Lua (and Fortran) in thermomechanical simulations

Vadim Zborovskii

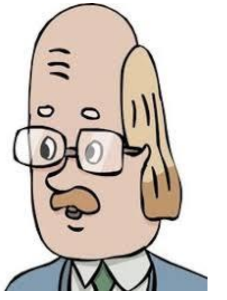
SRC RF TRINITI

Lua Workshop 2018

Lithuania, Kaunas, September 6-7, 2018

Outline

0. Introduction to the topic
1. Lua as an extension language for simulation software
    1.1. Problem definition
    1.2. Lua binding to Fortran-2003/2008
    1.3. Library to specify material properties in Lua
2. Lua as a glue language for computational modules
3. Conclusion



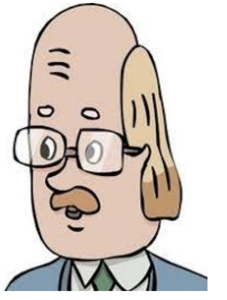
$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Software for thermomechanical simulations

- Proprietary
    - ANSYS, ABAQUS, LS-DYNA, COMSOL, …
- FLOSS
    - CalculiX, Z88, OpenFOAM, CodeAster, ElmerFEM …
- Programming languages: Fortran, C, C++ etc
- Why to write new code?
    - Simplicity for particular problems
    - More efficient and reliable numerical algorithms
    - Special physical models

Outline

0. Introduction to the topic
1. Lua as an extension language for simulation software
    1.1. Problem definition
    1.2. Lua binding to Fortran-2003/2008
    1.3. Library to specify material properties in Lua
2. Lua as a glue language for computational modules
3. Conclusion



$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Problem definition

How would the user specify material properties

in a simulation software?

- Thermophysical properties (heat conductivity, heat capacity, density)
- Mechanical properties (elastic moduli, thermal expansion coefficient, creep rate, swelling rate, …)
- Physico-chemical properties

… depending on temperature and other parameters

Ways to specify properties

Alternative 0:

Hard-code formulae for material properties

```
FUNCTION CLYOUN(TK)
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
CLYOUN = (9.6-0.06*(TK-273))*1.D9
END
```

☺ Easiest and obvious implementation

☹ User can’t change anything

😐 Certified propertied can be specified this way

More ways to specify properties

2. Compiled plug-ins

1. Parametrization

0. Hard-coding

3. Extension language

Ways to specify properties

Alternative 1:

Parametrize formulae

```
FUNCTION CLYOUN(TK)
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
INCLUDE ‘param.fi’
CLYOUN = (CLY_A+CLY_B*(TK-273))*1.D9
END
```

☺ Conceptual simplicity

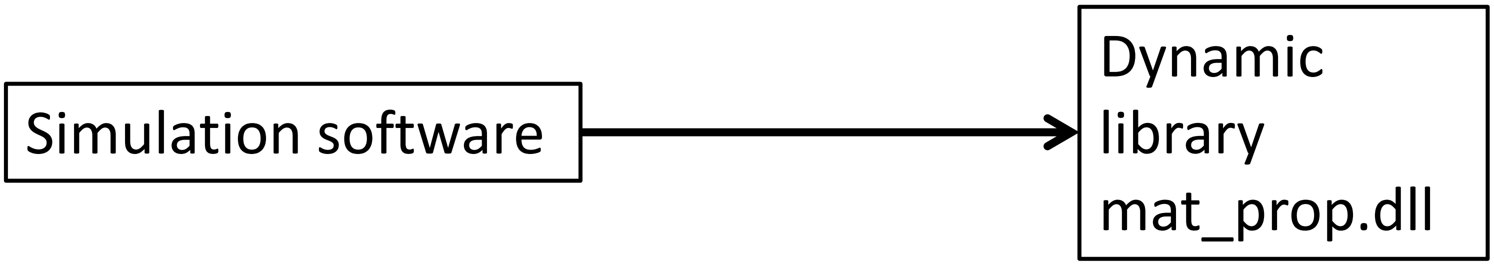
☹ Extremely tedious and error-prone

☹ User changes coefficients but not the formulae itself

Ways to specify properties

Alternative 2:

Load compiled plug-ins dynamically



☺ User can specify any relation

☺ Performance

☹ User himself has to program and compile plugins

☹ Errors if API/ABI doesn’t match

Ways to specify properties

Alternative 3:

Use extension language or DSL

```
FUNCTION CLYOUN(TK)
use ftnlf
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
LOGICAL r
r = luafunc(‘props_clad’, ‘young’, [TK], ZE)
CLYOUN = ZE
END
```

```
function Young (TK)
    local CLYOUN
    CLYOUN = (9.6-0.06*(TK-273))*1.e9
    return CLYOUN
end
```

Ways to specify properties

Alternative 3 (continued)

☺ Simple and easy way for end-user

☺ Any relation can be specified

😐 Performance?

