Is Julia ready to be adopted by HEP?

26th International Conference on Computing in High Energy & Nuclear Physics (CHEP2023)

Tamas Gal – Erlangen Centre for Astroparticle Physics

https://indico.jlab.org/event/459/contributions/11521/

Philippe Gras (IRFU, CEA, Université Paris-Saclay, Gif-sur-Yvette, France), **Pere Mato** (CERN, Switzerland), **Jerry Ling** (Harvard University), **Oliver Schulz** (TU Dormunt, Germany), **Uwe Hernandez Acosta** (CASUS, Görlitz, Germany), **Graeme A Stewart** (CERN, Switzerland)

My first encounter with the HEP software world

as a graduate student and research assistant in 2012

My first encounter with the HEP software world

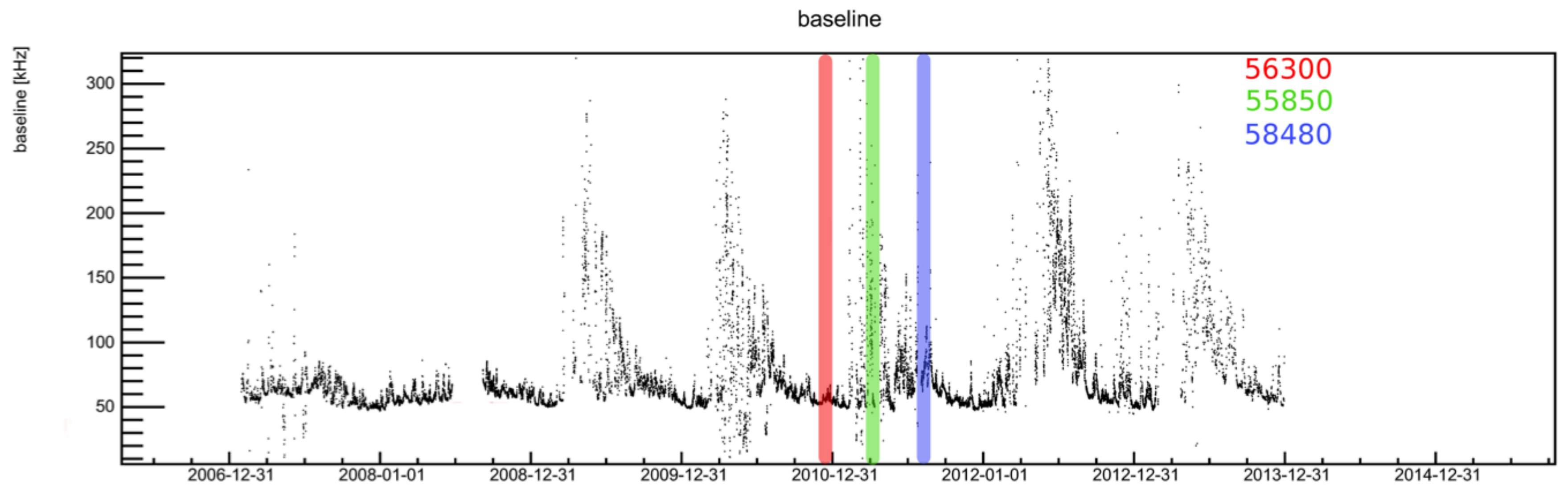
as a graduate student and research assistant in 2012

- Analysing and visualising bioluminescence data recorded by the **ANTARES neutrino detector**

My first encounter with the HEP software world

as a graduate student and research assistant in 2012

- Analysing and visualising bioluminescence data recorded by the **ANTARES neutrino detector**

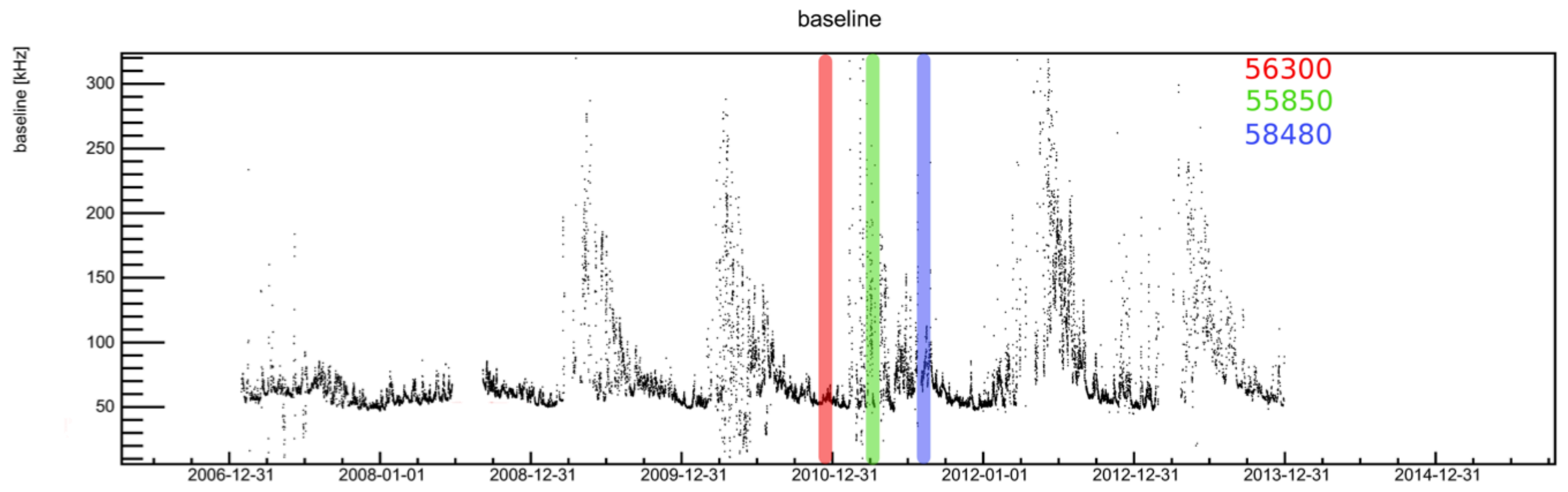


Source: "Untersuchung von Biolumineszenz im ANTARES Neutrinoteleskop", Maximilian Schandri

My first encounter with the HEP software world

as a graduate student and research assistant in 2012

- Analysing and visualising bioluminescence data recorded by the **ANTARES neutrino detector**
- Using a **ROOT**-based framework (which was btw. a nightmare to install on my MacBook running Mac OS X 10.6)

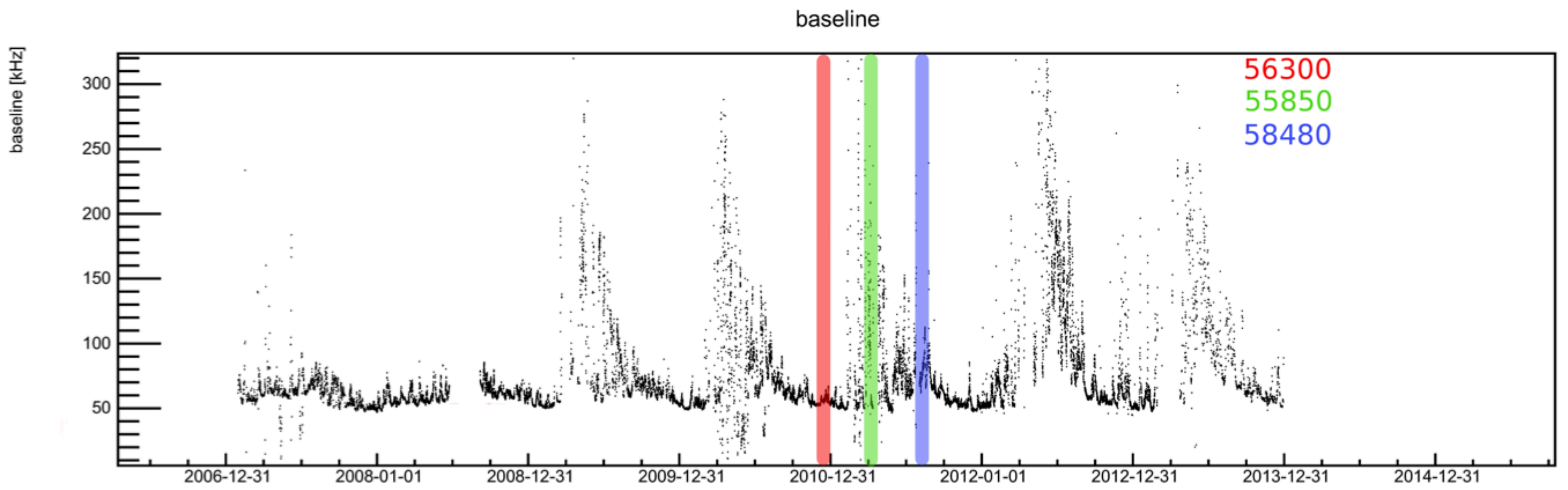


Source: "Untersuchung von Biolumineszenz im ANTARES Neutrinoteleskop", Maximilian Schandri

My first encounter with the HEP software world

as a graduate student and research assistant in 2012

- Analysing and visualising bioluminescence data recorded by the **ANTARES neutrino detector**
- Using a **ROOT**-based framework (which was btw. a nightmare to install on my MacBook running Mac OS X 10.6)
- **Why ROOT?** Because people who established ANTARES were familiar with ROOT and **humans crave convenience**

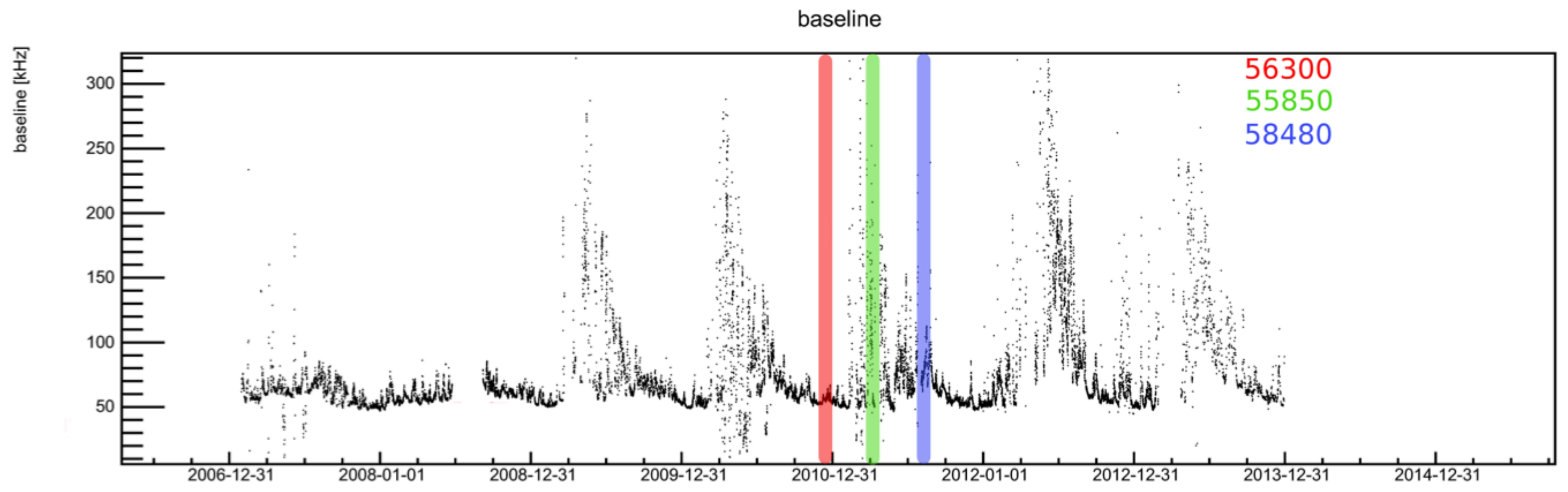


Source: "Untersuchung von Biolumineszenz im ANTARES Neutrinoteleskop", Maximilian Schandri

My first encounter with the HEP software world

as a graduate student and research assistant in 2012

- Analysing and visualising bioluminescence data recorded by the **ANTARES neutrino detector**
- Using a **ROOT**-based framework (which was btw. a nightmare to install on my MacBook running Mac OS X 10.6)
- **Why ROOT?** Because people who established ANTARES were familiar with ROOT and **humans crave convenience**



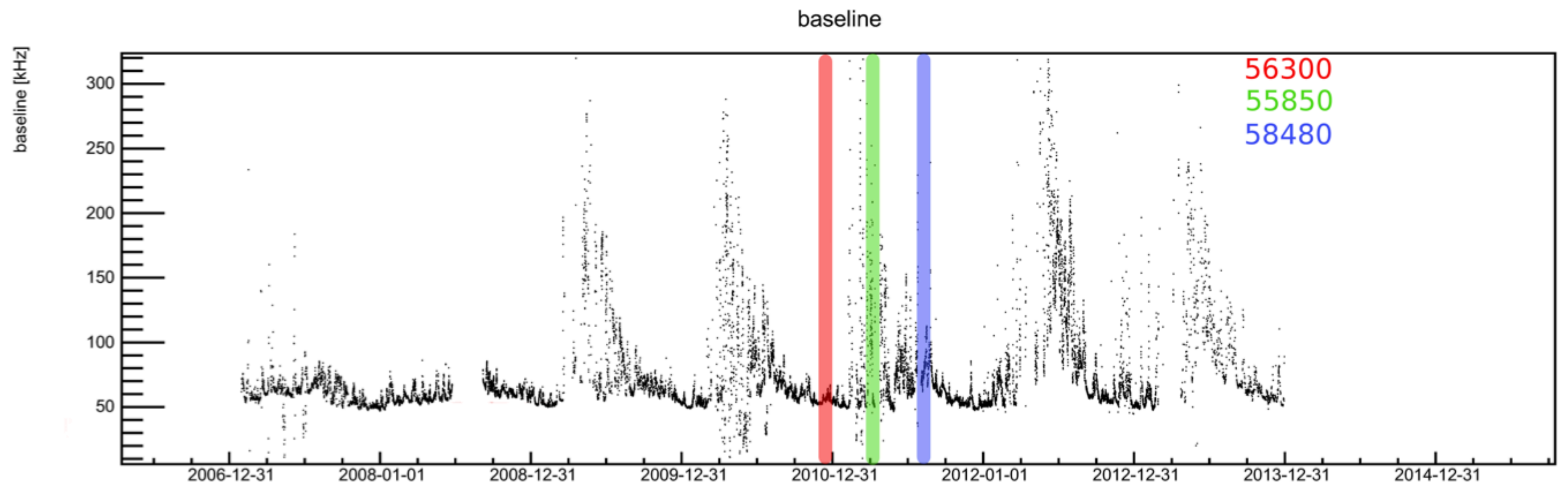
Source: "Untersuchung von Biolumineszenz im ANTARES Neutrinoteleskop", Maximilian Schandri

