviour, an object can forward a message to prototypes to invoke more general knowledge. Because class objects must be created before their instances can be used, and behaviour can only be associated with classes, inheritance fixes the communication patterns between objects at instance creation time. Because any object can be used as a prototype, and any messages can be forwarded at any time, **delegation is the more flexible and general of the two techniques**.

*OOPLSA'86*

# MAD extensions *(LuaJIT patch)*

*MAD extensions (on top of LuaJIT)*

```
! single line comment (Fortran style)

! number concatenation creates an xrange
a = 0..-1e6..-1
a = range(0,-1e6,-1)     ! need xrange module

! xrange (native) cdata type + xrange module
for i,v in ipairs(2..1e6) do
  sum = sum + v - i
end
print(sum) ! print 999999

! xrange numerical stability
for _,v in ipairs(1..1e8..0.1) do a = v end
print(a) ! print 100000000

! local 'in' table
local sin, cos, tan in math

! lambda syntax (much more available)
f = \x x*x               ! square
f = \... ...             ! identity (variadic)
f = \... {...}           ! pack (variadic)
f = \... ()              ! forget (... is optional)
f = \x,y (x+y,x-y)
f = \x\y (x+y,x-y)

a = { f=\s,y s.x*y+z, x=3 }
fib = \n n<2 and n or fib(n-1)+fib(n-2)

! deferred expressions (object model)
f=\ 2*x    ! free variable (space after \ matters)
{f=\ 2*x} ! free variable and table (f is a key)
{f:=2*x}  ! alternate syntax := equiv. =\ in table
```

*Equivalent Lua syntax and semantic*

```
-- single line comment

-- number concatenation creates string "0-1e6-1"
a = 0 .. -1e6 .. -1     -- need spaces around ..
a = range(0,-1e6,-1)    -- need xrange module

-- cdata struct xrange constructor + xrange module
for i in ipairs(range(2,1e6)) do
  sum = sum + i - (i-1)
end
print(sum) -- print 999999

-- common users (bad) loop
for v = 1,1e8,0.1 do a = v end
print(a) -- print 99999999.94

-- error prone multiple assignments from table
local sin, cos, tan = math.sin, math.tan, math.cos

-- anonymous functions
f = function (x) return x*x end
f = function (...) return ... end
f = function (...) return {...} end
f = function () return end
f = function (x,y) return x+y,x-y end
f = function (x) return
      function (y) return x+y,x-y end end
a = { f=function (s,y) return s.x*y+z end, x=3 }
fib = function (n)
        return n<2 and n or fib(n-1)+fib(n-2) end
-- anonymous functions in tables
f = function () return 2*x end
{f = function () return 2*x end}
{f = function () return 2*x end}
```

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# MAD extensions *(LuaJIT patch)* cont.