☹ Need to bind simulation software to implementation of some DSL or extension language

Which extension language to choose?

- Do-it-yourself DSL
  ☺ No external dependencies
  ☹ Need to do it ourselves …
  ☹ … and make an substandard language

Greenspun's tenth rule of programming:

> Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.

- Existing general-purpose “scripting” language
  - Lua, Tcl, JS, Python, Scheme, …

  ☺ Easy for user! Complex algorithm can be implemented
  ☹ We need binding and glue

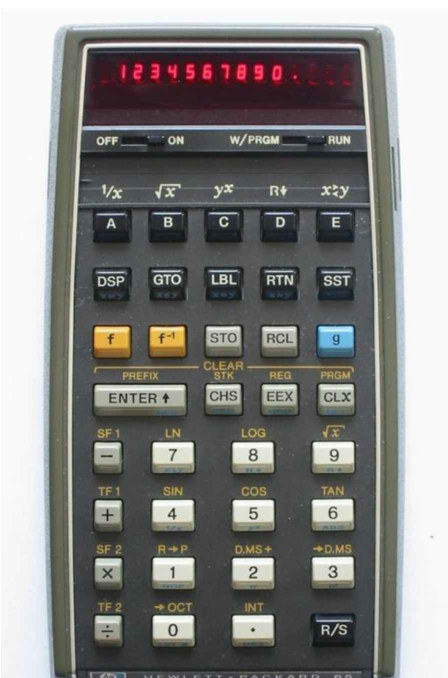
Why Lua?

- ☺ Compactness, minimalism, portability
  - Support of required platforms out-of-box
  - Easy to compile – just add a static library
  - Easy to deploy – no extra files
- ☺ Simple but full-featured language with humane syntax
- ☺ Works well for unseasoned programmers
- ☺ Convenient C API
- ☺ Performance
- ☺ Designed for engineers from the beginning

Challenges?

☺ Non-problems:

- 1-based indices
- “local” declarations: think about (let …) form
- Performance



sydneysmith.com

Lua C API features

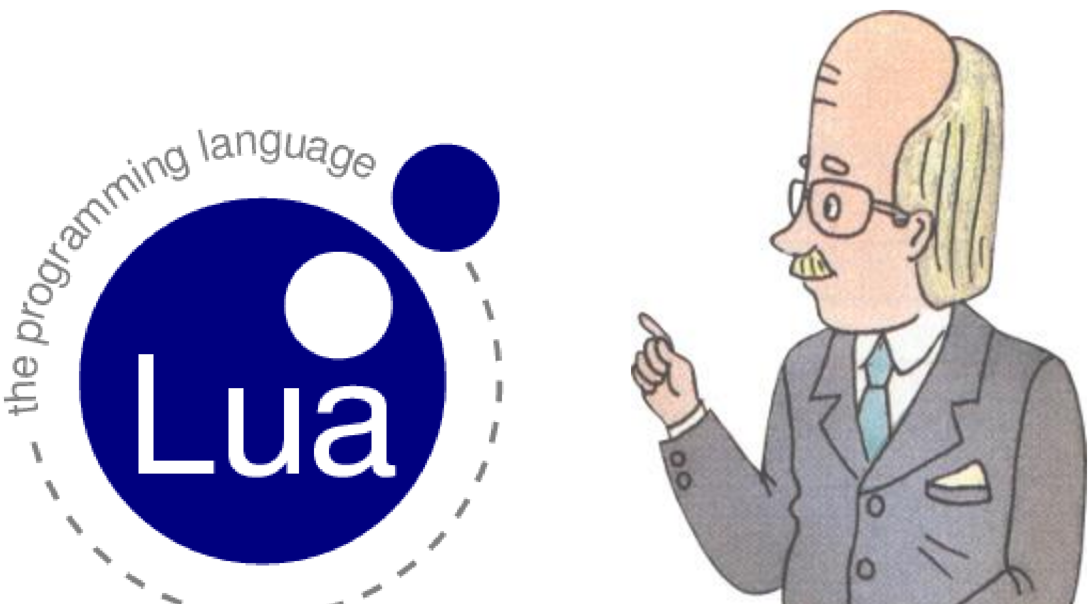
Stack manipulation is like the programming of RPN calculators