- **Even with** more than 15 years of (self-taught) coding **experience** in different programming languages: **it was a real challenge**

My first encounter with the HEP software world

as a graduate student and research assistant in 2012

- Analysing and visualising bioluminescence data recorded by the **ANTARES neutrino detector**
- Using a **ROOT**-based framework (which was btw. a nightmare to install on my MacBook running Mac OS X 10.6)
- **Why ROOT?** Because people who established ANTARES were familiar with ROOT and **humans crave convenience**



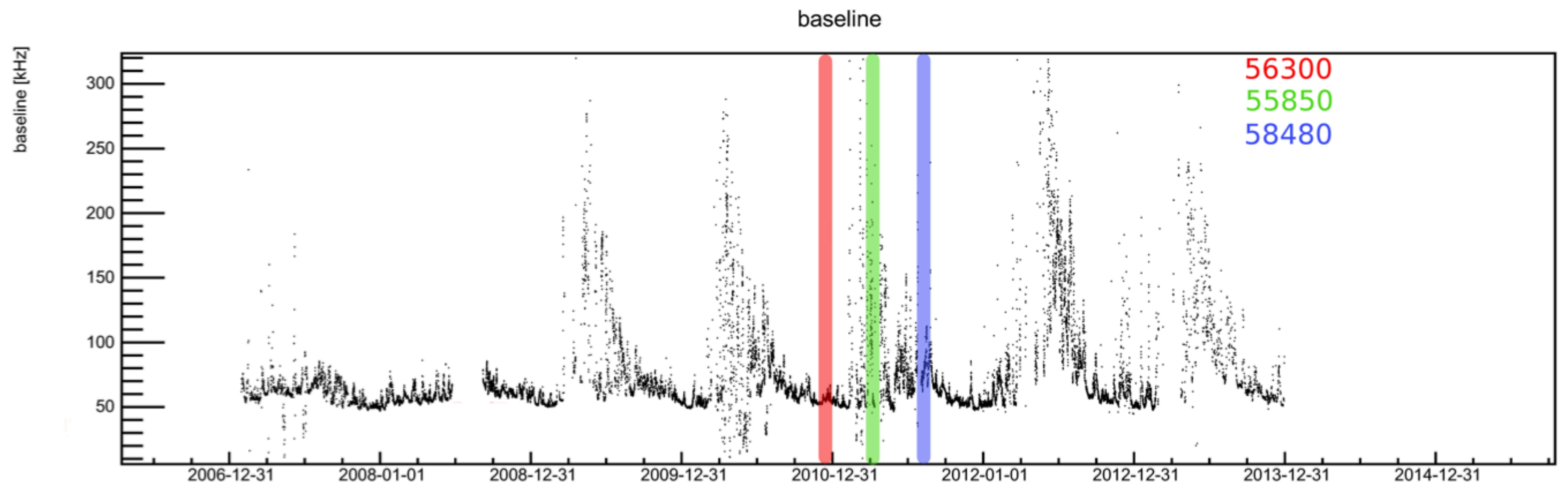
Source: "Untersuchung von Biolumineszenz im ANTARES Neutrinoteleskop", Maximilian Schandri

- **Even with** more than 15 years of (self-taught) coding **experience** in different programming languages: **it was a real challenge**
- **Lot of work** spent until the first results were presentable (kind of embarrassing how long it took to create some simple scatter plots)

My first encounter with the HEP software world

as a graduate student and research assistant in 2012

- Analysing and visualising bioluminescence data recorded by the **ANTARES neutrino detector**
- Using a **ROOT**-based framework (which was btw. a nightmare to install on my MacBook running Mac OS X 10.6)
- **Why ROOT?** Because people who established ANTARES were familiar with ROOT and **humans crave convenience**



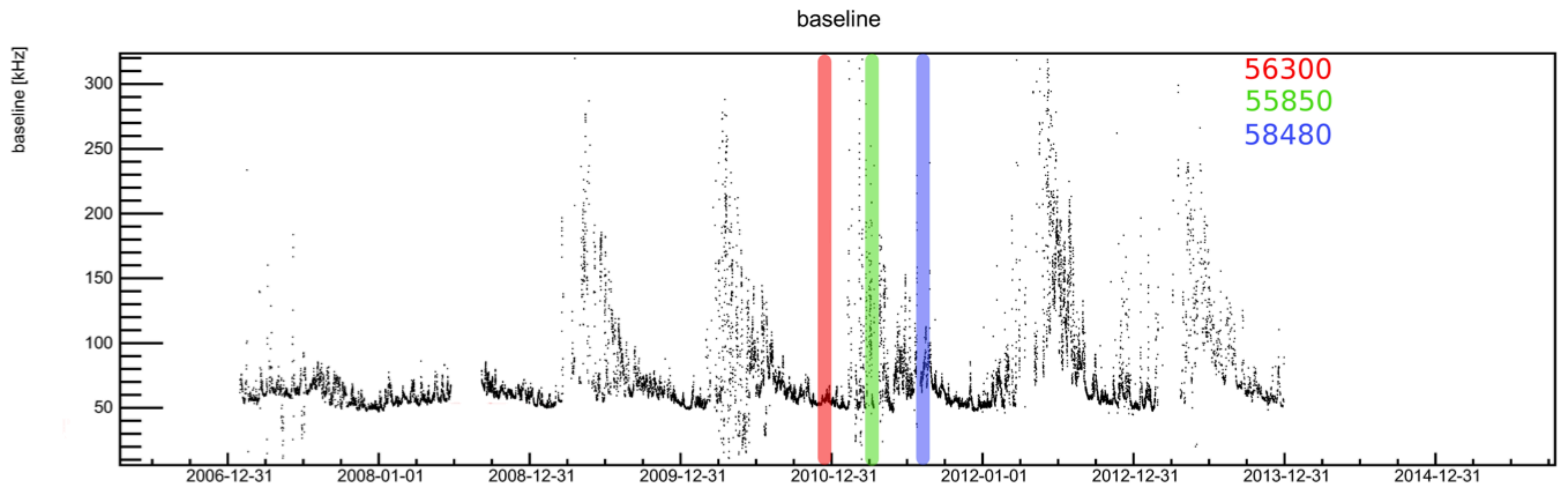
Source: "Untersuchung von Biolumineszenz im ANTARES Neutrinoteleskop", Maximilian Schandri

- **Even with** more than 15 years of (self-taught) coding **experience** in different programming languages: **it was a real challenge**
- **Lot of work** spent until the first results were presentable (kind of embarrassing how long it took to create some simple scatter plots)
- **Most of** my fellow **students** had a much worse starting situation, having almost **no coding experience** at all

My first encounter with the HEP software world

as a graduate student and research assistant in 2012

- Analysing and visualising bioluminescence data recorded by the **ANTARES neutrino detector**
- Using a **ROOT**-based framework (which was btw. a nightmare to install on my MacBook running Mac OS X 10.6)
- **Why ROOT?** Because people who established ANTARES were familiar with ROOT and **humans crave convenience**



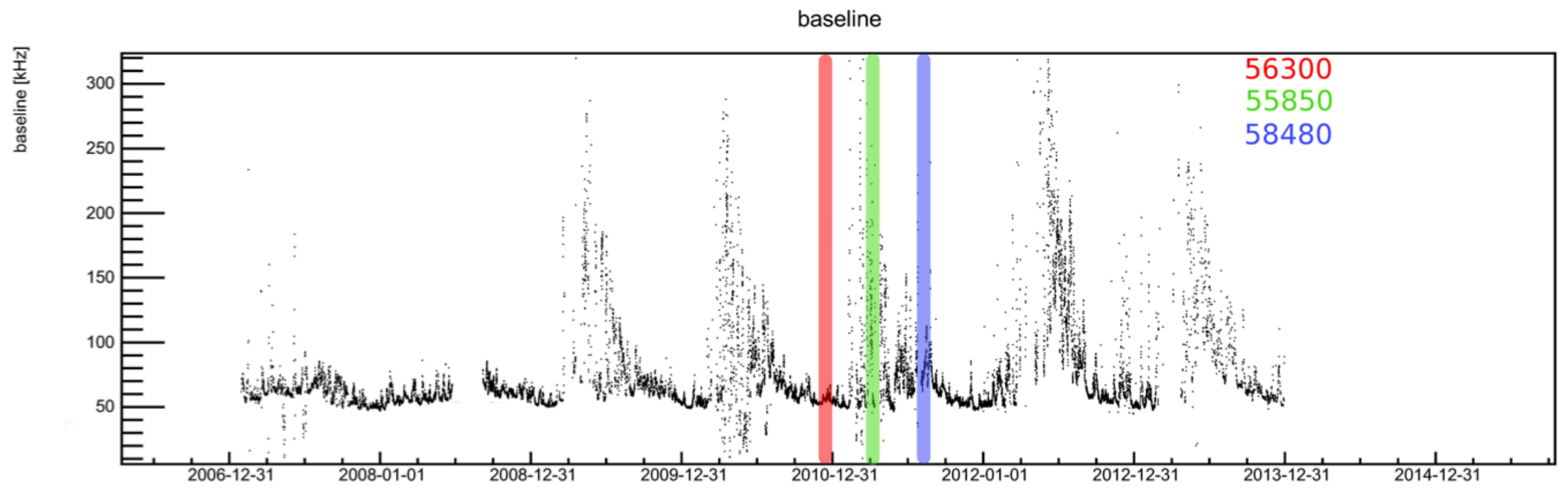
Source: "Untersuchung von Biolumineszenz im ANTARES Neutrinoteleskop", Maximilian Schandri

- **Even with** more than 15 years of (self-taught) coding **experience** in different programming languages: **it was a real challenge**
- **Lot of work** spent until the first results were presentable (kind of embarrassing how long it took to create some simple scatter plots)
- **Most of** my fellow **students** had a much worse starting situation, having almost **no coding experience** at all
- **Python started to gain some momentum in science**; I was already using it for a decade as a shell scripting replacement.

My first encounter with the HEP software world

as a graduate student and research assistant in 2012

- Analysing and visualising bioluminescence data recorded by the **ANTARES neutrino detector**
- Using a **ROOT**-based framework (which was btw. a nightmare to install on my MacBook running Mac OS X 10.6)
- **Why ROOT?** Because people who established ANTARES were familiar with ROOT and **humans crave convenience**



Source: "Untersuchung von Biolumineszenz im ANTARES Neutrinoteleskop", Maximilian Schandri

- **Even with** more than 15 years of (self-taught) coding **experience** in different programming languages: **it was a real challenge**
- **Lot of work** spent until the first results were presentable (kind of embarrassing how long it took to create some simple scatter plots)
- **Most of** my fellow **students** had a much worse starting situation, having almost **no coding experience** at all
- **Python started to gain some momentum in science**; I was already using it for a decade as a shell scripting replacement.
- Decided to work on (high-level) **Python tools** to **reduce boilerplates**, make things **more accessible** and exploit the benefits of **interactiveness** to lower the entry barrier especially for new-comers

The years after...

aka the "The Era of Python"

The years after...
aka the "The Era of Python"

- I joined **KM3NeT** (the **ANTARES** neutrino detector's successor) and **push**ed hard **for Python**

The years after...

aka the "The Era of Python"

- I joined **KM3NeT** (the **ANTARES** neutrino detector's successor) and **push**ed hard **for Python**
- **Lot's of library code and packages** written to do both **low-level** calculations (e.g. real-time detector time calibrations using K40 coincidences) and **high-level** analysis ("big-data", machine-learning, HDF5, ...)

The years after...

aka the "The Era of Python"

- I joined **KM3NeT** (the **ANTARES** neutrino detector's successor) and **push**ed hard **for Python**
- **Lot's of library code and packages** written to do both **low-level** calculations (e.g. real-time detector time calibrations using K40 coincidences) and **high-level** analysis ("big-data", machine-learning, HDF5, ...)
- **Convinced** many people that **Python is able to compete** with "compiled rivals" (mainly **C++/ROOT**) by using the **right tools to overcome its weak spots** regarding performance (GIL, duck typing, extremely slow loops...)

The years after...

aka the "The Era of Python"

- I joined **KM3NeT** (the **ANTARES** neutrino detector's successor) and **push**ed hard **for Python**
- **Lot's of library code and packages** written to do both **low-level** calculations (e.g. real-time detector time calibrations using K40 coincidences) and **high-level** analysis ("big-data", machine-learning, HDF5, ...)
- **Convinced** many people that **Python is able to compete** with "compiled rivals" (mainly **C++/ROOT**) by using the **right tools to overcome its weak spots** regarding performance (GIL, duck typing, extremely slow loops...)
- **Virtual environments** and the **Python packaging system** allowed to increase the **reusability** of code and **reproducibility** of analyses

The years after...