*MAD extensions (on top of LuaJIT)*

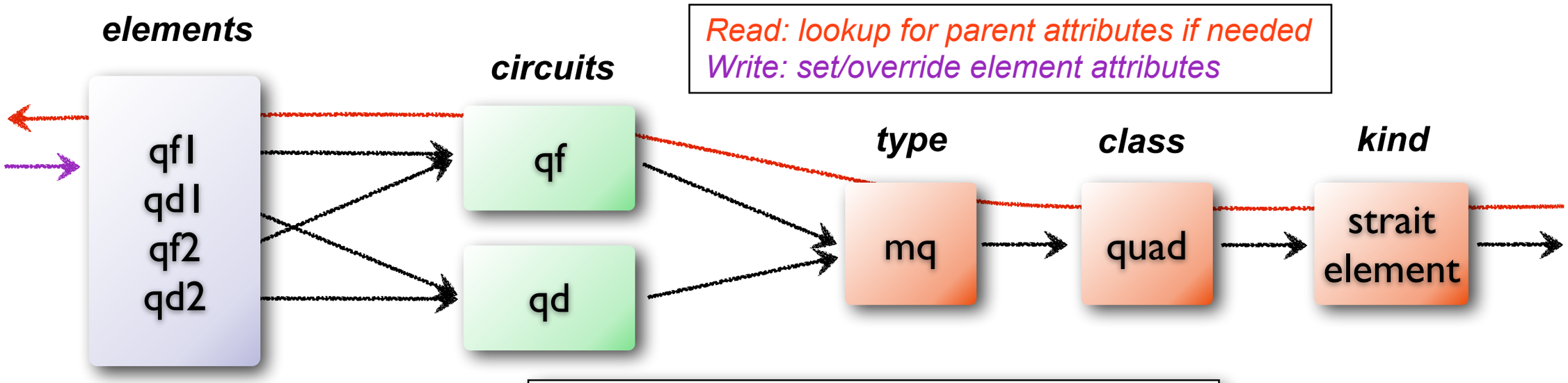
```
Accepted forms of lambda syntax
-------------------------------

! simple and compact
\par_list   (expr)
\(par_list)  expr
\(par_list) (expr)

! multiple returns
\par_list   (expr_list)
\(par_list) (expr_list)

! arrow syntax
\(par_list) -> (expr)
\(par_list) -> (expr_list)
\(par_list) => function_body

! deferred expression
{ key :=  expr }
{ key := (expr_list) }
```

```
Example of lambda use
---------------------

! shortcuts
sm, gm = setmetatable, getmetatable
id=\... ...
cat=\f,g \... f(g(...))

! bottom to build composition
_ = sm({id}, { __pow  = \f,g sm({cat(f,g[1])}, gm(g)),
               __call = \g,... g[1](...) })

! lambda functions
f=\x,y x*y
g=\x,y (x+y,x-y)
h=\x,y x^2-y^2

! composition idem cat(f,g)
fg = f^g^_

! property (a+b)(a-b) == a^2-b^2
p=\x,y fg(x,y)==h(x,y)

! check property
for i=1,1e8 do assert(p(i,i-1), "i="..i) end
! print "mad: compose.mad:11: i=94906267"

! fails for i=94906267, why?
! hint: 94906267^2 needs 53 bits for its representation
! transition from the realms of ints (exact) to floats…
```

***0.3** sec*

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# MAD object model *(principle)*




- MAD-X definition:
  - ➡ share attributes **statically** (clone)
  - ➡ no delegation (no lookup)
  - ➡ global workspace!

```
mq.l  =  2.10 ; ! type
qf.k1 =  0.05 ; ! circuit
qd.k1 = -0.06 ; ! circuit
mq: quadrupole, l  := mq.l;  ! type
qf: mq        , k1 := qf.k1; ! circuit
qd: mq        , k1 := qd.k1; ! circuit
qf1: qf;                     ! element
qf->k1 = 0.06 ; ! BAD! no update of children
value qf1->k1;  ! print 0.05
qf.k1 = 0.06;   ! update qf circuit
value qf1->k1;  ! print 0.06
```

3 variables + 4 elements

- MAD-NG definition:
  - ➡ inherit attributes using **dynamic lookup**
  - ➡ **delegation** should avoid the need of >90% of the **deferred expressions** and save many extra variables

```
mq = quad { l  =  2.10 } ! type
qf = mq   { k1 =  0.05 } ! circuit
qd = mq   { k1 = -0.06 } ! circuit
qf1 = qf {}              ! element
qf.k1 = 0.06   ! update qf circuit
print(qf1.k1)  ! print 0.06
```