Do we need a higher-level code constructs?

ru.wikipedia.org

Work objective

1. Lua binding to modern Fortran
2. The library (module) to specify material properties in an input file



“Prof. Fortran meets Lua”

Requirements (and desires) for binding

1. Portability
2. No intermediate wrappers
3. As complete mapping of Lua C API as possible
4. Trivial installation – just add some files to a project

My implementation:

bitbucket.org/vadimz/luaf/ (MIT/X11 license)

Other/similar projects (All – MIT/X11 licensed):

bitbucket.org/haraldkl/aotus/

github.com/adolgert/FortLua/

github.com/MaikBeckmann/f2k3-lua/

Language support

- Fortran 2003/2008 + TS 29113 – «Interoperability with C»
  - C calling conventions support
  - Transparent mapping of scalars, structures and arrays
  - Opaque pointers to arbitrary data and functions
- Lua C API
  - The interface is declared in ANSI C 89
  - All work with Lua VM is through opaque pointer lua_State
  - (*Almost*) no need in address arithmetic
  - Garbage collection

Implementation details

Easy to use:

```
use luaf
```

Multi-layered API

- Lua C API mapping
- Wrappers useful for Fortran
- Domain-specific API (depends on problem)

Interface example (semi-automatically generated):

```
!extern void (lua_pushinteger) (lua_State *L, lua_Integer n)
SUBROUTINE lua_pushinteger(L1, n2) BIND(C, name="lua_pushinteger")
USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_PTR, C_INTPTR_T
IMPLICIT NONE
TYPE(C_PTR), VALUE, INTENT(IN) :: L1
INTEGER(KIND=C_INTPTR_T), VALUE, INTENT(IN) :: n2
END SUBROUTINE lua_pushinteger
```

Overcoming the difficulties (1)

- Macros
  - Implemented in Fortran
- C-strings vs Fortran-strings (arrays of known size)
  - Conversion subroutines
  - Convenience wrappers
- Variety of integral types
  - Size is specified but type cast is required sometimes
- Variadic functions (…)
  - Not implemented. Probably we don’t need them (?)

Overcoming the difficulties (2)

- NULL as “default value”
  - Variables are passed by reference in Fortran => One can neither pass nor accept NULL
  - Two versions of interface: opaque pointer and by-reference passing
- Address arithmetics in macros luaL_addchar и luaL_addsize
  - Are implemented in a partially-portable way
- Platform-dependent constants
  - LUA_IDSIZE, LUA_MINSTACK, LUA_BUFFERSIZE
  - Include file is generated automatically by a simple C program. Or reasonable defaults are used.

To do?

- Support of Lua 5.2, 5.3, 5.4, …
  - Lua 5.1 supported
- Complete and clear documentation
  - We have brief and obscure one
- More convenience wrappers (in particular, automatic conversion of C/Fortran strings)

