# Scientific software development and the two language problem

## Kick-off meeting: Julia CASUS

**December 5th, 2022 // Uwe Hernandez Acosta**

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

www.casus.science


HZDR HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF
TECHNISCHE UNIVERSITÄT DRESDEN
CBG Max Planck Institute of Molecular Cell Biology and Genetics
HELMHOLTZ CENTRE FOR ENVIRONMENTAL RESEARCH – UFZ
Uniwersytet Wrocławski

SPONSORED BY THE
Federal Ministry of Education and Research
STAATSMINISTERIUM FÜR WISSENSCHAFT UND KUNST
Freistaat SACHSEN

# Contents

# Who of you would say ...

*I am able to write something useful in a scripting/interpreted programming language. (something like Python, R, Octave, Matlab, Shell, Javascript, Ruby, etc.)*

# Who of you would say ...

*I am able to write something useful in a scripting/interpreted programming language.*
*(something like Python, R, Octave, Matlab, Shell, Javascript, Ruby, etc.)*

*I can write competitive programs in one these scripting languages.*
*(e.g. in Python, competitive means, you know what a meta-class is)*

# Who of you would say ...

*I am able to write something useful in a scripting/interpreted programming language.*
*(something like Python, R, Octave, Matlab, Shell, Javascript, Ruby, etc.)*

*I can write competitive programs in one these scripting languages.*
*(e.g. in Python, competitive means, you know what a meta-class is)*

*I can write something useful in a low-level programming language.*
*(e.g. C, C++, Fortran, Rust, Go, Kotlin, Java, etc.)*

# Who of you would say ...

*I am able to write something useful in a scripting/interpreted programming language.*
*(something like Python, R, Octave, Matlab, Shell, Javascript, Ruby, etc.)*

*I can write competitive programs in one these scripting languages.*
*(e.g. in Python, competitive means, you know what a meta-class is)*

*I can write something useful in a low-level programming language.*
*(e.g. C, C++, Fortran, Rust, Go, Kotlin, Java, etc.)*

*I can write competitive code in one of these low-level languages.*
*(e.g. in C++, competitive means, you can write a template class without getting mad)*

# Who of you would say ...

*I am able to write something useful in a scripting/interpreted programming language.*
*(something like Python, R, Octave, Matlab, Shell, Javascript, Ruby, etc.)*

*I can write competitive programs in one these scripting languages.*
*(e.g. in Python, competitive means, you know what a meta-class is)*

*I can write something useful in a low-level programming language.*
*(e.g. C, C++, Fortran, Rust, Go, Kotlin, Java, etc.)*

*I can write competitive code in one of these low-level languages.*
*(e.g. in C++, competitive means, you can write a template class without getting mad)*

*I can write library code in one of these low-level languages.*
*(e.g. you are one of the authors of Boost)*

# Who of you would say ...

*I am able to write something useful in a scripting/interpreted programming language.*
*(something like Python, R, Octave, Matlab, Shell, Javascript, Ruby, etc.)*

*I can write competitive programs in one these scripting languages.*
*(e.g. in Python, competitive means, you know what a meta-class is)*

*I can write something useful in a low-level programming language.*
*(e.g. C, C++, Fortran, Rust, Go, Kotlin, Java, etc.)*

*I can write competitive code in one of these low-level languages.*
*(e.g. in C++, competitive means, you can write a template class without getting mad)*

*I can write library code in one of these low-level languages.*
*(e.g. you are one of the authors of Boost)*

⇒ There are (at least) two camps for software developers in science.

# Scientific software - an attempt at a definition

*"[...] Scientific software is an application constructed within the framework of scientific research."*

*[Neumann, Espida, Daza. J. Softw. 12.2 (2017): 114-124.]*

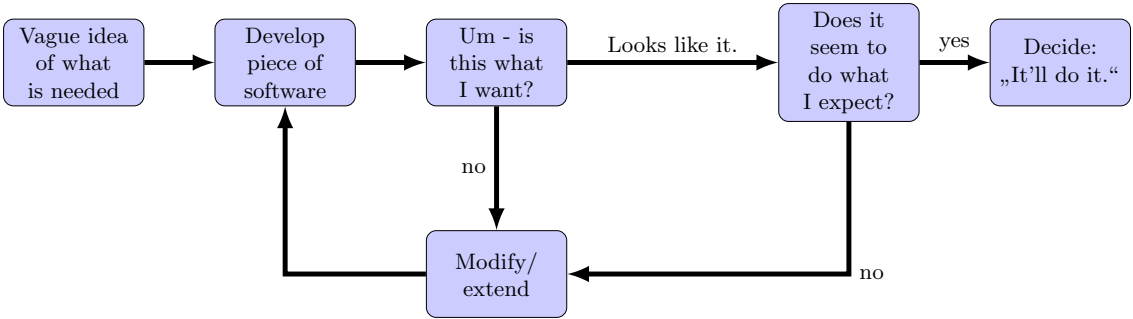
*"We define scientific software broadly as software used for scientific purposes."*

*[Kanewala, Biemann. Inf. Softw. Technol. 56.10 (2014)]*

*"Developing scientific software is fundamentally different from developing commercial software. [...] For this reason, the scientist often is the developer."*

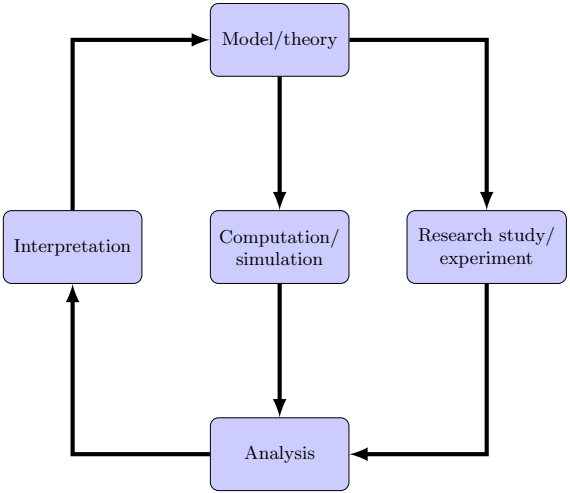
*[Segal, Morris. IEEE software 25.4 (2008)]*