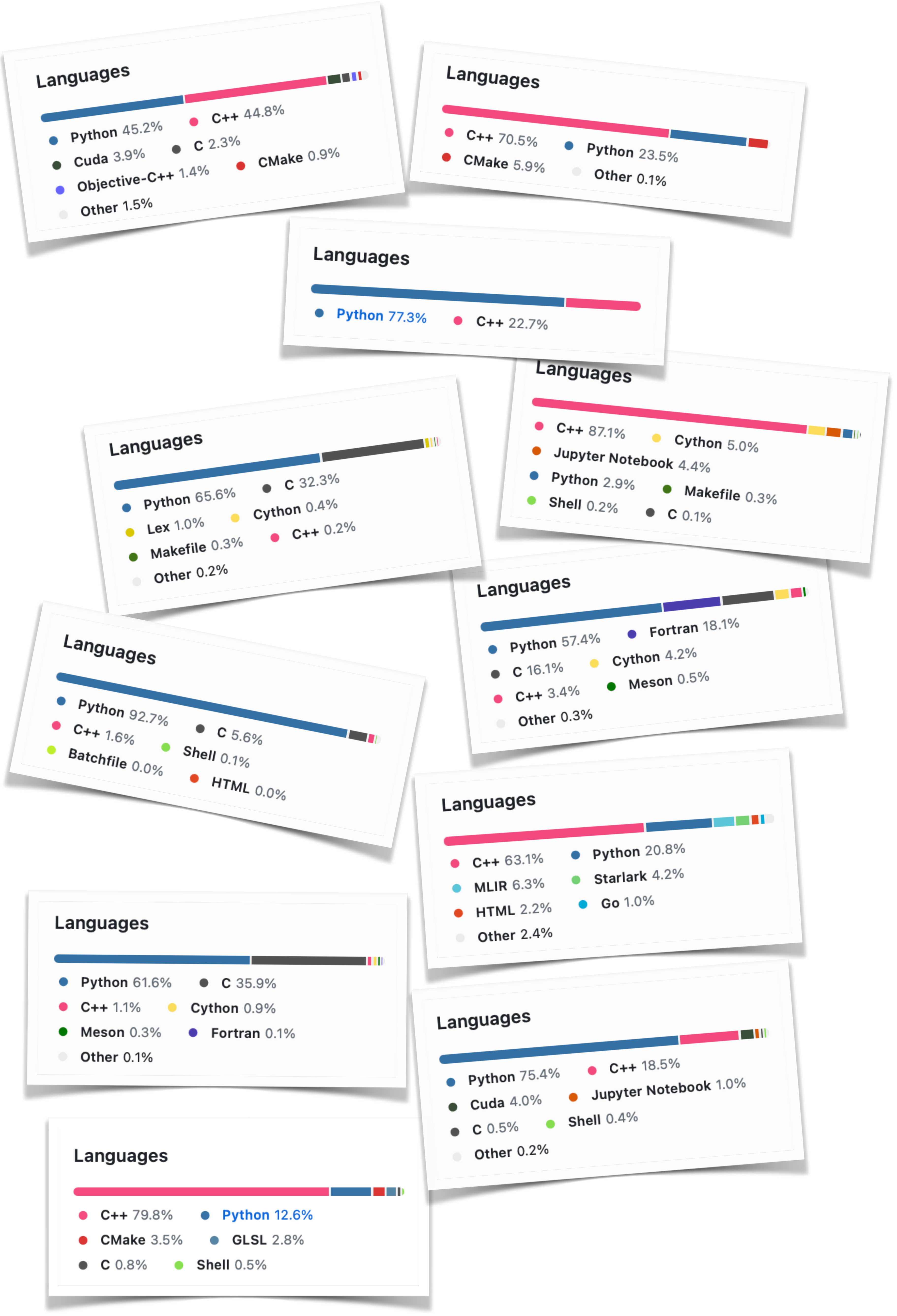
aka the "The Era of Python"

- I joined **KM3NeT** (the **ANTARES** neutrino detector's successor) and **push**ed hard **for Python**
- **Lot's of library code and packages** written to do both **low-level** calculations (e.g. real-time detector time calibrations using K40 coincidences) and **high-level** analysis ("big-data", machine-learning, HDF5, ...)
- **Convinced** many people that **Python is able to compete** with "compiled rivals" (mainly **C++/ROOT**) by using the **right tools to overcome its weak spots** regarding performance (GIL, duck typing, extremely slow loops...)
- **Virtual environments** and the **Python packaging system** allowed to increase the **reusability** of code and **reproducibility** of analyses
- Still, we ended up in a **technological Mikado**

The years after...

aka the "The Era of Python"

- I joined **KM3NeT** (the **ANTARES** neutrino detector's successor) and **push**ed hard **for Python**
- **Lot's of library code and packages** written to do both **low-level** calculations (e.g. real-time detector time calibrations using K40 coincidences) and **high-level** analysis ("big-data", machine-learning, HDF5, ...)
- **Convinced** many people that **Python is able to compete** with "compiled rivals" (mainly **C++/ROOT**) by using the **right tools to overcome its weak spots** regarding performance (GIL, duck typing, extremely slow loops...)
- **Virtual environments** and the **Python packaging system** allowed to increase the **reusability** of code and **reproducibility** of analyses
- Still, we ended up in a **technological Mikado**



The Reality

The "two-language problem"

The Reality

The "two-language problem"

- Crafting **high-performant** code in the "**Python**" programming language is **demanding**

The Reality

The "two-language problem"

- Crafting **high-performant** code in the "**Python**" programming language is **demanding**
- It requires a **profound understanding** of

The Reality

The "two-language problem"

- Crafting **high-performant** code in the "**Python**" programming language is **demanding**
- It requires a **profound understanding** of
  - **computer architecture**

The Reality

The "two-language problem"

- Crafting **high-performant** code in the "**Python**" programming language is **demanding**
- It requires a **profound understanding** of
  - **computer architecture**
  - languages **interdependencies**

The Reality

The "two-language problem"

- Crafting **high-performant** code in the "**Python**" programming language is **demanding**
- It requires a **profound understanding** of
  - **computer architecture**
  - languages **interdependencies**
  - the art of **producing reusable code** libraries

The Reality
The "two-language problem"

- Crafting **high-performant** code in the "**Python**" programming language is **demanding**
- It requires a **profound understanding** of
  - **computer architecture**
  - languages **interdependencies**
  - the art of **producing reusable code** libraries
- **Many** "solution **attempts**" exists to tackle the "**two-language problem**"

The Reality

The "two-language problem"

- Crafting **high-performant** code in the "**Python**" programming language is **demanding**
- It requires a **profound understanding** of
  - **computer architecture**
  - languages **interdependencies**
  - the art of **producing reusable code** libraries
- **Many** "solution **attempts**" exists to tackle the "**two-language problem**"
- The **maintenance overhead** rapidly escalates with each additional technology, which are mandatory

The Reality

The "two-language problem"

- Crafting **high-performant** code in the "**Python**" programming language is **demanding**
- It requires a **profound understanding** of
  - **computer architecture**
  - languages **interdependencies**
  - the art of **producing reusable code** libraries
- **Many** "solution **attempts**" exists to tackle the "**two-language problem**"
- The **maintenance overhead** rapidly escalates with each additional technology, which are mandatory
- **Python** is often merely **utilised as** the **high-level layer**, **restricting access to low-level** modifications

The Reality

The "two-language problem"

- Crafting **high-performant** code in the "**Python**" programming language is **demanding**
- It requires a **profound understanding** of
  - **computer architecture**
  - languages **interdependencies**
  - the art of **producing reusable code** libraries
- **Many** "solution **attempts**" exists to tackle the "**two-language problem**"
- The **maintenance overhead** rapidly escalates with each additional technology, which are mandatory
- **Python** is often merely **utilised as** the **high-level layer**, **restricting access to low-level** modifications
- **Loops in Python are a disaster** (as we all know), yet they remain a familiar paradigm for many programmers

The Reality

The "two-language problem"

- Crafting **high-performant** code in the "**Python**" programming language is **demanding**
- It requires a **profound understanding** of
  - **computer architecture**
  - languages **interdependencies**
  - the art of **producing reusable code** libraries
- **Many** "solution **attempts**" exists to tackle the "**two-language problem**"
- The **maintenance overhead** rapidly escalates with each additional technology, which are mandatory
- **Python** is often merely **utilised as** the **high-level layer**, **restricting access to low-level** modifications
- **Loops in Python are a disaster** (as we all know), yet they remain a familiar paradigm for many programmers
- The solutions require to **make lots of compromises**

The Reality

The "two-language problem"

We need stuff like this to be able to enjoy Python's strengths...

- Crafting **high-performant** code in the "**Python**" programming language is **demanding**
- It requires a **profound understanding** of
  - **computer architecture**
  - languages **interdependencies**
  - the art of **producing reusable code** libraries
- **Many** "solution **attempts**" exists to tackle the "**two-language problem**"
- The **maintenance overhead** rapidly escalates with each additional technology, which are mandatory
- **Python** is often merely **utilised as** the **high-level layer**, **restricting access to low-level** modifications
- **Loops in Python are a disaster** (as we all know), yet they remain a familiar paradigm for many programmers
- The solutions require to **make lots of compromises**

The Reality

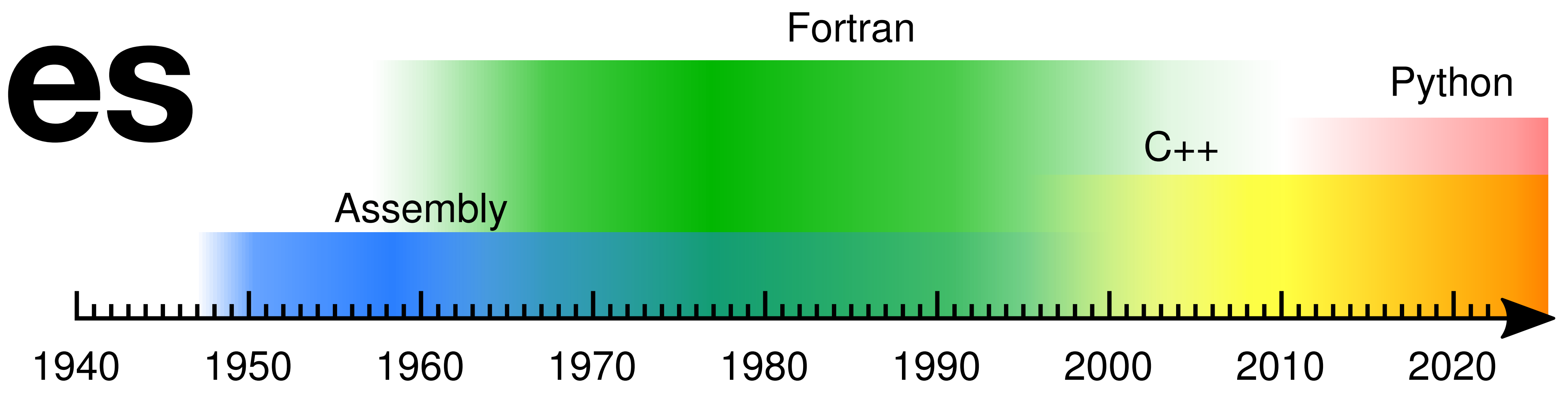
The "two-language problem"

- Crafting **high-performant** code in the "**Python**" programming language is **demanding**
- It requires a **profound understanding** of
    - **computer architecture**
    - languages **interdependencies**
    - the art of **producing reusable code** libraries
- **Many** "solution **attempts**" exists to tackle the "**two-language problem**"
- The **maintenance overhead** rapidly escalates with each additional technology, which are mandatory
- **Python** is often merely **utilised as** the **high-level layer**, **restricting access to low-level** modifications
- **Loops in Python are a disaster** (as we all know), yet they remain a familiar paradigm for many programmers
- The solutions require to **make lots of compromises**

We need stuff like this to be able to enjoy Python's strengths...

Reasons to switch languages

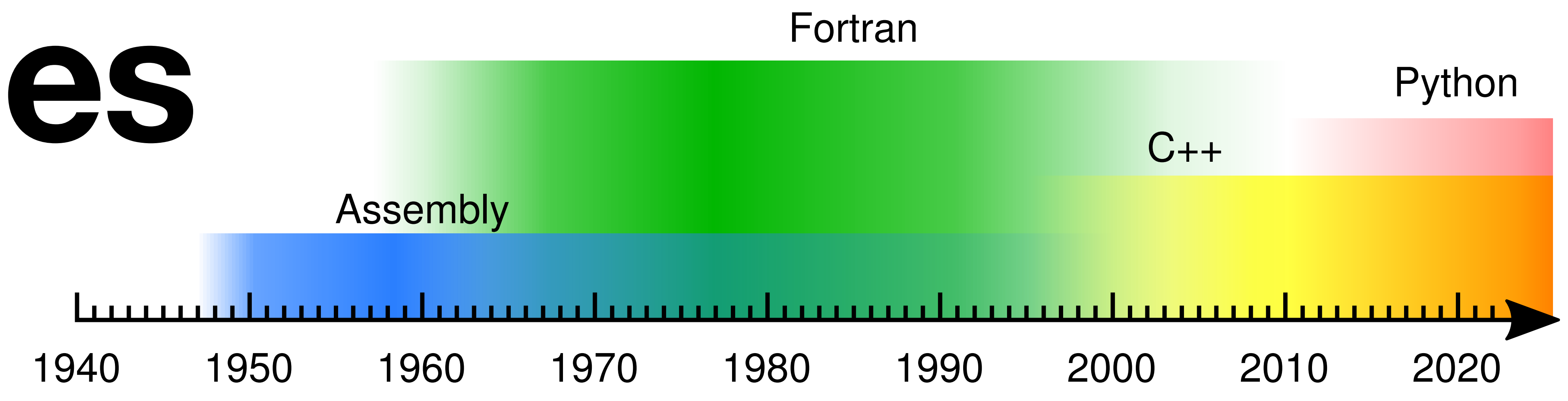
A simplified storyline in HEP



Taken from "Jagged, ragged, awkward arrays" by Jim Pivarski
(Strange Loop Conference 2019)

Reasons to switch languages

A simplified storyline in HEP

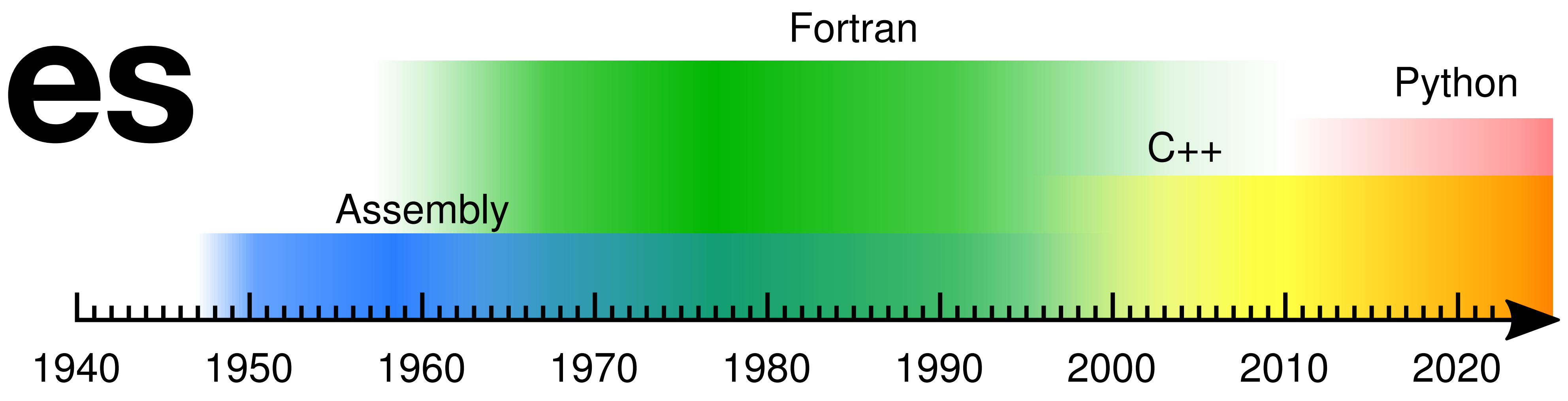


Taken from "Jagged, ragged, awkward arrays" by Jim Pivarski
(Strange Loop Conference 2019)