only 4 elements

# MAD object model *(description)*

*MAD online help for object module*

```
NAME
  object -- creates objects

SYNOPSIS
  object = require 'object'
  obj1 = object {}                -- create a new empty object
  obj2 = object { ... }           -- create a new object with values
  obj3 = object 'name' { ... }    -- create a new object with name and values
  obj4 = obj3 ‘name' { ... }      -- create a new object from parent (class)

DESCRIPTION
  The 'object' module implements the necessary machinery to support prototype-
  based programming with value semantic for functions and further extensions:
  - On read, the lookup of values follows the inheritance down to 'Object' with
    the following precedence: immediate methods, variables, inherited methods.
    + If the retrieved value is a function, it is called with 'self' passed as
      argument (can be ignored) and the result is returned.
    + To store functions with arguments in variables, use 'set_function'.
    + To store methods or metamethods (both are inherited), use 'set_method'
      or 'set_metamethod' respectively.
  - On write, the value is simply stored (no lookup).
    + To override this behavior, just (re)defined the __newindex metamethod
      using set_metamethod with 'override' set to 'true' (use with care!).
  - On build, the new instance is connected to its parent (inheritance).
    + If the new instance has a defined __init metamethod (inherited), it will
      be called on the new instance and non-nil result or the new instance will
      be returned. This feature is particularly useful to copy immediate methods
      to new instances.
  - Root 'Object' defines the following variables:
    + 'name'  points to 'self' name unless overridden (alias)
    + '__par' points to 'self' parent unless overridden (alias)
    + '__var' points to 'self' variables unless overridden (alias)
    + '__id'  holds 'self' name if provided (variable)

RETURN VALUES
  The constructor of objects.
```

***Implemented in only 180 lines of pure Lua. But it would be an overkill without LuaJIT optimisations behind.***

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# MAD object model *(details)*

```
local Object = require 'object'      ! root of all objects
local Point = Object 'Point' {}      ! class 'Point'

local p1 = Point 'p1' { x=3, y=2, z=1  }  ! Point child
local p2 = p1    'p2' { x=2, y=1 }        ! p1 child
local p3 = p2    'p3' { x=1  }            ! p2 child
local p4 = p3    'p4' { }                 ! p3 child

print(p1.name, p1.x, p1.y, p1.z)     ! p1  3   2   1
print(p2.name, p2.x, p2.y, p2.z)     ! p2  2   1   1
print(p3.name, p3.x, p3.y, p3.z)     ! p3  1   1   1
print(p4.name, p4.x, p4.y, p4.z)     ! p4  1   1   1

local s = 0
for i=1,5e8 do
  s = s + p1.z + p2.z + p3.z + p4.z
end
print('s=',s)                        ! s=  2000000000

p1.getz = \s s.z                  ! another way to lookup

s = 0
for i=1,5e8 do ! lookup path x2
 s = s + p1.getz + p2.getz + p3.getz + p4.getz
end
print('s=',s)                        ! s=  2000000000

p1.getz = nil
p1:set_method('getz', \s s.z) ! yet another way to lookup

s = 0
for i=1,5e8 do ! lookup path x2
 s = s + p1:getz() + p2:getz() + p3:getz() + p4:getz()
end
print('s=',s)                        ! s=  2000000000
```

*0.5 sec*

*0.5 sec*

*0.5 sec*

**~1 billion lookups per sec including the 4 additions**

```
index: z 'p1'
getv : z 'p1'
index: getz 'p2'
getv : getz 'p2'
getv : getz 'p1'
getv : getz 'Point'
getv : getz 'Object'
getv : getz 'nil'
get  : getz 'p1'
index: z 'p2'
getv : z 'p2'
getv : z 'p1'
index: getz 'p3'
getv : getz 'p3'
getv : getz 'p2'
getv : getz 'p1'
getv : getz 'Point'
getv : getz 'Object'
getv : getz 'nil'
get  : getz 'p2'
get  : getz 'p1'
index: z 'p3'
getv : z 'p3'
getv : z 'p2'
getv : z 'p1'
index: getz 'p4'
getv : getz 'p4'
getv : getz 'p3'
getv : getz 'p2'
getv : getz 'p1'
getv : getz 'Point'
getv : getz 'Object'
getv : getz 'nil'
get  : getz 'p3'
get  : getz 'p2'
get  : getz 'p1'
index: z 'p4'
getv : z 'p4'
getv : z 'p3'
getv : z 'p2'
getv : z 'p1'
s    : 4
```

**trace of one iteration**

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# Deferred expressions

- Very powerful concept used everywhere in MAD-X
  e.g. injected deeply inside the physics and the inner loops…