# Scientific software development workflow (in general)




[Segal, Morris. IEEE software 25.4 (2008)]

# Scientific lifecycle




- Data acquisition
- Computation & HPC
- Data management
- Data analysis
- Presentation/ Visualisation

# For this reason, the scientist often is the developer.
## The good old days

**History:** The FORTRAN compiler

- John W. Backus 1957
- FORmula TRANslating System
- Invented for numerical computations and scientific computing
- Computing was done in universities and big companies
- barely for data acquisition, management, analysis, visualisation

Back then, software *was* science...

..., but only for number crunching.

# For this reason, the scientist often is the developer.
## The modern days: Ousterhout's dichotomy

*" [I propose that] you should use *two* languages for large software system: one, such as C or C++, for manipulating the complex internal data structures where performance is key and another, such as Tcl, for writing small-ish scripts that tie together the C pieces and are used for extensions."*

*[Ousterhout. "Re: Why you should not use Tcl" 1994] [Ousterhout. IEEE Computer magazine 31.3 (1998)]*

# Two types of languages

| system languages | scripting languages |
| --- | --- |
| • statically types<br>• advanced support for complex data structures<br>• compiled into machine code<br>• needs prior design | • dynamically typed<br>• little support for complex data structures<br>• scripts are interpreted<br>• interactive usage |
| C, C++, Java/Kotlin, Rust, Swift, C#, Fortran, ...<br>"fast"<br>"hard-to-use" | Python, R, Octave, Perl, Ruby, Matlab,...<br>"slow"<br>"easy-to-use" |

# But what shall we use for science?
## How scientific software is build (simplified)

1. Start prototyping in a scripting language
2. build a proof-of-concept (if possible) in that scripting language
3. rebuild performance critical parts in a systems language
4. replace parts of the p.o.c. with more performant versions (if possible)
5. write glue code to keep everything together
6. repeat 3-5 if necessary (and if possible)

# Why is this problematic?

- rewriting parts = refactoring
- different language = different logic
- need of glue-code
- extending a two-language program is a mess
- debugging a two-language program is a mess
- scientist needs to be a polyglot
- multithreading? anyone?


⇒ We have a two-language problem!

# "But I just use `numpy/scipy/pandas/... !`"
## Third-party libraries

- "use C only where needed!"
  - they use performant implementations, where necessary
- they are a valid tool
  → go with them as far as possible
- **not** capable to do something outside-the-box
  → usually not easy to extent
- science usually *is* outside-the-box
  → the goal is to build something new, isn't it?

# "But I just use `Numba/Pyhtran/PyPy/Cython/...` !"

- sufficient for small code pieces
- these *are* second languages
  - only support of a subset of the scripting language...
  - .. or add new commands/logic/concepts
- Usually **not** a full system programming language
  - e.g. `Cython` is neither `Python`, not `C/C++`
  - e.g. `Numba` is neither `Python`, nor `C/C++`
- welcome to maintenance hell
  → dependence to **one single** library

# A proposal for a solution

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

## The Julia programming language

*"We are greedy: we want more."*

*[Bezanson, Karpinski, Shah, Edelman - "Why We Created Julia" (2012)]*

# The Julia programming language

- Invented 2012 mostly at the MIT
- from Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman
- main design goals:
  - a language that's open source, with a liberal license
  - the speed of C with the dynamism of Ruby
  - obvious, familiar mathematical notation like Matlab
  - as usable for general programming as Python
  - as easy for statistics as R
  - as powerful for linear algebra as Matlab
  - as good at gluing programs together as the shell

*"Something that is dirt simple to learn, yet keeps the most serious hackers happy."*

*[Bezanson, Karpinski, Shah, Edelman - "Why We Created Julia" (2012)]*

*"Julia: come for the syntax, stay for the speed."*

*[Perkel. Nature 572.7768 (2019): 141-143]*

# Come for the syntax ...
## Julia's language features

- easy-to-write
  - garbage collector
  - dynamically typed
  - extensive standard library (written mostly in Julia)
  - transparent compilation
- easy-to-use
  - syntax similar to Python/Matlab
  - just-in-time compiled
  - generic programming
  - powerful type inference
- modern dev-environment
  - build-in packaging system
  - build-in package manager
  - build-in testing framework

# Come for the syntax ...
## Advanced language features

- Multiple dispatch
  - high order of expressiveness
  - allows arbitrary efficient code
  - easy to extend and combine code
- meta-programming support
  - Julia source code is a datatype in Julia
  - fairly easy to generate/analyse Julia code with Julia
- Multithreading
  - supports OpenMP-like models
  - supports hybrid threading
  - supports task-migration

# Stay for the speed
## Standard micro-benchmarks

# Scalability
## Julia is member of the peta-flop club

- The Celeste project (2017) at Cori in Berkley
  - → Analysis of 178 TB of data...
  - → ... estimates parameters of $1.88 \times 10^8$ stars...
  - → ... in **14.6 minutes!**

⇒ **1.54 pflop/s** ($1.3 \times 10^6$ threads over 9300 nodes)

# Summary

- scientific software has a two-language problem
  - scripting languages are slow in execution
  - system languages are hard-to-use
  - gluing things together is a mess
- The problem is not solved by
  - third-party libraries
  - language extensions and alike
- Julia might be a solution
  - as easy as Python
  - as fast as C