ASSEMBLY

Reasons to switch languages

A simplified storyline in HEP

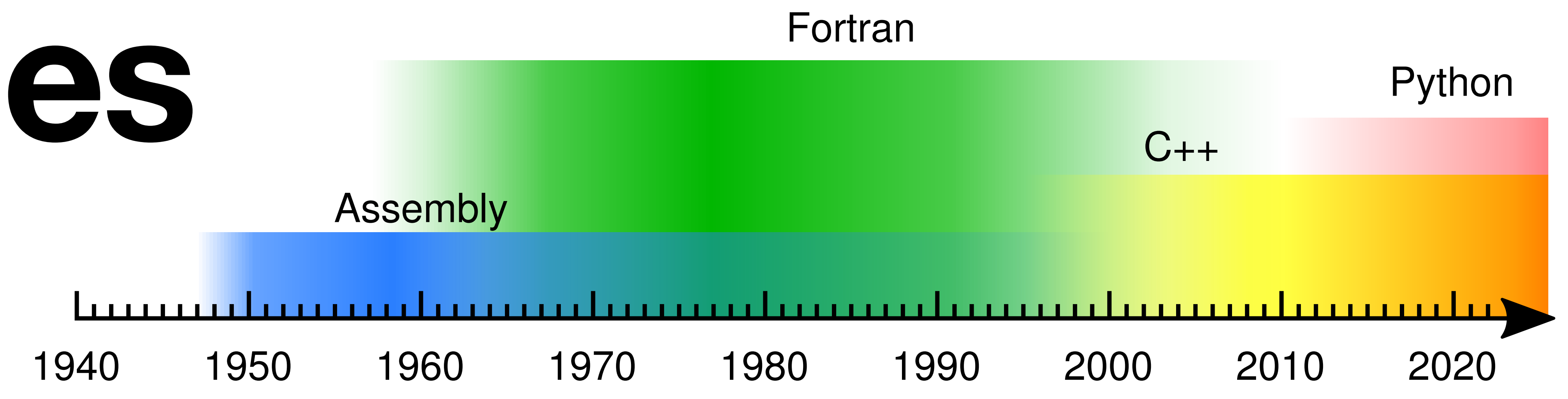


Taken from "Jagged, ragged, awkward arrays" by Jim Pivarski
(Strange Loop Conference 2019)

ASSEMBLY

Reasons to switch languages

A simplified storyline in HEP



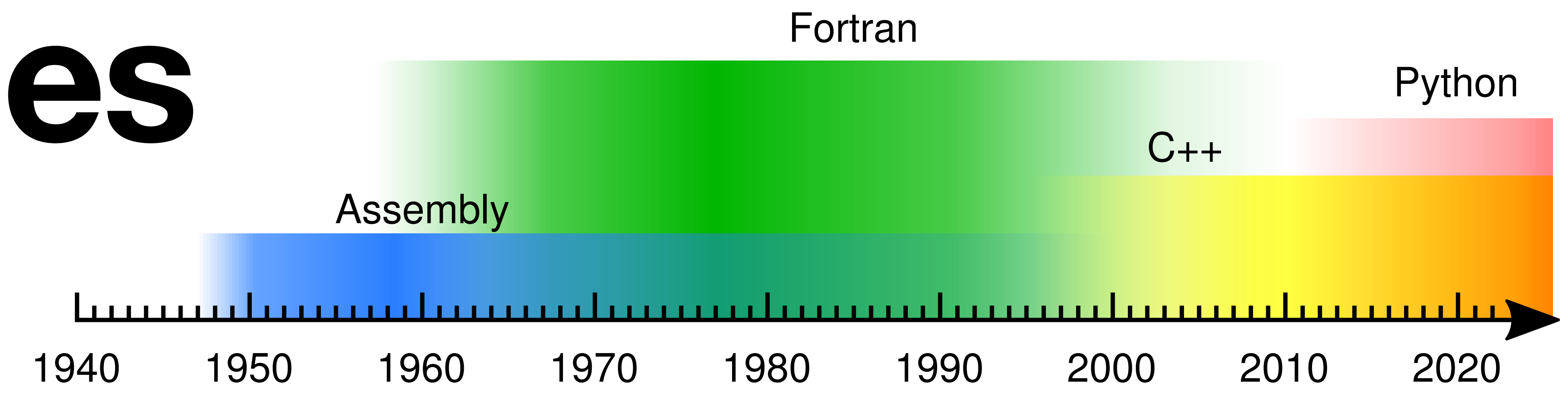
Taken from "Jagged, ragged, awkward arrays" by Jim Pivarski
(Strange Loop Conference 2019)

ASSEMBLY

Readability and
hardware independence.

Reasons to switch languages

A simplified storyline in HEP



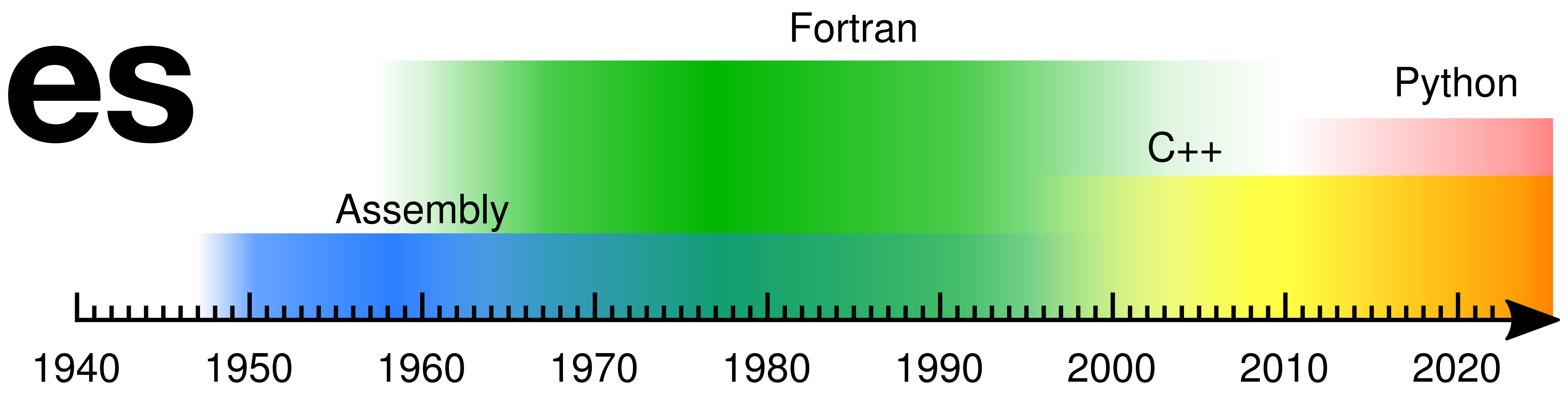
Taken from "Jagged, ragged, awkward arrays" by Jim Pivarski
(Strange Loop Conference 2019)

ASSEMBLY

Readability and
hardware independence.

Reasons to switch languages

A simplified storyline in HEP



Taken from "Jagged, ragged, awkward arrays" by Jim Pivarski
(Strange Loop Conference 2019)

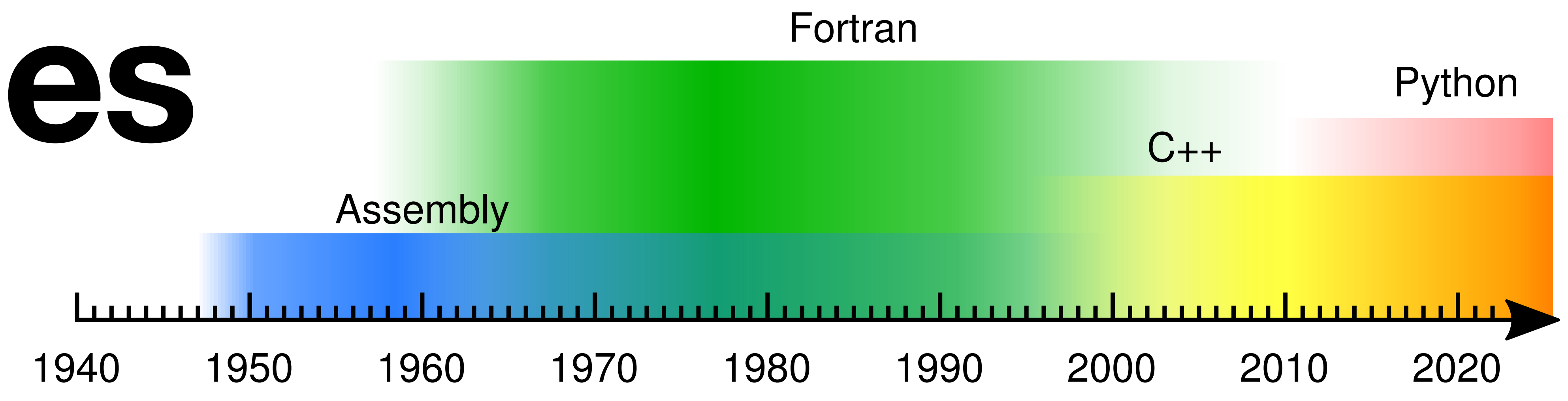
ASSEMBLY

Readability and
hardware independence.

Complex and nested
data structures

Reasons to switch languages

A simplified storyline in HEP



Taken from "Jagged, ragged, awkward arrays" by Jim Pivarski
(Strange Loop Conference 2019)

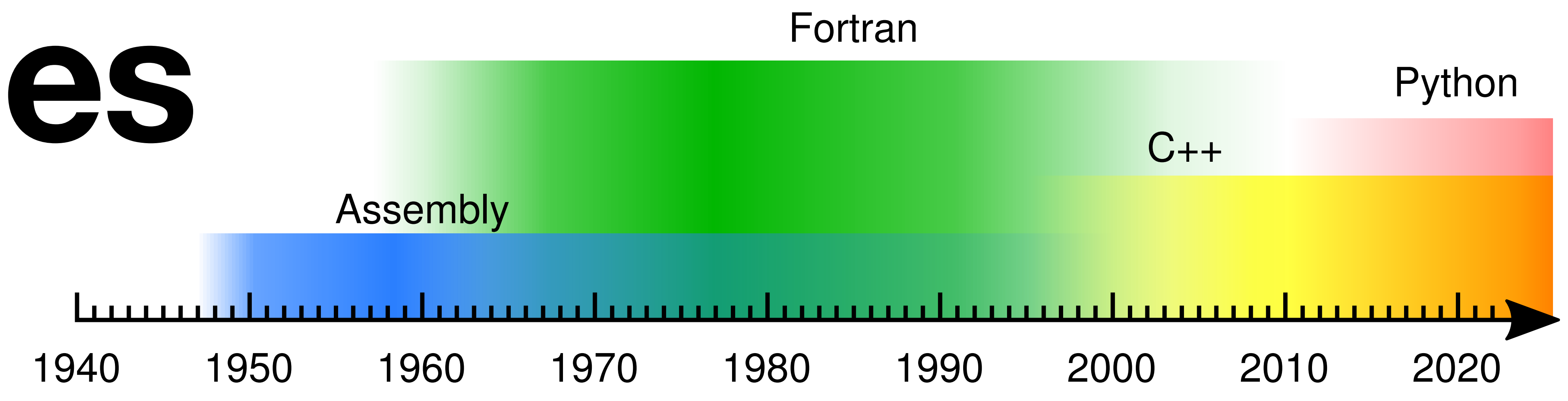
ASSEMBLY

Readability and
hardware independence.

Complex and nested
data structures

Reasons to switch languages

A simplified storyline in HEP



Taken from "Jagged, ragged, awkward arrays" by Jim Pivarski
(Strange Loop Conference 2019)

ASSEMBLY

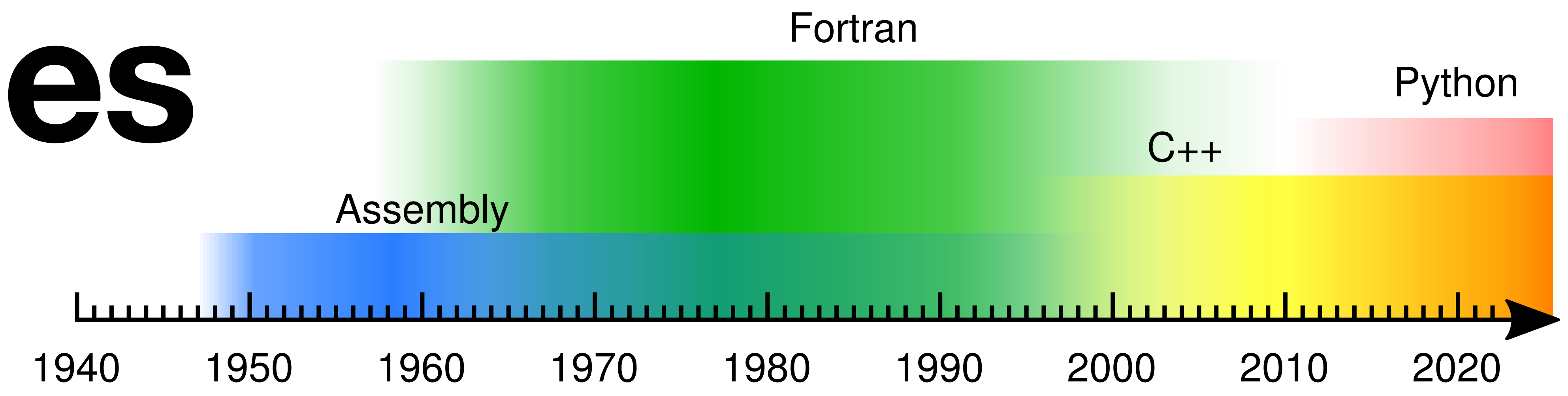
Readability and hardware independence.

Complex and nested data structures

Interactivity, ease of use, packaging

Reasons to switch languages

A simplified storyline in HEP



Taken from "Jagged, ragged, awkward arrays" by Jim Pivarski (Strange Loop Conference 2019)

ASSEMBLY

Readability and hardware independence.