- MAD-X definition:

```
mq.l  =  2.10 ; ! type
qf.k1 =  0.05 ; ! circuit
mq: quadrupole, l  :=  mq.l;
qf: mq        , k1 :=  qf.k1;
qd: mq        , k1 := -qf.k1*0.95; ! deferred expression
qf.k1 = 0.06;   ! update qf circuit
value qd->k1;   ! print -0.057
```

deferred expressions are implemented as lambda without argument

- MAD definition:

```
mq = quad { l  =  2.10 } ! type
qf = mq   { k1 =  0.05 } ! circuit
qd = mq   { k1 := -qf.k1*0.95 } ! deferred expression
qf.k1 = 0.06   ! update qf circuit
print(qd.k1)   ! print -0.057
```

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# Content

I. Road to MAD

II. Technologies overview for MAD

III. MAD performance overview

IV. Lua(JIT) technology

V. MAD technology

**VI. MAD examples**

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# Sequence: accelerator definition

- Simple construction of lattice using operator overloading
  - ➡ *can mix **sequence**s, sub**sequence**s, **line**s and **element**s arbitrarily (not in MAD-X)*
  - ➡ *names can be anonymous or non-unique (not in MAD-X)*

```
function make_ALS()

  ...
  BEND  = rbend 'BEND' {length=l_bend, angle=alpha, k1=-0.7787}
  BEND1 = rbend 'BEND1'{length=l_bend, angle=alpha, k1=-0.7787}
  CAV   = rfcav 'CAV'  {length=0.0, volt=-1.0, freq=500e6}

  SUP1  = L1+L2+L3+QF1+VC5+L4+L5+QD1+L6+L7+L8+VC5+
          BEND+VC5+L9+SF+L10+
          L11+QFA1+L12+SD+L13+
          L14+BEND+L15+L16+SD+L17+
          QFA2+L18+L19+SF+L20+BEND+L21+
          L22+QD2+L23+L24+QF2+L25+
          L26+VC5+L27

  SUPB  = L1+L2+L3+QF1+VC5+L4+L5+QD1+L6+L7+L8+VC5+
          BEND+VC5+L9+SF+L10+
          L11+QFA1+L12+SD+L13+
          L14+BEND+L15+L16+SD+L17+
          QFA2+L18+L19+SF+L20+BEND1+L21+
          L22+QD2+L23+L24+QF2+L25+
          L26+VC5+L27

  return sequence 'ALS' { 11*SUP1+SUPB+CAV }
end
```

```
SPS:    LINE = (6*SUPER);
SUPER:  LINE = (7*P44,INSERT,7*P44);
INSERT: LINE = (P24,2*P00,P42);
P00:    LINE = (QF,DL,QD,DL);
P24:    LINE = (QF,DM,2*B2,DS,PD);
P42:    LINE = (PF,QD,2*B2,DM,DS);
P44:    LINE = (PF,PD);
PD:     LINE = (QD,2*B2,2*B1,DS);
PF:     LINE = (QF,2*B1,2*B2,DS);
```

```
pf  = line {qf,2*b1,2*b2,ds}
pd  = line {qd,2*b2,2*b1,ds}
p24 = line {qf,dm,2*b2,ds,pd}
p42 = line {pf,qd,2*b2,dm,ds}
p00 = line {qf,dl,qd,dl}
p44 = line {pf,pd}
insert = line {p24,2*p00,p42}
super  = line {7*p44,insert,7*p44}
SPS    = sequ 'SPS' {6*super}
```

**CERN machines**
**PS, SPS, LEP: easy**
**LHC, FCC: current 2016**

# Survey: 3D geometrical tracking

- Tracking code for 3D affine maps
- Survey module **extends** the elements of the module Element
  - ➡ **E** hereafter refers to the imported module Element within module Survey

```
function E.sbend.survey (elem, m)          -- elem is the current sbend
  local angle, length in elem              -- m    is the tracked "map"
  local rho, ca, sa = length/angle, cos(angle), sin(angle)
  local R = vec{rho*(ca-1), 0, rho*sa}
  local S = mat{{ ca, 0, -sa },
                {  0, 1,   0 },
                { sa, 0,  ca }}
  local tilt = elem.tilt or 0
  if tilt ~= 0 then
    local ct, st = cos(tilt), sin(tilt)
    local T = mat{{ ct, -st, 0 },
                  { st,  ct, 0 },
                  {  0,   0, 1 }}
    R, S = T*R, T*S*T.t() -- matrix equ.
  end
  m.V = m.W * R + m.V     -- matrix equ.
  m.W = m.W * S           -- matrix equ.
  return elem.s_pos + length
end
```

