R - Python - Julia

Insights into old and new languages for data science and machine learning and implications for their use in (high performance) production environments

Simon Wenkel

https://www.simonwenkel.com

useR Tallinn - January 16th, 2020

License Info

Contents

Contents

What you (don't) get in this talk

- No recommendations what language to use (R, Python, Julia, C, Fortran, etc.) but things to consider when (re)-designing a product from scratch

Benchmark disclaimer

- setup often unclear
- generic code vs. hand-optimized
- micro-benchmarks vs. end-to-end benchmark
- data set properties not well documented

... but from personal experience: most benchmark results give good indications ...

(High performance) production environments

- **fundamental trade-offs**
  - implementation time vs. run-time performance
  - salaries vs. infrastructure costs
- **challenges**
  - licensing issues (e.g. some Boost (C++) libraries)/dependency hell
  - prototyping speed
  - performance issues with interpreted languages
  - program in C/C++/F03 and make it fast, secure and memory safe
- **general setting**
  - "manual workflow": 1 hour vs. 7 day coffee break
  - max. allowed runtime - product useless otherwise
- **implications of (simple) 10x speed-ups (R/Python/Julia)**
  - *(if all bottlenecks are fixed)*
  - serving the same amount of customers with a fraction of hardware
  - if real-time requirements: product/no product
  - much faster prototyping
  - no C/C++ conversion department needed

Contents

Introducing Julia (1)

"[...] We've **generated more R plots than any sane person should**. C is our desert island programming language.
We love all of these languages; they are wonderful and powerful. For the work we do — **scientific computing, machine learning, data mining, large-scale linear algebra, distributed and parallel computing** — each one is perfect for some aspects of the work and terrible for others. **Each one is a trade-off**. [...]

We want a language that's open source, with a liberal license. We want the **speed of C** with the dynamism of Ruby. We want a language that's homoiconic, with **true macros like Lisp**, but with obvious, familiar **mathematical notation like Matlab**. We want something as usable for **general programming** as **Python**, as easy for **statistics as R**, as natural for **string processing** as **Perl**, 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. We want it **interactive and** we want it **compiled**.[...]"

`https://julialang.org/blog/2012/02/why-we-created-julia/`

Introducing Julia (2)

- backed by MIT
- many packages developed by US gov. (funded) institutions
- not just another language - aims to solve a bunch of significant problems
- mostly implemented in itself
- very good packages for mathematical optimization (written in Julia instead of Fortran 77)
- strong use cases (so far): numerics, mathematical optimization
- number of users in academia and industry grows rapidly
  - seems to start replacing Matlab
  - gets a lot of attention in math heavy industries (e.g. engineering, finance/insurance)
- seems to raise awareness in statistics, less in machine learning (yet)

Introducing Julia - code example (1)

```julia
# import libraries
using CSV # exports functions
import Flux # functions accessed via Flux.function

# supports pipelines (code from https://juliadata.github.io/CSV.jl/stable/)
db = SQLite.DB()
tbl = CSV.File(file) |> SQLite.load!(db, "sqlite_table")

# function definition
function add_one(Ω::UInt8)
    Ω+=i
    return Ω # in this case the return statement is optional
end          # without return, the last expression is returned

# multiple dispatch and inline function definition
add_one(α::Float64) = α+=1
```

Introducing Julia - code example (2)

```julia
# macros
@time add_one(1)
@time add_one(1.)

# structs instead of classes
struct airplane
    model::String
    engine_ID::UInt64
end

# dictionaries available
dict = Dict("a" => 1, "b" => 3)
# creates: Dict{String,Int64}
```