Library to specify material properties

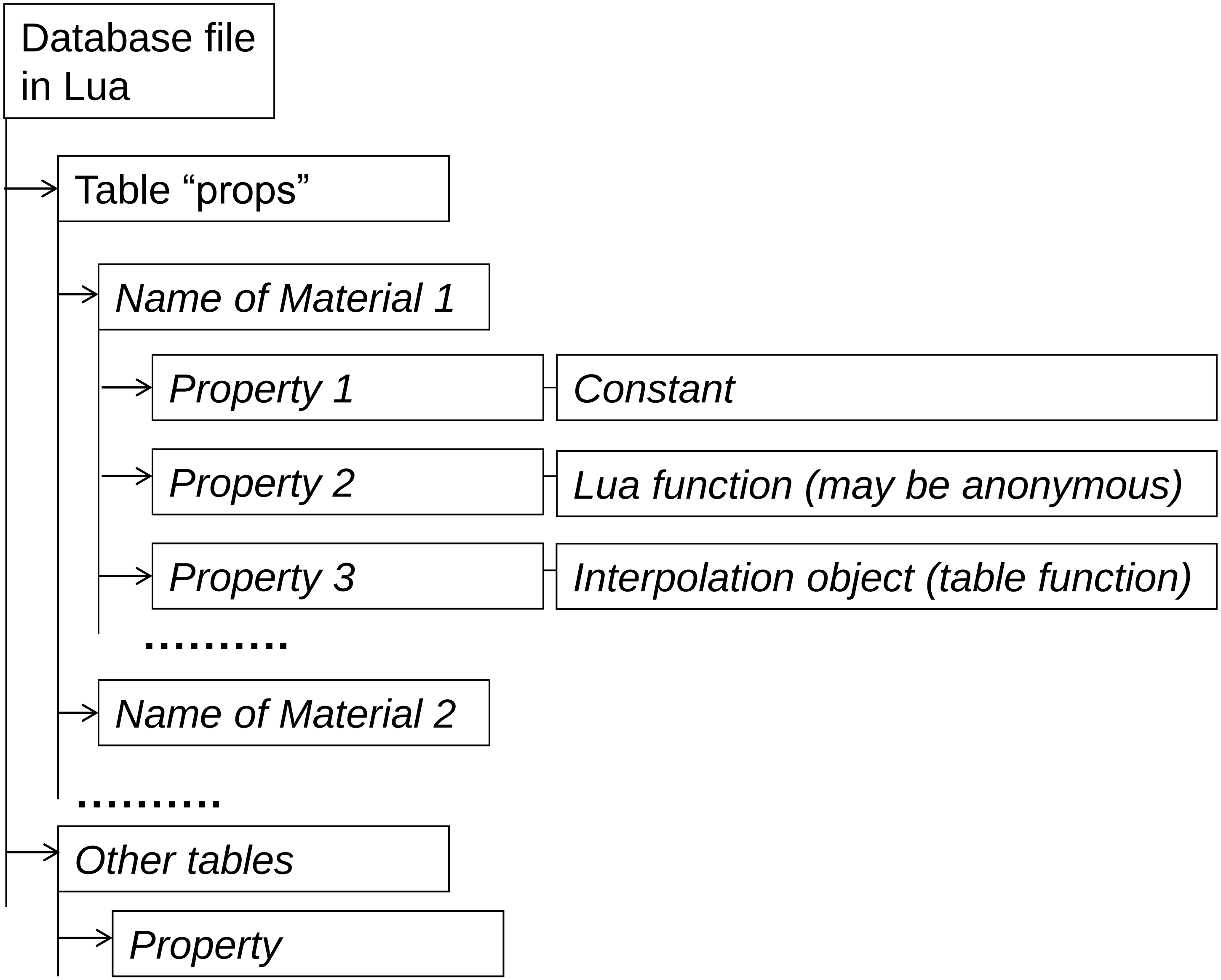
Requirements

1. Material properties are stored in “database” in Lua language
2. Properties are specified as a constant, tabular data or function
3. Various properties for different materials can be specified
4. Multiple input and output parameters are possible
5. Input and output parameters can be arrays
   - More than just properties!

My implementation: bit.ly/vz-ftnlf/ (MIT/X11 license)

Similar project: Lee Busby, “IREP and Lua”, Lua Workshop 2016
github.com/LLNL/irep/ (MIT/X11 license)

Model of database for material properties



Examples of properties specification (1)

```
local TRLE= -- linear expansion wrt 0C, %
   {273.,     0.000,
    298.,     0.025,
    300.,     0.027,
    400.,     0.125,
    500.,     0.223,
    600.,     0.322,
…………………………………………………………
    2900.,    4.021,
    3000.,    4.314,
    3100.,    4.624,
    3120.,    4.688,
   }

local RO0 = 10.e3  -- [kg/m3] density at initial temperature
for i = 1,#TRLE/2 do
   TRLE[2*i]=RO0/(1+0.03*TRLE[2*i])
end
```