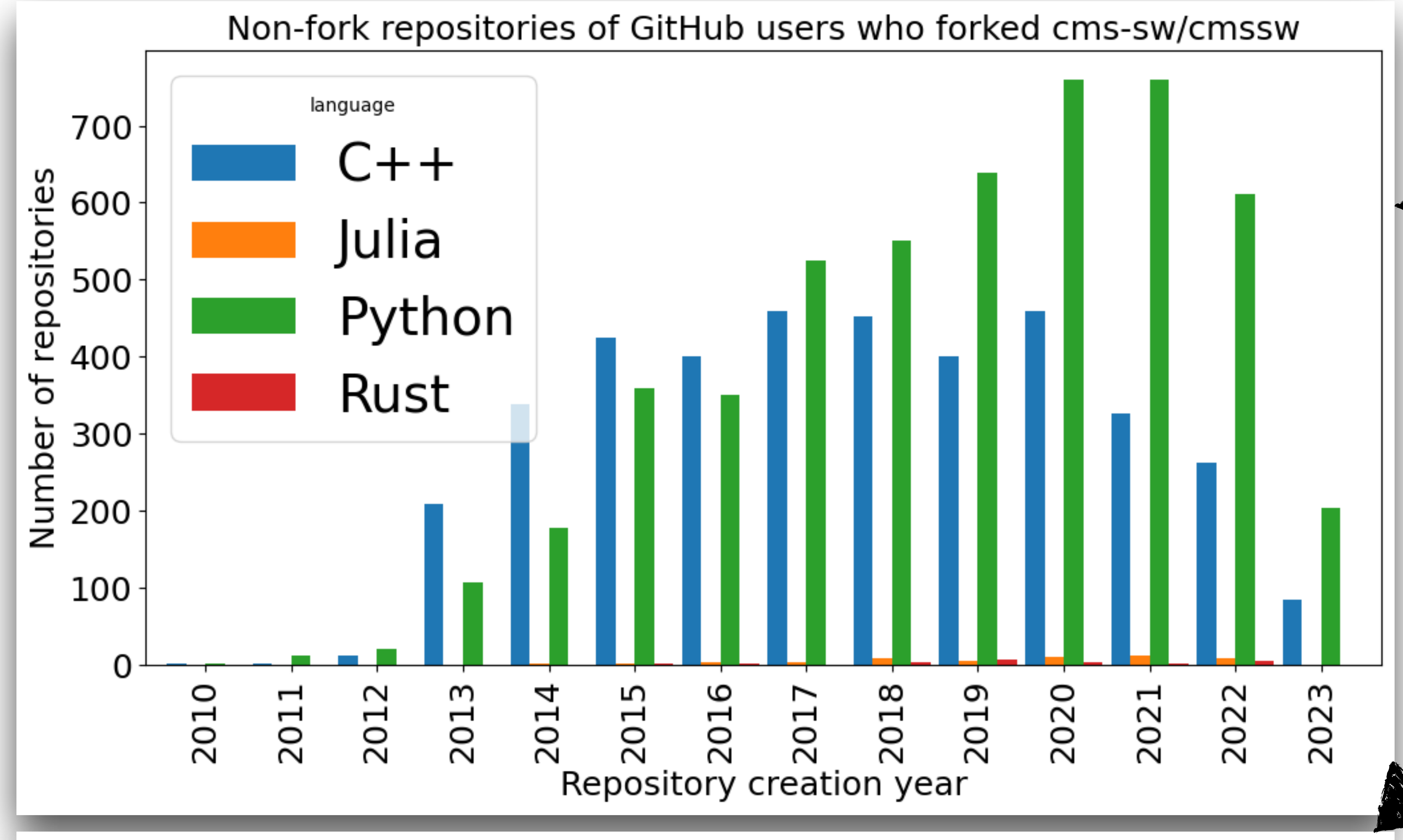
Complex and nested data structures

where I encountered HEP

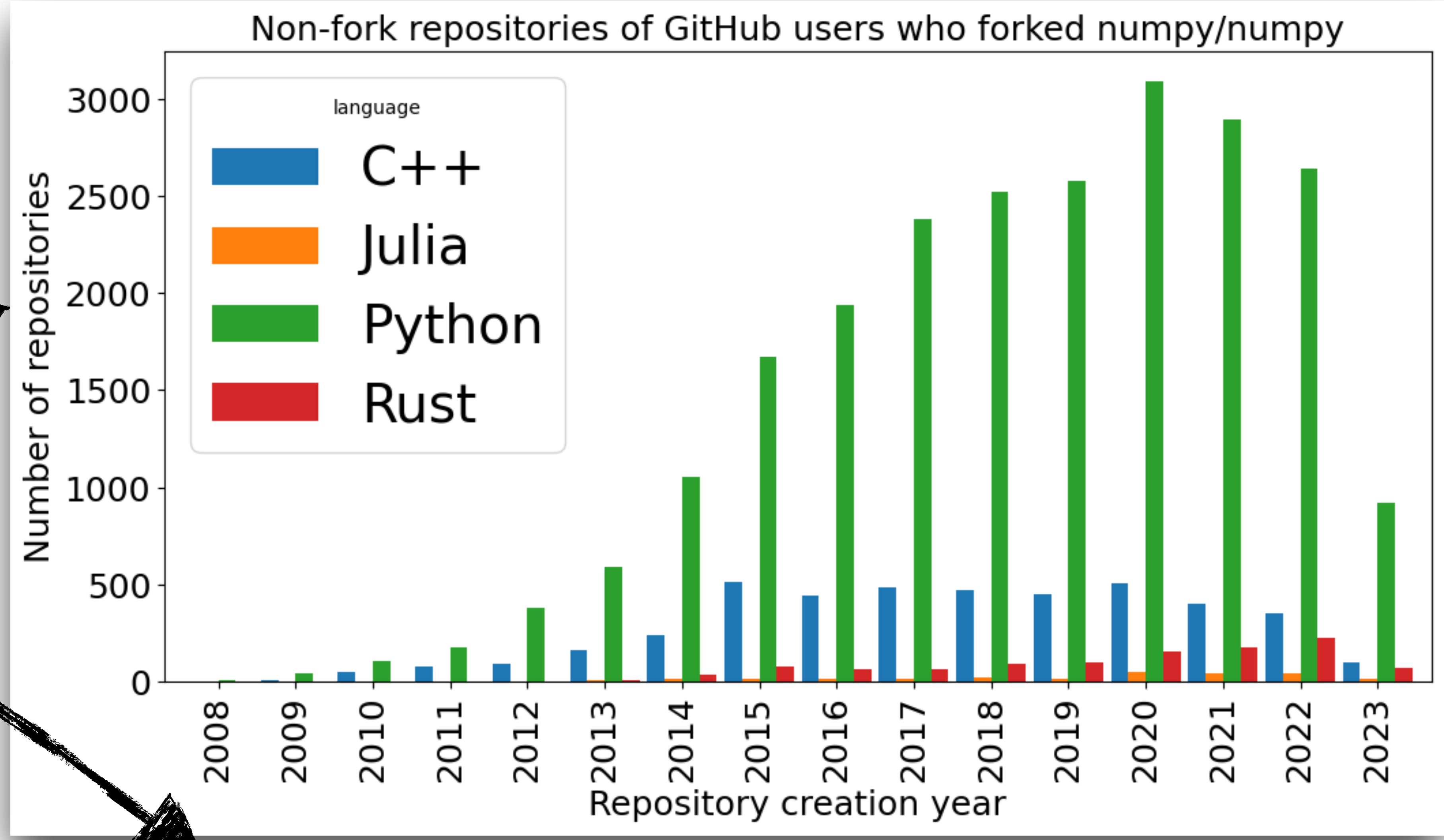
Interactivity, ease of use, packaging

Language usage development in the past 13 years

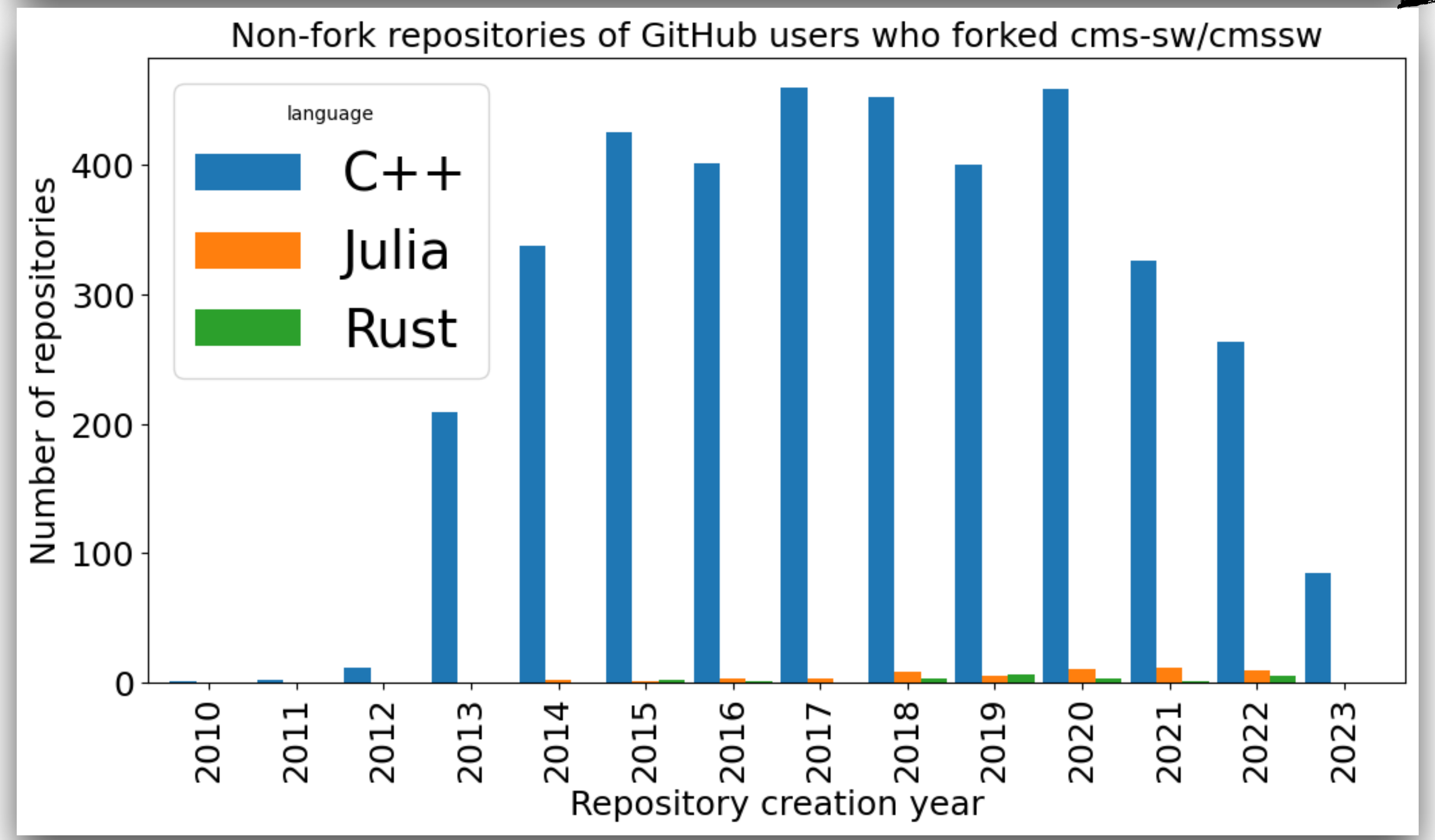
Based on counting non-fork GitHub repositories created by people who forked a specific software.



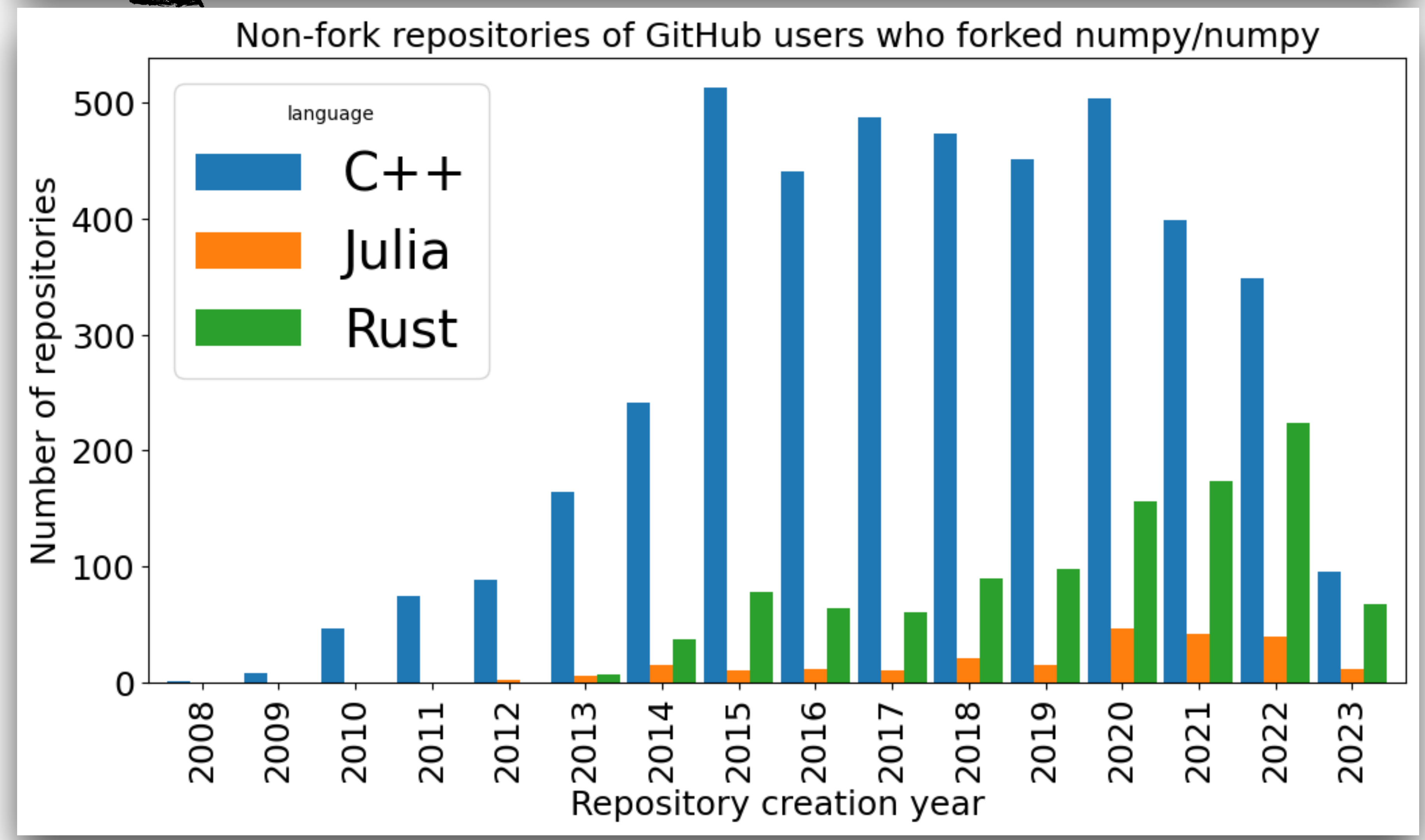
cmssw users
("**HEP**")



numpy users
("**data scientist**")