*As simple and clean as in the physics guide...*

*MAD-X survey: ~600 lines*
*MAD survey: ~100 lines*

```
function E.element.survey (elem, m) -- drift
  local length in elem
  local R = vec{0, 0, length}
  m.V = m.W * R + m.V     -- matrix equ.
  return elem.s_pos + length
end
```

*Orbit correction:*
*MAD-X cororbit: ~2000 lines*
*MAD cororbit: ~10 lines*

# Track: 6D dynamical tracking

- Tracking code for 6D dynamical maps (i.e. Hamiltonian mechanics)
  - ➡ **One tracking code to "rule them all"**
    - **same code** for particles, linear maps and high order DA maps!
- Track module **extends** the elements from module Element
  - ➡ **E** hereafter refers to the imported module Element within module Track

```
function E.multipole.track (elem, m)      -- elem is the current multipole
  local  iB0,  byt = 1/m.beam.beta0       -- m    is the tracked "map"
  local  knl,  ksl = elem.knl or {}, elem.ksl or {}
  local kn0l, ks0l = knl[1] or 0, ksl[1] or 0
  local nmul = max(#knl, #ksl)
  m.by = scalar(m.by or same(m.px), knl[nmul] or 0)
  m.bx = scalar(m.bx or same(m.py), ksl[nmul] or 0)
  for j=nmul-1,1,-1 do
      byt = m.x * m.by - m.y * m.bx + (knl[j] or 0)
    m.bx  = m.y * m.by + m.x * m.bx + (ksl[j] or 0)
    m.by  = byt
  end
  m.px = m.px - m.by + kn0l
  m.py = m.py + m.bx
  if kn0l ~= 0 or ks0l ~= 0 then
    m.pz = sqrt(1 + 2*iB0*m.pt + m.pt^2)
    m.t  = m.t + (kn0l * m.x - ks0l * m.y) * (iB0 + m.pt) / m.pz
  end
  return elem.s_pos
end
```

*MAD-X multipole: ~200 lines*
*MAD multipole: ~20 lines*

```
function E.element.track (elem, m)  -- drift
  local L, iB0, T = elem.length, 1/m.beam.beta0, m.total_path
  m.l_pz = L/sqrt(1 + 2*iB0*m.pt + m.pt^2 - m.px^2 - m.py^2)
  m.x   = m.x + m.px * m.l_pz
  m.y   = m.y + m.py * m.l_pz
  m.t   = m.t + (iB0 + m.pt) * m.l_pz - (1-T)*L*iB0
  return elem.s_pos + L
end
```

Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# Conclusions

- **MAD** is a “special corner case” in scientific computing.
  - ~~Special?~~ Better to say uncommon because difficult to achieve.
    - This technology could profit to many applications.
  - Embeds LuaJIT (with few small extensions) ➠ very good performance.
  - Release 1.0 scheduled for end 2017.
- **LuaJIT + C** seems to be the best compromise for implementation.
  - Provides expected simplicity, performance, integration and portability.
  - Benchmarks show the potential of this choice.
    - Do not underestimate well designed micro-benchmarks.
    - Micro-benchmarks can guide the design of your application.
  - First demos show that “large scale” performance is x5 better than MAD-X.
    - Disclaimer: not all the physics is implemented…
- **PyPy + C**: Fallback
  - Similar technology and approach as LuaJIT (running after?). Less port. and perf.
  - Needs to do compromises with Python for an efficient JIT (by design).