Examples of properties specification (2)

```
local FXC = require('FX.Core')
props = {
   -- Properties of Material 1
mat1 = {
      -- Heat conductivity
      lambda = function(BU,TK,SW,ZTFACT,KFUDN,ZX)
………………
         local AK=100/(8.0 + 2.e-2*TK+4.e-6*TK^2
                          +(0.35-8e-5*ttc)*ZBU1)
         local AK4=AK+6400/(TK/1000)^2.5*math.exp(-EA/TK)
         return (AK4*FMM)
      end,

      -- Density as function of temperature
      dens = FXC.interp(TRLE), -- ⇦ Create interpolation object
      -- One can calculate density as dens(TK)
   },
}
```

Examples of properties specification (3)

```
local FXC = require('FX.Core')

tbl1 = {
       v2 = function(x,y) return (x/y) end,
              valN = function(x, y, ai, am)
              -- Input and modified ⇧ arrays
              local ao = FXC.array(10)
              ao[1] = (x+y)*ai[1]
              ao[2] = (x*y)/ai[2]
              ao[3] = math.atan2(y*ai[3], x*ai[4])
              am[1] = am[1]*(-10.)
              return x-y, x+y, ao -- ⇦ Output array
       end,
       pk2 = function(x,y)
              return FXC.apackt({x,y}) -- ⇦ Pack args to array
       end,
}
```

Database queries (1)

- Import of module

```
use ftnlf
```

- Initialization

```
res = ftnlf_init(filename)
```

- Finalization

```
call ftnlf_done()
```

Database queries (2)

- Evaluation of property by “two-level” address, simple version
  - Upper level table name (e.g. «props»)
    - Use `''` for one-level address
  - Lower level table name (e.g. material name)
  - Property name
  - Input parameters for calculation

```
r = luafun('props','Mat1','dens',[TK],DENS)
```

```
props = {
Mat1 = { dens = interp(TRLE), },
}
```

Database queries (3)

- Evaluation of property, advanced version
  - Scalar input parameters
  - Scalar output parameters
  - Input array
  - Modified (in-out) array,
  - Output array

```
r = luafuna('', 'tbl1', 'valN', [1.d0,-3.5d0], &
fvals(1:2), [1.d0, 2.d0, .3d0, 4.d1], arr_m, arr_o)
```

Database queries (4)

- Caching of table (two- or one- level) to make later queries easier

```
r = luacache(lvl2_name, material_name, cached_name)
```

- Queries of cached table (simple)

```
r = luafunc(cached_name, prop_name, inp, val)
```

- Queries of cached table (advanced)

```
r = luafunca(cached_name, prop_name, inp, output, &
  arr_inp, arr_m, arr_out)
```

Example of property calculation

```
real(8) FUNCTION PRMAT(TK,IERR)
……………………………………………………
      use ftnlf ⇦ ① Import of the module
……………………………………………………
      ⇩ ② DB query
      r = luafunc('props_mat1','dens',[TK],ZTDENS)
      ⇩ ③ Error handling
      if (.not. r) goto 1000
……………………………………………………
END
```

Design of the library

- Single program module ftnlf written in Fortran
- Encapsulation of Lua state (one instance)
- Almost everything runs inside protected call
- Cached tables are stored in a registry
- Special Lua module FX.Core to handle arrays and interpolation objects, written in Fortran

Lua view of the module FX.Core (1)

- Load: `local FXC = require('FX.Core')`
  - Works through package.preload
- Construction of zero-filled array: `a = FXC.array(len)`
  - len >= 0, zero-length array is cached as upvalue
  - Implemented as userdata with metatable
    - Lua takes care of memory (de)allocation
- Metamethods: __len, __index, __newindex
  - Type of userdata and metatable are checked
  - Bounds check is obligatory

Lua view of the module FX.Core (2)

Pack arguments to existing or new array

```
a = FXC.apack(arg1, …, argn, arr, index)
```

- n >= 0, ‘arr’ and ‘index’ must be present (but may be nil)
- ‘ix’ is the initial offset or 1 if ix == nil
  - ‘arr’ is the existing array, #arr >= n + ix – 1
  - Or ‘arr’ is integer, arr >= n + ix – 1
    - zero-field array of length ‘arr’ is created and arguments
  - Or arr == nil
    - array of length n + ix – 1 is created
- Arguments are packed afterwards

Lua view of the module FX.Core (3)