- **Python peaked** in **2020/2021**
- **Julia** is slowly emerging
- "**HEP**" seems to **follow the** "data scientist" **trend**
- **Turn-over** point of **Rust vs. C++** on the horizon for "data scientists"

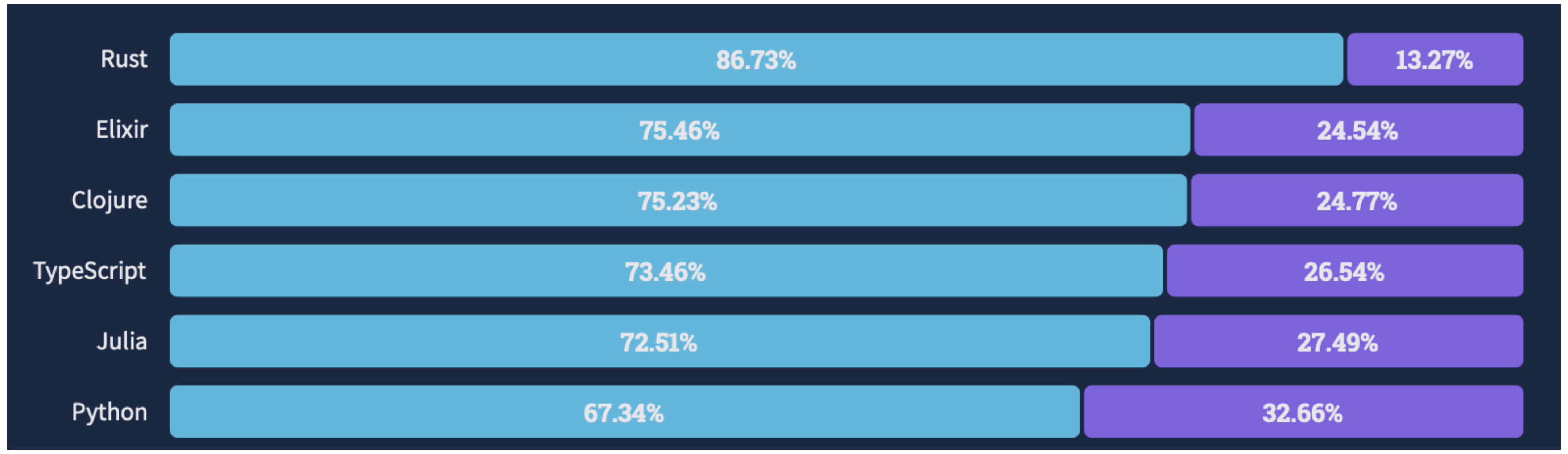


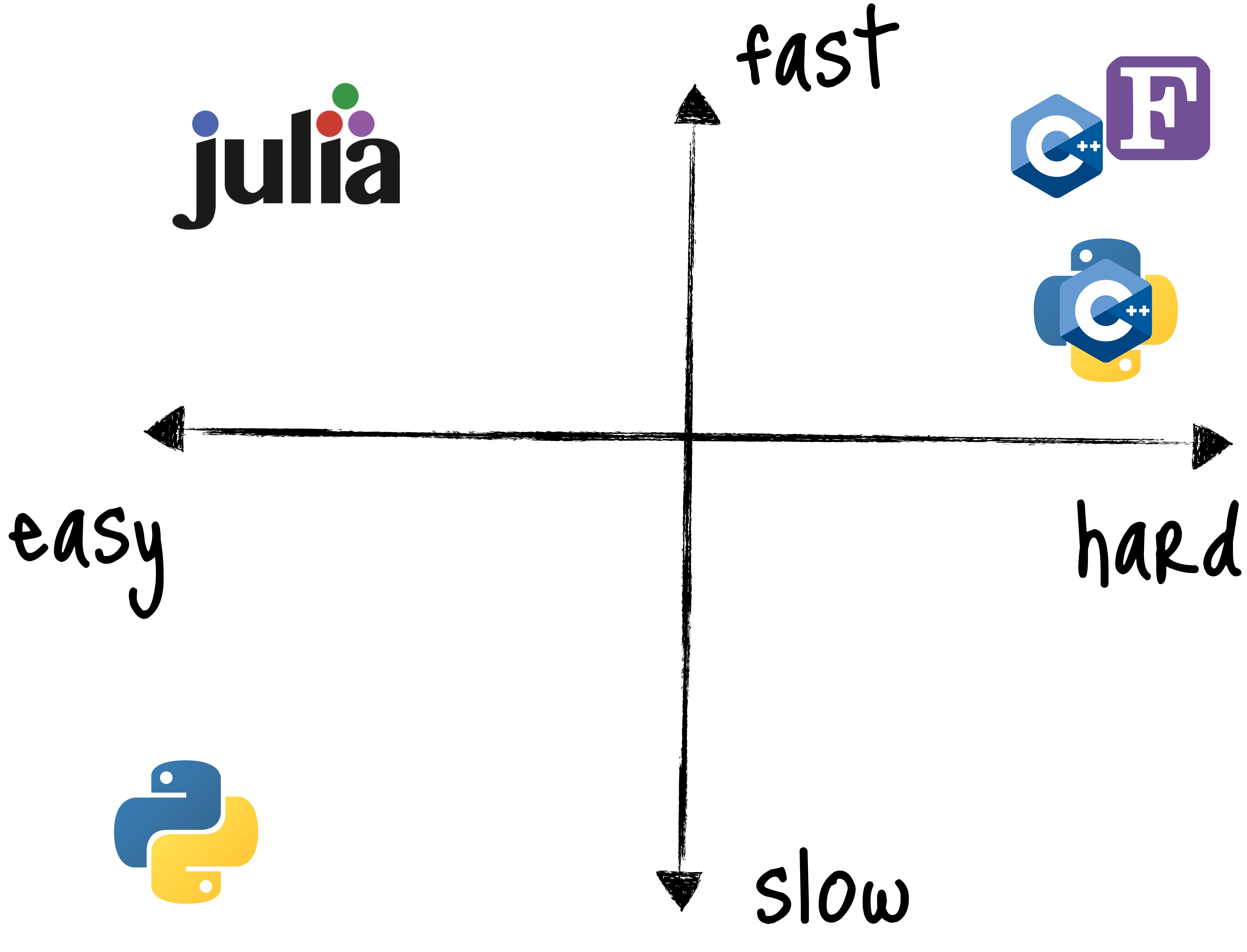
Which language would we have picked in 2013 if we had to choose from today's programming languages?

We think **Julia** is a suitable candidate.

- **High-level** ("easy" and interactive) language without penalty on **performance**
- **Massive code reuse** and **sharing** due to the **multiple-dispatch** design
- **Interface with legacy code** written in different languages
- Well-designed **packaging/distribution system**
- **Parallel and distributed computing** are core features of Julia
- Ability to write **GPU kernels in native Julia**

Most loved languages (top 6 shown) https://survey.stackoverflow.co/2022

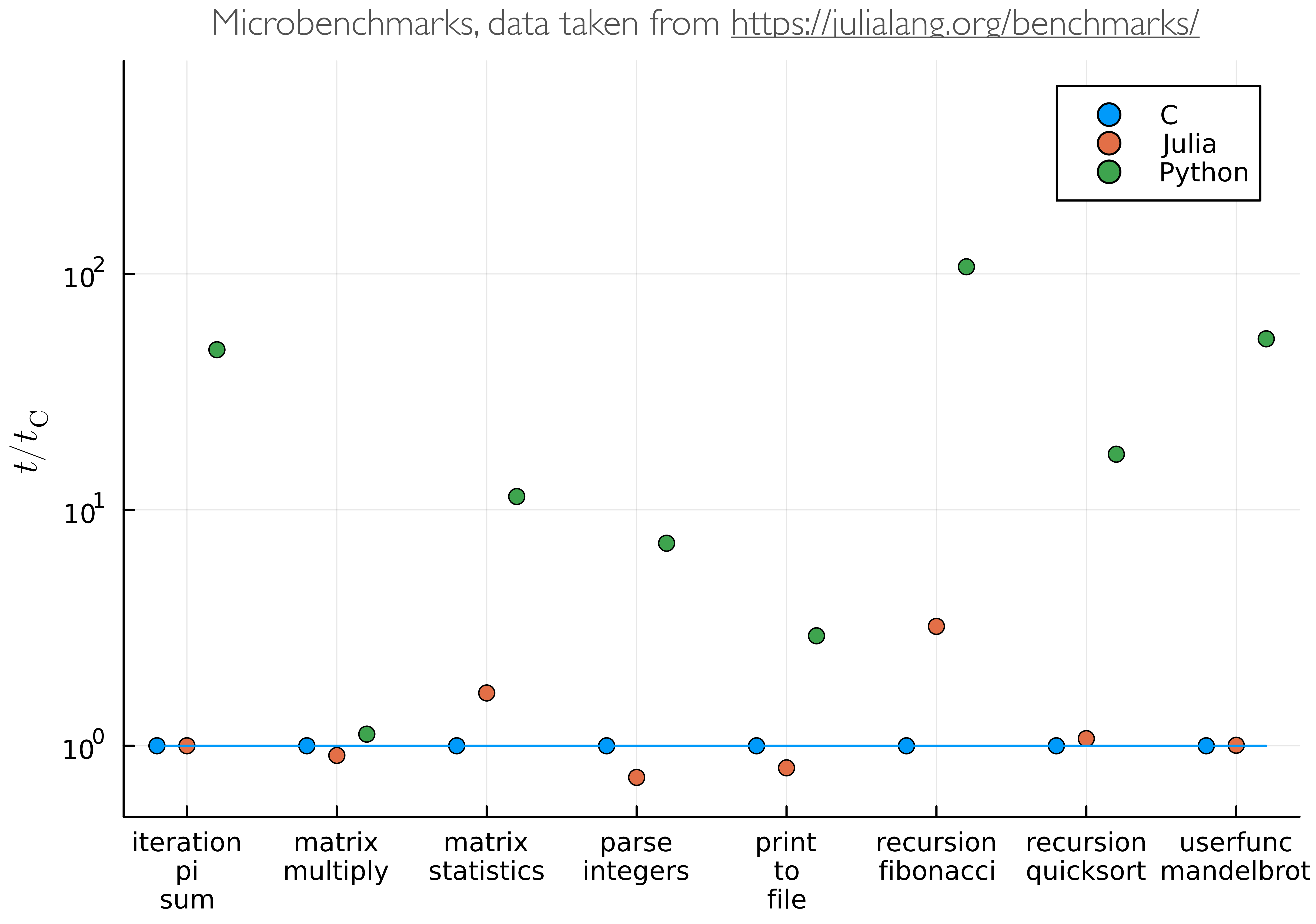




Julia's native speed (compared to C and Python)

Microbenchmarks

- Code "**naively**" **written in Julia** is often close to the peak performance
- It's a big deal since **physics students** do not have CS education and often **approach problems "naively"**
    - Such a code is (according to my experience) often **1-2 orders of magnitude slower** than it should be
    - **memory issues** all over the place (vectorised operations with unnecessary temporary allocations)
    - **bad scaling** due to "whole-meal" programming style
- "**Julia: A language that walks like Python, runs like C" --** K. S. Kuppusamy



Accessing data formats used in HEP

The entry point...

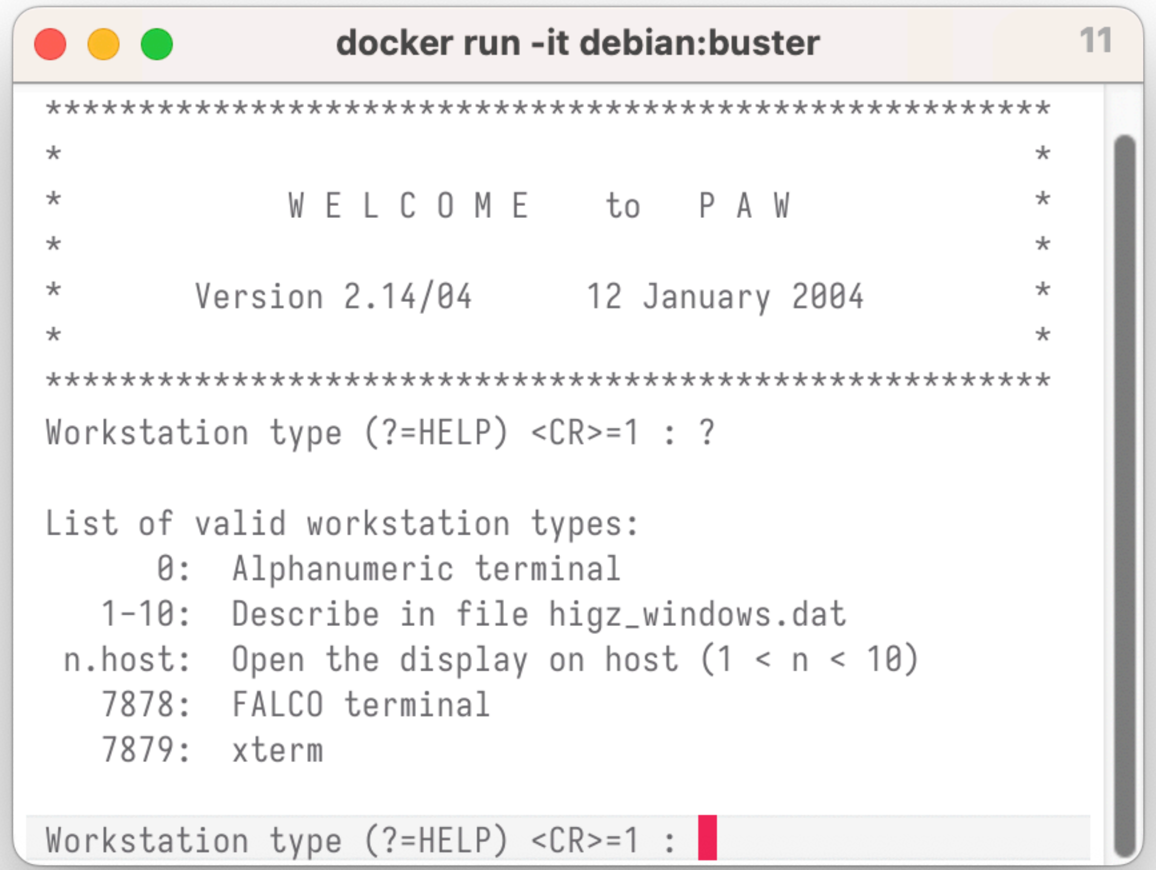
- Being able to **read** (write) data is **essential**
- The most **popular data formats** used in **HEP** are supported with **native Julia** packages*
- Addition formats can be introduced to HEP through Julia

