Who we are

- Nick Papior (DTU) and Alberto Garcia (ICMAB)
- Core developers of SIESTA – density functional theory code (condensed matter physics)
- ESL is a *community* effort – get involved – create your community driven library

Motivation

What to choose?

- Encourage active participation in development
- Create extensions for a **very** complicated code base
- Add flexibility in the program that is also useful for expert users and developers
- Do all of the above with little to no overhead(?!)

Lua – simple scripting language

- Mostly made famous due to World of Warcraft extensions
- luatex (scripting enabled replacement for pdftex)
- Also made it into controlling PLASMA (threaded LAPACK replacement)
- window managers (awesome)
- editors (neovim)
- Torch (we'll get back to this later)
- lots of game engines...
- ... many more

Lua was born in ~1993 and created with a focus on embedding

- ✓ Performance is impressive even simple for loops!
- ✓ Interpreter with stdlibs: 269 kB
- ✓ Library (.a) 449 kB
  - ~29000 lines of C code
- ✓ Simple data-structures (tables – that's it)
- ✓ A very small standard library
- ✗ Metatable paradigm (confusing ?)
- ✗ No standardized libraries for linear algebra
- ✗ Not extensively used in the HPC community
- ✗ A very small standard library

Got everything up and running in 1 week
User experience/Tweaking takes time

Enabled scripting capabilities with very little effort

- The basic principle for script interaction is

1. Add communication points/break points in hosting program
2. Call script-function from hosting program
3. Script request data → program sends data
4. Script manipulates data
5. Script sends data → program receives data

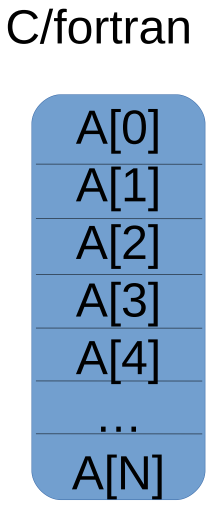
This is very much like MPI based communications, for MPI programs each processor

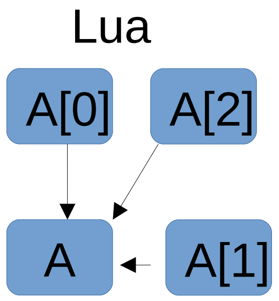
Lua tables t)

6. LuaJIT is a

1. A communication layer

2. A way of defining hosting program variables allowed for communication





Communication layer

aotus: low level fortran-Lua binding

An API for Lua interaction, can do many things such as
1. creation of Lua scripts (on the fly)
2. call fortran from Lua
3. call Lua from fortran

```
type(aot_out_type) :: dummyOut

call aot_out_open(put_conf = dummyOut, &
     filename = 'dummy.lua')

call aot_out_open_table(dummyOut, 'screen')
! Everything done in table "screen"
call aot_out_val(dummyOut, 123, 'width')
call aot_out_val(dummyOut, 456, 'height')
call aot_out_val(dummyOut, [100.0, 0.0], vname='origin')
call aot_out_close_table(dummyOut)

call aot_out_val(dummyOut, [0, 1, 2, 3], &
     vname='testarray')
call aot_out_close(dummyOut)
```

flook: high level fortran-Lua binding

Basically a wrapper around aotus for making certain things easier

```
type(luastate) :: lua
type(luatbl) :: tbl
call lua%init()
call tbl%set("screen.width", 123)
call tbl%set("screen.height", 456)
call tbl%set("screen.origin", [100., 0.])
call tbl%set("testarray", [0, 1, 2, 3])
```

```
screen = {
    width = 123,
    height = 456,
    origin = { 100.00000000, 0.00000000 }
}
testarray = { 0, 1, 2, 3 }
```

aotus: https://geb.sts.nt.uni-siegen.de/doxy/aotus/
flook: https://github.com/ElectronicStructureLibrary/flook
ESL: https://esl.cecam.org/Main_Page

Communication layer – calling fortran/Lua Lua/fortran

fortran from Lua – register

```
function f(lua_c) result(nret) bind(c)
   type(c_ptr), value : lua_c
   type(luastate) :: lua
   call lua%init(lua_c)
   ! do operations
   nret = 0 ! number of return values in stack
end function
...

subroutine register_fortran_funcs()
 call lua%register('fortran_f',func=f)
 call lua%register('send',func=send_func)
 call lua%register('receive',func=receive_func)
end subroutine
```

Lua code:

```
   fortran_f() -- call fortran function "f"
   send()
   receive()
```

Lua from fortran

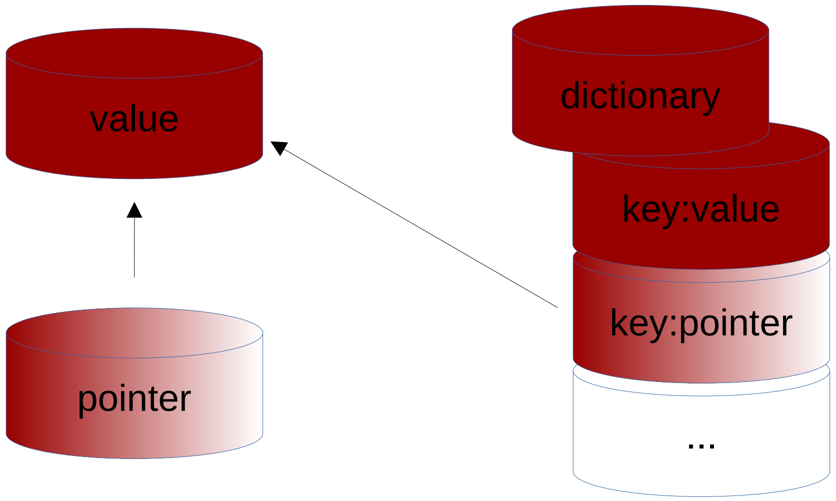
```
subroutine setup_lua()
 call lua%run(file="main.lua")
end subroutine

subroutine run_lua() `
 call lua%run(code="communication()")
end subroutine
```