Pack table to existing or new array

```
a = FXC.apackt({arg1, …, argn}, arr, index)
```

Unpack elements of array

```
v1 , …, vn = FXC.aunpack(arr, ix1, ix2)
```

- ‘ix1’, ‘ix2’ may be integers or missing/nil
  - 1 <= ix1, ix2 <= #arr
  - If ‘ix1’ is nil or missing then ix1 = 1
  - If ‘ix2’ is nil or missing then ix2 = #arr
- Elements of ‘arr’ from ix1 to ix2 (both included) are returned
  - If ix1 > ix2 then nothing is returned

Lua view of the module FX.Core (4)

Construction of interpolation object: `iobj = FXC.interp(tbl)`

- like FXC.apackt with arr, ix == nil
-
```
local tbl = {
    x1,    y1,
    ..........................
    xn,    yn, }
```
- Metamethods: __len, __index, __newindex
    - Also __call: iobj(x0) => interpolate 'tbl' linearly at 'x0'

Implementation details: inside FX.Core

Copy Fortran data from/to FX.Core array

- Generic pattern to handle arrays e.g. in metamethods

```
subroutine fx_fa_copy(L, ix, kbeg, from, to)
.........................................................
       ⇩ Check arg, get length
       ud = luaL_checkudata(L, ix, F_C_STR(mt_FA))
       s = INT(lua_objlen(L, ix)/8, 4)
       ⇩ Associate
       call c_f_pointer(ud, arr, [s])
......................................................... (copy from/to)
       arr => NULL() ⇦ Disassociate
end subroutine fx_fa_copy
```

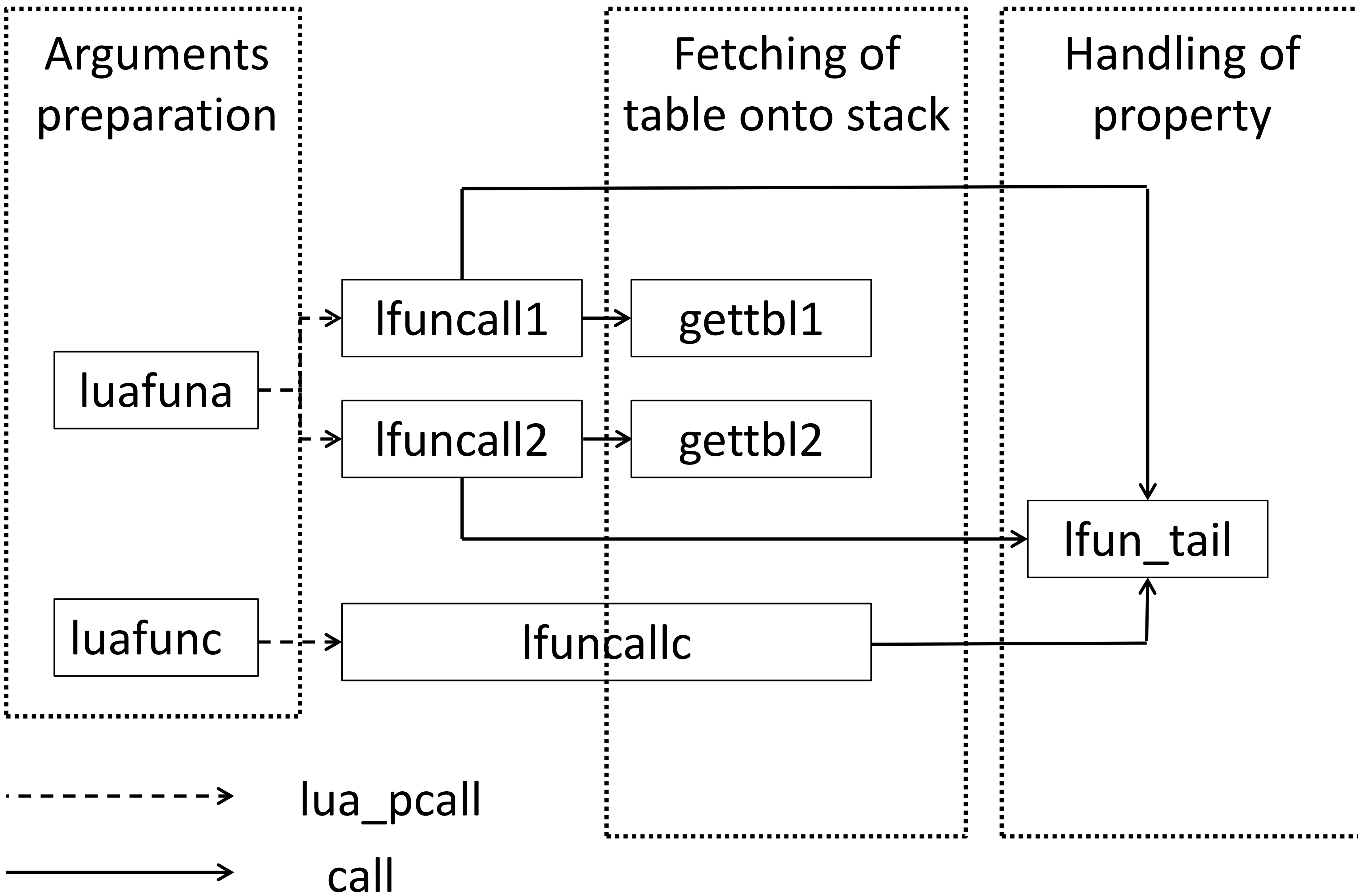
Implementation details: initialization

```
function ftnlf_init(fname) result(r)
……………………………………………………………
    L_st = luaL_newstate()
…………………………………………………………… (check for errors; prepare loaders)
    ⇩ continue initialization inside protected call
    res = lua_cpcall(L_st, c_funloc(linit1), c_null_ptr)
…………………………………………………………… (handle errors)
    ⇩ load database file
    res = luaL_dofile_r(L_st, F_C_STR(fname))
…………………………………………………………… (handle errors)
end function ftnlf_init
```