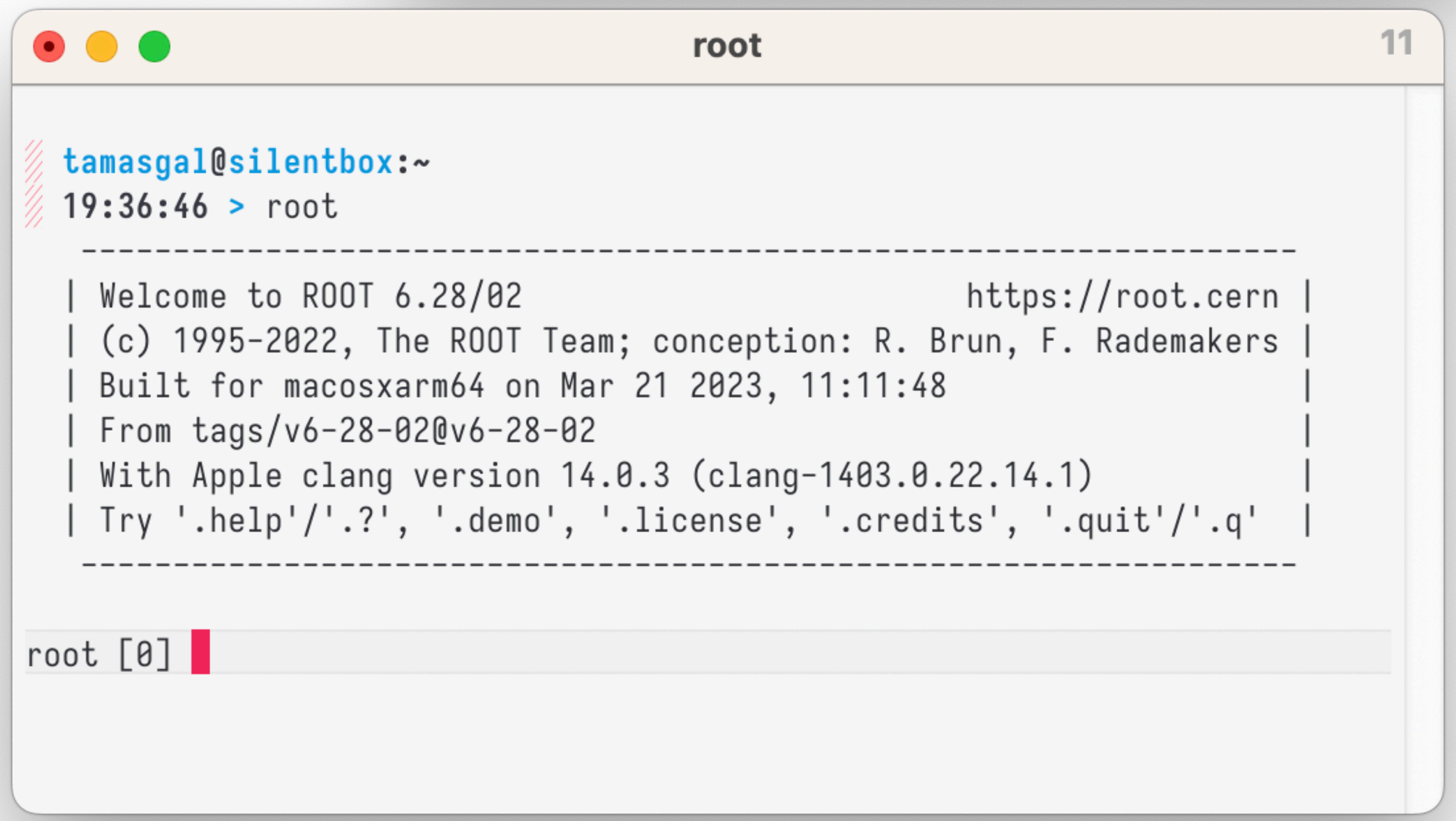
- ROOT Data Analysis Framework → UpROOT.jl, UnROOT.jl
- Les Houches Event File Format → LHE.jl
- LCIO → LCIO.jl
- APACHE ARROW → Arrow.jl
- HDF → HDF5.jl

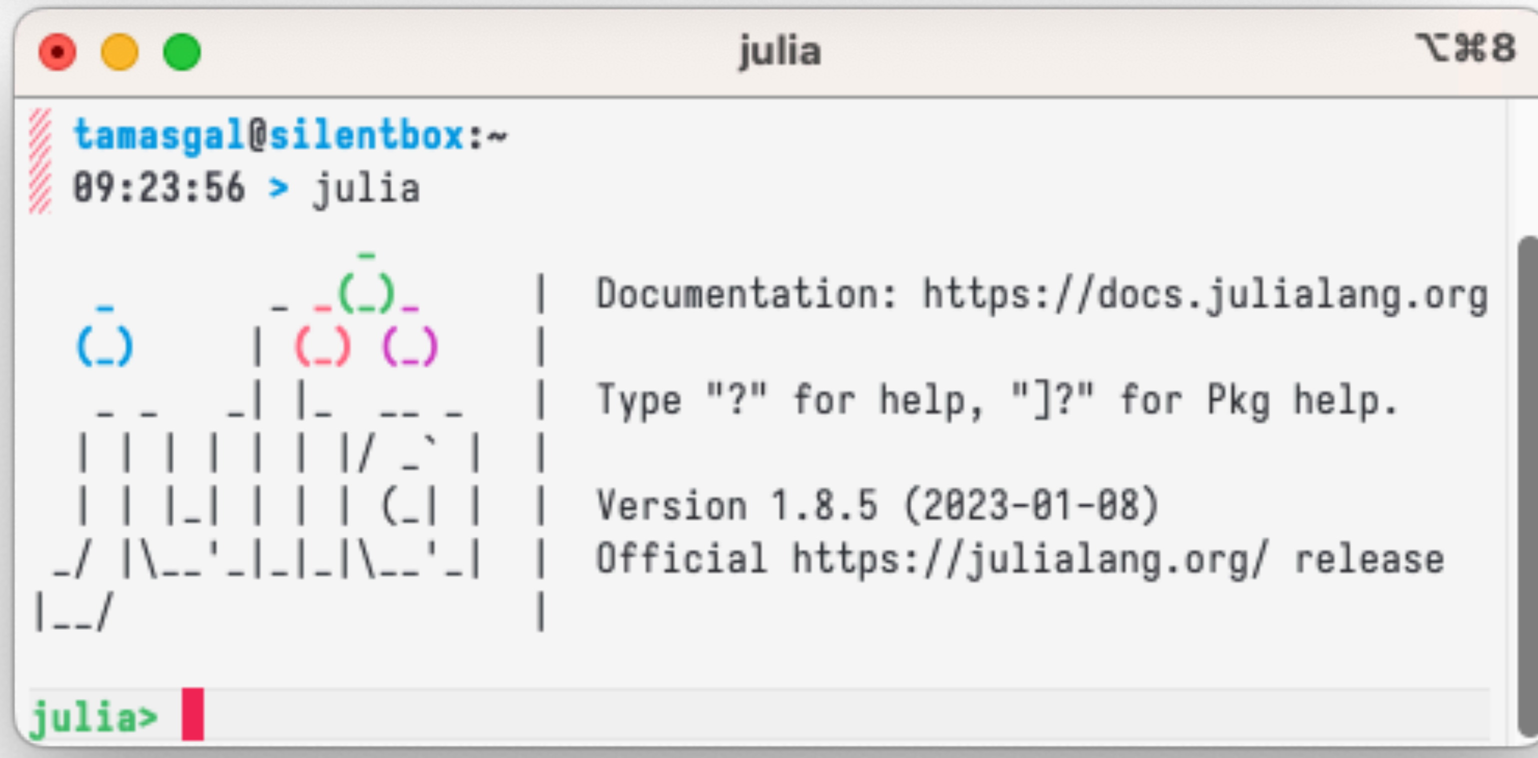
* reading of ROOT files has some limitations
writing ROOT relies on the Python package uproot

High-level and interactive coding

Without penalty on performance







- **Interactive scientific computing** for rapid prototyping has a **long history** in **HEP**, introduced by **PAW** (1986) at CERN and later in **ROOT** (**CINT** 1995, **Cling** 2013)
- **Python** among other languages **popularised the REPL** in other **scientific fields**
- **Julia offers** the **same interactivity** without penalty on **performance**
- **Type inference** allows **generic programming** and yet type safety and optimised machine code
- **Jupyter** notebook support (btw. **Ju** stands for **Julia**...)

Code reusability and extensibility

"The Expression Problem"

Code reusability and extensibility

"The Expression Problem"

- The ability to easily **define new types to which existing operations apply**

Code reusability and extensibility

"The Expression Problem"

- The ability to easily **define new types to which existing operations apply**
  - **Easy in object-oriented languages / Hard in functional languages**

Code reusability and extensibility

"The Expression Problem"

- The ability to easily **define new types to which existing operations apply**
    - **Easy in object-oriented languages** / **Hard in functional languages**
- The ability to easily **define new operations which apply to existing types**

Code reusability and extensibility

"The Expression Problem"

- The ability to easily **define new types to which existing operations apply**
    - **Easy in object-oriented languages / Hard in functional languages**
- The ability to easily **define new operations which apply to existing types**
    - Easy in functional languages / Hard in object-oriented languages

Code reusability and extensibility

"The Expression Problem"

- The ability to easily **define new types to which existing operations apply**
    - **Easy in object-oriented languages / Hard in functional languages**
- The ability to easily **define new operations which apply to existing types**
    - Easy in functional languages / Hard in object-oriented languages
- **Being able to do both easily is "The Expression Problem"**

Code reusability and extensibility

"The Expression Problem"

- The ability to easily **define new types to which existing operations apply**
    - **Easy in object-oriented languages / Hard in functional languages**
- The ability to easily **define new operations which apply to existing types**
    - Easy in functional languages / Hard in object-oriented languages
- **Being able to do both easily is "The Expression Problem"**

Code reusability and extensibility

"The Expression Problem"

- The ability to easily **define new types to which existing operations apply**
  - **Easy in object-oriented languages / Hard in functional languages**
- The ability to easily **define new operations which apply to existing types**
  - Easy in functional languages / Hard in object-oriented languages
- **Being able to do both easily is "The Expression Problem"**

An **elegant solution** is **multiple-dispatch** – the **main paradigm** of the **Julia language**

Code reusability and extensibility

"The Expression Problem"

- The ability to easily **define new types to which existing operations apply**
  - **Easy in object-oriented languages / Hard in functional languages**
- The ability to easily **define new operations which apply to existing types**
  - Easy in functional languages / Hard in object-oriented languages
- **Being able to do both easily is "The Expression Problem"**

An **elegant solution** is **multiple-dispatch** – the **main paradigm** of the **Julia language**

- **"Generic programming"** and **JIT type inference** allows **mixing code** from different Julia packages

Code reusability and extensibility

"The Expression Problem"

- The ability to easily **define new types to which existing operations apply**
    - **Easy in object-oriented languages / Hard in functional languages**
- The ability to easily **define new operations which apply to existing types**
    - Easy in functional languages / Hard in object-oriented languages
- **Being able to do both easily is "The Expression Problem"**

An **elegant solution** is **multiple-dispatch** – the **main paradigm** of the **Julia language**

- **"Generic programming"** and **JIT type inference** allows **mixing code** from different Julia packages
- Add new methods to **existing generic functions** for **new types**

Code reusability and extensibility

"The Expression Problem"

- The ability to easily **define new types to which existing operations apply**
    - **Easy in object-oriented languages / Hard in functional languages**
- The ability to easily **define new operations which apply to existing types**
    - Easy in functional languages / Hard in object-oriented languages
- **Being able to do both easily is "The Expression Problem"**

An **elegant solution** is **multiple-dispatch** – the **main paradigm** of the **Julia language**

- **"Generic programming"** and **JIT type inference** allows **mixing code** from different Julia packages
- Add new methods to **existing generic functions** for **new types**
- Add new methods to **new generic functions** for **existing types**

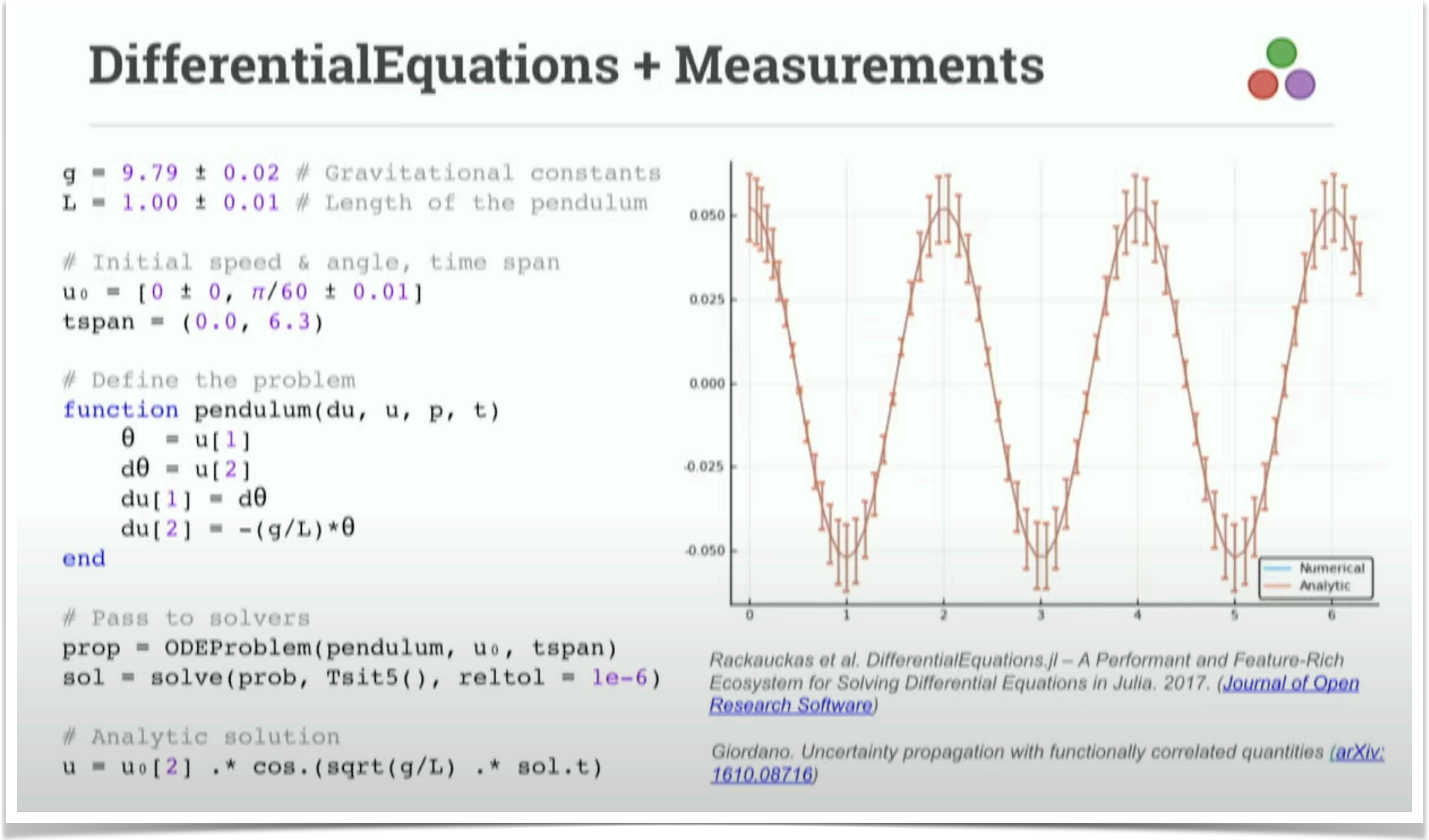
Code reusability and extensibility

"The Expression Problem"