# Extra slides



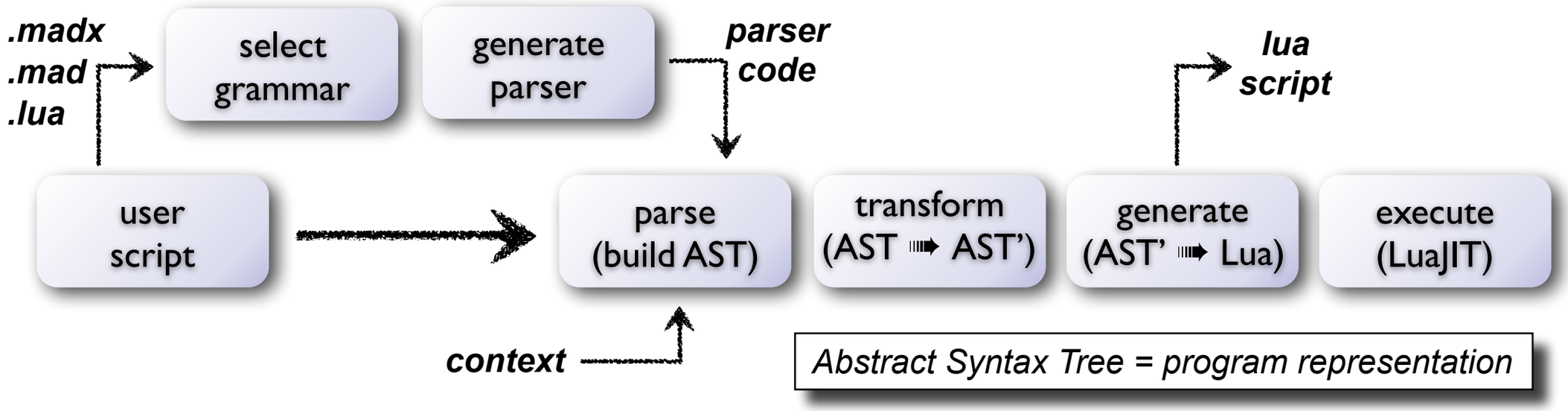
Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch

# R&D achievements

- Domain Specific Language in Lua (DSL)
  - ➡ use the expressive power of the language to embed a domain-friendly syntax
  - ➡ *Pros*: entirely written in Lua, adapted to the domain ➠ simple and intuitive to specialists
  - ➡ *Cons*: heavy machinery behind the scene, make debugging more difficult

*Is the syntax so important?*

- On-the-fly Parser Generator in Lua (advanced topic)
  - ➡ Read and extend other languages (e.g. MAD-X, MAD, Lua), eval. 100 KLOC/sec
  - ➡ Already able to parse LHC files, MAD-X dict (C), deferred. expr. (lambda), macros, ...
  - ➡ *Pros*: read new inputs, compatibility with other codes (as soon as grammars exist)
  - ➡ *Cons*: heavy machinery, difficulty to report errors location, interactive mode



# R&D achievements (cont.)

- New interpreter (fallback, aka MAD-X 6 ➠ 100% in C)
  - ➡ highly portable and extensible interpreter written in C (x10 faster than Python ➠ callback)
  - ➡ rely on tagged pointers, NaN mangling and CPS dynamic dispatch techniques (advanced)
  - ➡ slower than JIT, less expressive, bigger code ➠ **physics not in the scripting language!**
  - ➡ *Pros*: standalone, main stream tech., “full” control, lang. syntax, optimized data structures
  - ➡ *Cons*: heavy & complex code (experts!), less flexibility, less efficiency (x5-x10 slower), potential conflicts of API with garbage collectors and scripting languages (e.g. Python), only one language supported…

- Generalised Truncated Power Series Algebra (GTPSA Library)
  - ➡ improve and extend Berz TPSA
  - ➡ automatic differentiation (differential algebra)
  - ➡ real and complex (for high order normal forms)
  - ➡ support hundreds of Knobs
  - ➡ fast, parallel, 100% in C + Lua API
- Linear Algebra ($\mathbb{R}$ & $\mathbb{C}$ Matrix libraries)
  - ➡ fast, Lua + 20% in C + Lapack F77

*R&D*
*Proof of concepts*
*Helped a lot to take decisions*
*to make the design,*
*and to have fallbacks.*

*Not mandatory for MAD-NG*
*Can be included later*
*(except Matrix & GTPSA)*



Laurent Deniau, CERN BE/ABP, 1211 Geneva 23, laurent.deniau@cern.ch