Implementation details: initialization inside cpcall

```
function linit1(L) bind(C) result(r)
……………………………………………………………
    call luaL_openlibs(L)
     ⇩ Register FX modules
    call lua_getglobal(L, F_C_STR('package'))
    call lua_getfield(L, -1, F_C_STR('preload'))
    call luaFE_registerlist(L, fx_loaders)
       list of the loaders, including FX.Core ⇧
……………………………………………………………
    r = 0
end function linit1
```

Implementation details: calculation of properties (1)



Implementation details: calculation of properties (2)

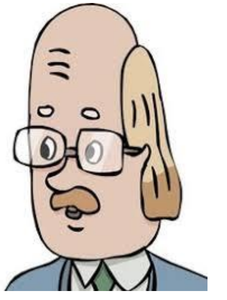
1. Arguments are loaded onto the stack.
   - Arrays are created and filled.
2. Property object is obtained from tables
3. Property is calculated
   - Constants are put onto stack
   - Functions and interpolations objects are called with lua_call
4. Results are checked
   - Output arrays are unpacked

Possible future improvements

- Documentation (is absent now)
- Systematic benchmark of performance
    - Looks like there’s no significant slowdown
- Optimization of memory-handling issues
    - Too many allocations
    - Simplicity is the goal

Outline

0. Introduction to the topic
1. Lua as an extension language for simulation software
    1.1. Problem definition
    1.2. Lua binding to Fortran-2003/2008
    1.3. Library to specify material properties in Lua
2. Lua as a glue language for computational modules
3. Conclusion



$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Definition of the problem

How to develop user-friendly and flexible software to simulate broad class of physical processes?

Example: nonlinear 2D anisotropic heat equation with some boundary conditions

$$\nabla\left(\hat{\lambda}(T,x,y)\nabla T\right)+q(x,y)=0$$

Numerical method: finite-difference approach (*support operators method*) for unstructured grid