- The ability to easily **define new types to which existing operations apply**
    - **Easy in object-oriented languages** / **Hard in functional languages**
- The ability to easily **define new operations which apply to existing types**
    - Easy in functional languages / Hard in object-oriented languages
- **Being able to do both easily is "The Expression Problem"**

An **elegant solution** is **multiple-dispatch** – the **main paradigm** of the **Julia language**

- **"Generic programming"** and **JIT type inference** allows **mixing code** from different Julia packages
- Add new methods to **existing generic functions** for **new types**
- Add new methods to **new generic functions** for **existing types**



JuliaCon 2019 | **The Unreasonable Effectiveness of Multiple Dispatch** | Stefan Karpinski
https://www.youtube.com/live/kc9HwsxE1OY

Code reusability and extensibility

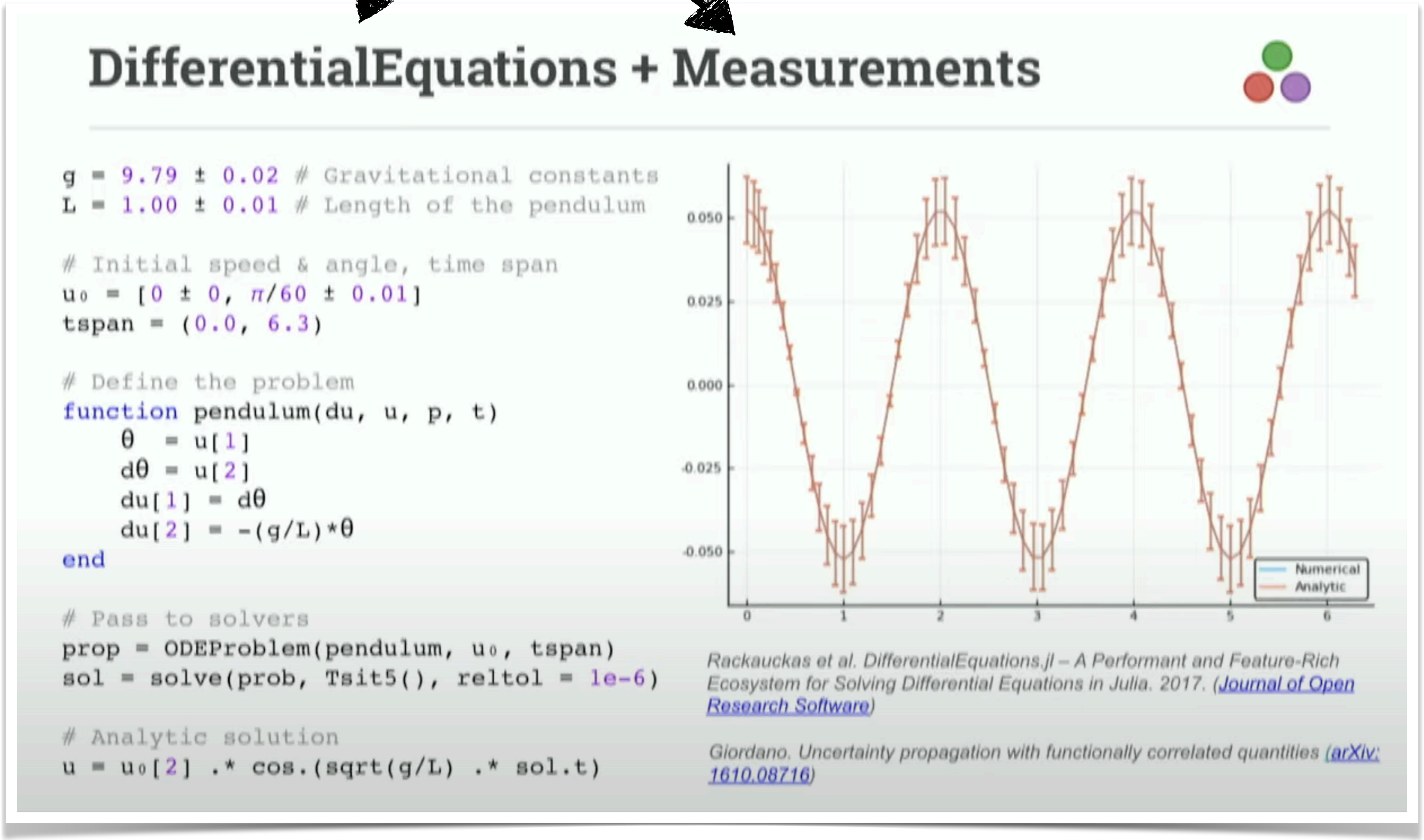
"The Expression Problem"

- The ability to easily **define new types to which existing operations apply**
    - Easy in object-oriented languages / Hard in functional languages
- The ability to easily **define new operations which apply to existing types**
    - Easy in functional languages / Hard in object-oriented languages
- **Being able to do both easily is "The Expression Problem"**

An **elegant solution** is **multiple-dispatch** – the **main paradigm** of the **Julia language**

- **"Generic programming"** and **JIT type inference** allows **mixing code** from different Julia packages
- Add new methods to **existing generic functions** for **new types**
- Add new methods to **new generic functions** for **existing types**

These two packages don't know about each other!



JuliaCon 2019 I **The Unreasonable Effectiveness of Multiple Dispatch** I Stefan Karpinski
https://www.youtube.com/live/kc9HwsxE1OY

Interfacing legacy code

- Many high-quality, **mature libraries** for numerical computing written in C and Fortran were developed and optimised over **the past decades**
- Julia supports **native call** (without any glue code) **into C and Fortran** libraries (via the built-in `ccall()` function)
- **C++** wrapping available via external packages like **CxxWrap.jl**
- **Zero-overhead Python** wrapping (**PyCall.jl**)
- An honorable mention for a **fully wrapped HEP** software
    - **Geant4.jl** (fully wrapped using CxxWrap.jl)

```
julia> using PyCall

julia> np = pyimport("numpy");

julia> np.random.rand(3) * 100
3-element Vector{Float64}:
 38.961726053176136
 71.3368957480925
  8.307181033489208

julia> np.sin(rand(5, 2))
5×2 Matrix{Float64}:
 0.784982   0.282252
 0.202079   0.220945
 0.637406   0.0921307
 0.0869371  0.395478
 0.383479   0.150941

julia>
```

Julia's packaging and distribution system

Reproducible environments, private package registries

- **Reproducible environments** with **exact versions** of all **dependencies** is a **built-in** feature in **Julia**
- **Private package registries** can be utilised to distribute unpublished packages, **seamless integration** into the package dependency solver
- **Distribution** of **pre-built binaries** of external dependencies (e.g. HDF5lib, libdeflate, ...) for a **large combinatorics of OS**, **architectures**, **compiler features**, etc.

Parallel, Distributed and GPU Computing
"Built-in" or "built for" ;)

Parallel, Distributed and GPU Computing

"Built-in" or "built for" ;)

- Loops can **easily** be **parallelised** by adding a **keyword** (macro-/meta- programming)

Parallel, Distributed and GPU Computing

"Built-in" or "built for" ;)

- Loops can **easily** be **parallelised** by adding a **keyword** (macro-/meta- programming)

```
julia> for event ∈ mytree
           # process event
       end
```

Parallel, Distributed and GPU Computing

"Built-in" or "built for" ;)

- Loops can **easily** be **parallelised** by adding a **keyword** (macro-/meta- programming)

```
julia> for event ∈ mytree
           # process event
       end
```

```
julia> Threads.@threads for event ∈ mytree
           # process event
       end
```

Parallel, Distributed and GPU Computing

"Built-in" or "built for" ;)

```
julia> for event ∈ mytree
           # process event
       end
```

```
julia> Threads.@threads for event ∈ mytree
           # process event
       end
```

- Loops can **easily** be **parallelised** by adding a **keyword** (macro-/meta- programming)

- **Loop optimization** with processor-level parallelisation **(SIMD).** Can be **fine-tuned** with **third-party packages** like **LoopVectorization.jl.**
  Related talk here at CHEP 2023 from **Graeme Stuart**
  https://indico.jlab.org/event/459/contributions/11540

Parallel, Distributed and GPU Computing

"Built-in" or "built for" ;)

- Loops can **easily** be **parallelised** by adding a **keyword** (macro-/meta- programming)

- **Loop optimization** with processor-level parallelisation **(SIMD).** Can be **fine-tuned** with **third-party packages** like **LoopVectorization.jl.**
  Related talk here at CHEP 2023 from **Graeme Stuart**
  https://indico.jlab.org/event/459/contributions/11540

```
julia> for event ∈ mytree
           # process event
       end
```

```
julia> Threads.@threads for event ∈ mytree
           # process event
       end
```

```
julia> function dotbaseline(a::AbstractArray{T}, b::AbstractArray{T}) where {T}
           s = zero(T)
           @fastmath @inbounds @simd for i ∈ eachindex(a,b)
               s += a[i]' * b[i]
           end
           s
       end
```

Parallel, Distributed and GPU Computing

"Built-in" or "built for" ;)

- Loops can **easily** be **parallelised** by adding a **keyword** (macro-/meta- programming)

```
julia> for event ∈ mytree
           # process event
       end
```

```
julia> Threads.@threads for event ∈ mytree
           # process event
       end
```

- **Loop optimization** with processor-level parallelisation **(SIMD).** Can be **fine-tuned** with **third-party packages** like **LoopVectorization.jl.**
  Related talk here at CHEP 2023 from **Graeme Stuart**
  https://indico.jlab.org/event/459/contributions/11540

```
julia> function dotbaseline(a::AbstractArray{T}, b::AbstractArray{T}) where {T}
           s = zero(T)
           @fastmath @inbounds @simd for i ∈ eachindex(a,b)
               s += a[i]' * b[i]
           end
           s
       end
```

- An impressive example from **KernelAbstractions.jl** which allows **Julia code** to be passed as a **kernel** function **to GPUs**:

Parallel, Distributed and GPU Computing

"Built-in" or "built for" ;)

- Loops can **easily** be **parallelised** by adding a **keyword** (macro-/meta- programming)

```
julia> for event ∈ mytree
           # process event
       end
```

```
julia> Threads.@threads for event ∈ mytree
           # process event
       end
```

- **Loop optimization** with processor-level parallelisation **(SIMD).** Can be **fine-tuned** with **third-party packages** like **LoopVectorization.jl.**
  Related talk here at CHEP 2023 from **Graeme Stuart**
  https://indico.jlab.org/event/459/contributions/11540

```
julia> function dotbaseline(a::AbstractArray{T}, b::AbstractArray{T}) where {T}
           s = zero(T)
           @fastmath @inbounds @simd for i ∈ eachindex(a,b)
               s += a[i]' * b[i]
           end
           s
       end
```

- An impressive example from **KernelAbstractions.jl** which allows **Julia code** to be passed as a **kernel** function **to GPUs**:

```
@kernel function mul2_kernel(A)
  I = @index(Global)
  A[I] = 2 * A[I]
end
```

Parallel, Distributed and GPU Computing

"Built-in" or "built for" ;)

- Loops can **easily** be **parallelised** by adding a **keyword** (macro-/meta- programming)

```
julia> for event ∈ mytree
           # process event
       end
```

```
julia> Threads.@threads for event ∈ mytree
           # process event
       end
```

- **Loop optimization** with processor-level parallelisation **(SIMD).** Can be **fine-tuned** with **third-party packages** like **LoopVectorization.jl.**
  Related talk here at CHEP 2023 from **Graeme Stuart**
  https://indico.jlab.org/event/459/contributions/11540

```
julia> function dotbaseline(a::AbstractArray{T}, b::AbstractArray{T}) where {T}
           s = zero(T)
           @fastmath @inbounds @simd for i ∈ eachindex(a,b)
               s += a[i]' * b[i]
           end
           s
       end
```

- An impressive example from **KernelAbstractions.jl** which allows **Julia code** to be passed as a **kernel** function **to GPUs**:

```
@kernel function mul2_kernel(A)
  I = @index(Global)
  A[I] = 2 * A[I]
end
```

- **Distributed** (built-in): execute code **asynchronously** in **multiple processes** and/or **multiple machines** (like MPI)

Summary

- We think that the **two-language problem needs** more attention and a **fundamentally different approach than** creating more and more **Python extensions and libraries**
- **Julia** is an **excellent language for scientific computing** with **high potential for HEP**
- **HEP specific needs** are very **well covered** by Julia
- **Code sharing** and **extending** foreign packages **are a no-brainer**, thanks to the package distribution system and the **multiple dispatch** design
- **Distributed and parallel computing** as first-class citizens in Julia
- Upcoming paper:
  Potential of the Julia language for High Energy Physics computing
- Join the **JuliaHEP GitHub** organisation**:** https://github.com/JuliaHEP