“main.lua”:

```
function communication()
  fortran_f()
  receive()
  -- do stuff here
  send()
  print("Done with communication")
end
```

Exposing variables to scripting language – dictionaries

The most important thing is that the dictionary should point to the data



```
use dictionary
type(dictionary_t) :: d
integer, target :: i
integer, pointer :: ip
real, target :: r(10)

d = ('icopy'.kv.i) ! copy
d = d // ('ipointer'.kvp.i) ! pointer
d = d // ('r10'.kvp.r) ! pointer

ip => i
```

Variable dictionary

```
type(dictionary_t) :: variables
variables = &
     ('geom.na_u'.kvp.na_u)
variables = variables // &
     ('geom.cell'.kvp.ucell)
variables = variables // &
     ('geom.xa'.kvp.xyz)
variables = variables // &
     ('matrix.hamiltonian'.kvp.H)
```

Dictionary → Lua table

```
do ! loop variables
  key = .key. variables
  value = .value. variables
  call tbl%set(trim(key), value)
end do
```

Lua code

```
na_u = siesta.geom.na_u -- integer
cell = siesta.geom.cell -- table
H = siesta.matrix.hamiltonian -- table
```

Dictionary – how?

Fast hash map

- Searching 100 keys 300000 times
  ~0.04 secs
- Overload interface for UDT
- Little memory overhead
  same as pointer
- Easy to use(?)
  One needs to do a type check to retrieve the correct data, so you can’t do without a select case
- *Move away from global variables?*

fdict: https://github.com/zerothi/fdict

```
program test_point

  implicit none

  real, target :: a(10)
  real, pointer :: ap(:)

  ! Data-container, use a small memory footprint
  ! A data-container *per* data-type and dimension
  type r1_container
    real, pointer :: p(:)
  end type r1_container
  type(r1_container) :: rc

  character(len=1), allocatable :: encoding(:)
  character(len=1) :: local_enc_type(1)

  ! Create data
  a(:) = 1.
  ! Store pointer to data in container
  rc%p => a(:)

  ! Allocate encoding for pointer type
  !    casting ‘r_container‘ to character(len=1)
  allocate(encoding(size(transfer(rc, local_enc_type))))
  encoding(:) = transfer(rc, local_enc_type)
  print *, "length of encoding = ", size(encoding)

  ! To retrieve data, we back-cast
  rc = transfer(encoding, rc)
  ap => rc%p

  print *, a(2)
  ap(2) = 2
  print *, a(2)

  deallocate(encoding)

end program test_point
```

```
function siesta_comm()

   -- In this case we only request the coordinates and forces
   local get_tbl = {"geom.xa","geom.fa"}
   local ret_tbl = {}

   -- schedule communication, only receive requested data
   -- (this IS a copy!). Data will be put here:
   --    siesta.geom.xa
   --    siesta.geom.fa
   siesta.receive(get_tbl)
```

- create monitoring tools for writing out data in specific formats
- stop program at will
- change convergence parameters while running
  - particularly useful when monitoring SCF
- implement molecular dynamics modules
- Implement convergence tests in Lua (change variables → converge → change variables …)
- interactive terminal

```
out","a"))
```

```
...
call lua%run(
   "siesta.s
call lua%run(
...
do iscf = 1, N
  call lua%run(c
     "siesta.state = siesta.SCF_LOOP")
  call lua%run(code="siesta_comm()")
end do
...
call lua%run(code= &
   "siesta.state = siesta.FORCES")
call lua%run(code="siesta_comm()")
...
```

```
   if siesta.state == siesta.INIT_MD then
      siesta:print("...Right before entering the SCF loop...")
      local fh = assert(io.open("lua_out_".. siesta.Node ..".out","a"))
      -- Write each molecular dynamics steps geometry to a file
      write_mat(fh,"xa",siesta.geom.xa)
      fh:close()
      -- Call a Lua function, returning a table of send-values
      ret_tbl = init_md(siesta)
   end

   siesta.send(ret_tbl)
end
```

What have we learned?

How to control program flow?

Build up your program around the scripting environment

- everything controlled by script
- program becomes API for script code
- makes your program flexible
- requires a great deal of effort to ensure users are not doing stuff they aren't supposed to do(!)

Program *must* be built from scratch with this paradigm

Enable scripting in existing programs

- some limitations on scope
- the developers decide where interaction is performed
- rather easy for developers
- allows core developers to try out *easy* stuff before doing real thing
- difficult to check if users do stuff they are not supposed to do(!)

Scripting capabilities
gives (too) many ideas!
procrastination

What have we learned?

Stuff missing?

- Performance issues (for large matrices)
  - LuaJIT enables direct interaction with C-pointers and data arrays (almost C-speed and consecutive memory!)
    Also allows direct manipulation with data
- Small stdlib comes at a price
  No linear algebra
- MPI communication, either:
  - use Lua based MPI (exists)
  - register fortran functions for communication

  Either way, LuaJIT is required
- Users
  - documentation is important!
  - adaptation takes time

Users – was Lua the right choice?

The future will tell

- Documentation improvements
- More feedback from users
- Lua requires adaptation
- Python? Julia? Other?

Don’t forget:

End-users are the main target!

Thank you for your attention!