Using Fortran and Lua to solve the problem

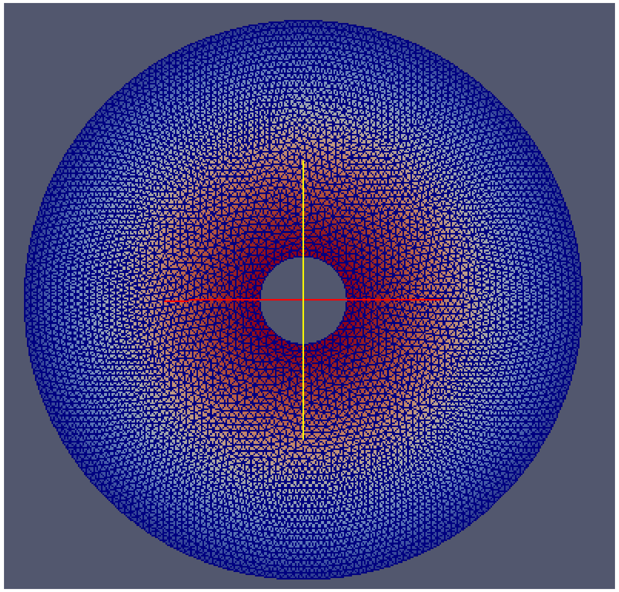
- Fortran (C):
  - Computational modules, solvers
- Lua as a glue language:
  - Reads input files including mesh data in various formats
  - Preprocesses mesh topology (if required)
  - Controls computational modules
  - Writes output files

Implementation (Proof of concept) :

https://bitbucket.org/vadimz/som/

Capabilities of the code

- Simulation for arbitrary 2D geometry with unstructured grid
- Heat conductivity, heat source and boundary conditions can be defined as relations for specific domains
- Output of data in the legacy VTK format suitable for ParaView



Input file example

```
mesh = {
   fname = "pellet.mesh",
   format = "Netgen",
}
problem = 'nonlinear'
options = {
   linear = 'LDL',
   nonlinear = 'NITSOL',
   NITSOL = {
      iplvl = 4,
      ikrysl = 1,
   },
}
out = {
  format = 'VTK',
  fname = 'pellet_nl.vtk',
  subformat = 'Poly',
}
```

```
function lam(x, y, dom, t)
   return 0.04/(1+(t-300.0)/700.0)
end

defs = {
   boundaries = {
      {bctype = 2, q = 0.0},
      {bctype = 1, T = 600},
   },
   domains = {
      {lam1 = lam, Q = 900},
   },
}
```

Some architectural details (1)

- Main program in Fortran
  - Export of solvers through package.preload
  - Load and execution of control module in Lua using luaL_loadfile и lua_pcall
- Control module in Lua
  - Input and output of files
  - Calls to solvers depending on user-defined configuration

Some architectural details (2)

- Data communication
  - Tables: input data passed from Lua to Fortran
  - Userdata or lightuserdata: results produced by solvers
- User-defined relations
  - The approach was presented as the first part of presentation

Using Lua as a glue: an example (1)

```
local FC = require('FCore') ⇦ Connection to Fortran module
⇩ Definition of nonlinear solvers
local nl_solvers = {
   NITSOL = {
      init = FC.nls_nitsol_init,
      solve = FC.nls_nitsol_solve,
      finalize = FC.nls_nitsol_finalize,
      iopts = {
         [1] = 'nnimax',
……………………………………………………………
      },
      ropts = {
……………………………………………………………
      },
   },
}
```

Using Lua as a glue: an example (2)

```
local function heat_nl_init(Conf)
   local sdata = heat_lin_init(Conf) ⇦ Initialization of linear solver
……………………………………………………………
   local nlsolv = assert(nl_solvers[opts.nonlinear],
                         'Unknown non-linear solver')
…………………………………………………………… (preparation of non-linear solver options)
   sdata.ns = nlsolv.init(sdata.hm, iopts, ropts) ⇦ Call to Fortran
   sdata.nlsolv = nlsolv
   return sdata
end

local function heat_nl_solve(sdata)
   sdata.nlsolv.solve(sdata.msh, sdata.hm, sdata.ns)⇦ Call to Fortran
end
```

Conclusion

- Fortran and Lua can work together in complex projects
- Lua is successfully used in thermomechanical simulations
  - As an extension language
  - As a glue language
- Main achievements:
  - Friendly to the end user (skilled engineer not programmer)!
  - Easy to use by the developer of simulation software

Epilogue

- The future is more than glue …

"Reinforced concrete (RC) is a composite material in which concrete's relatively low tensile strength and ductility are counteracted by the inclusion of reinforcement having higher tensile strength or ductility."

https://en.wikipedia.org/wiki/Reinforced_concrete

Thank you for attention!

vzborovsky@gmail.com