Introduction

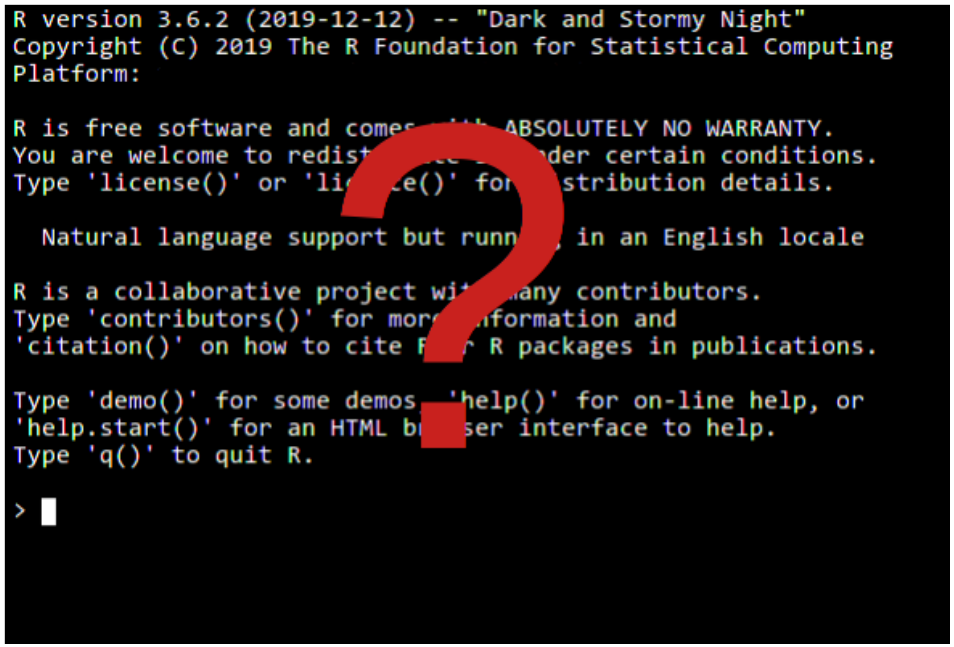
			
Release Year	1993 (S: 1976)	1990	2012
License	GPL (v2) (core + (most?) packages), (tidyverse: GPLv3, MIT, on github: copyright/no license?)	PSFL (packages: BSD, MIT, Apache, GPL)	MIT (core + most packages)
Typing Discipline	dynamic	duck, dynamic, gradual	dynamic, nominative, parametric, optional
Language Type (default)	interpreted	interpreted	compiled JIT (via LLVM)

Common features

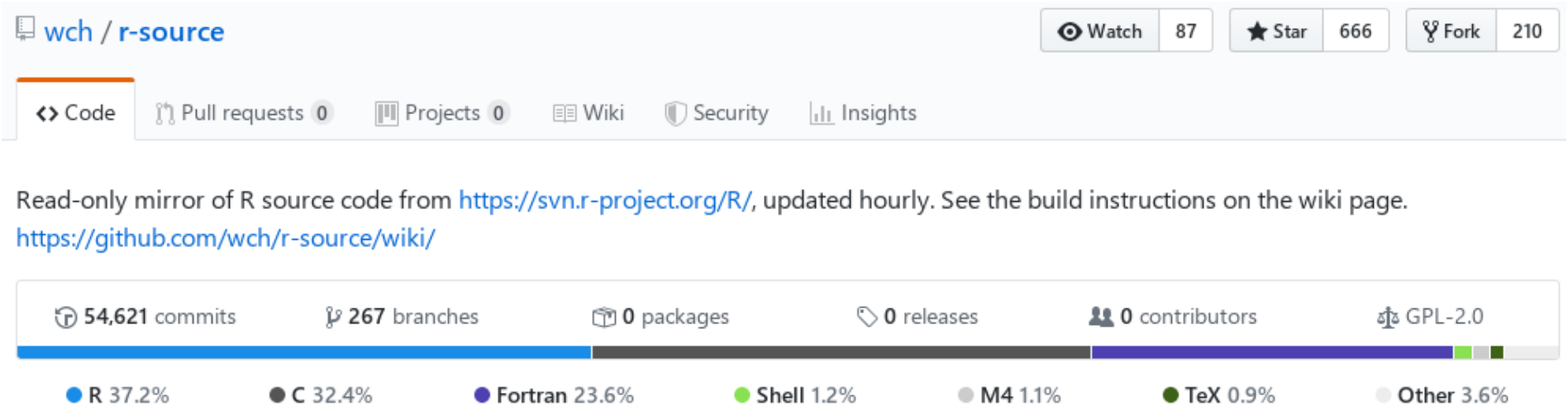
- can use Jupyter notebooks (and RMarkdown)
- can use software written in other languages (FFIs)
- Garbage collected

Contents

Are we really using what we think we are using?



Language source code - R

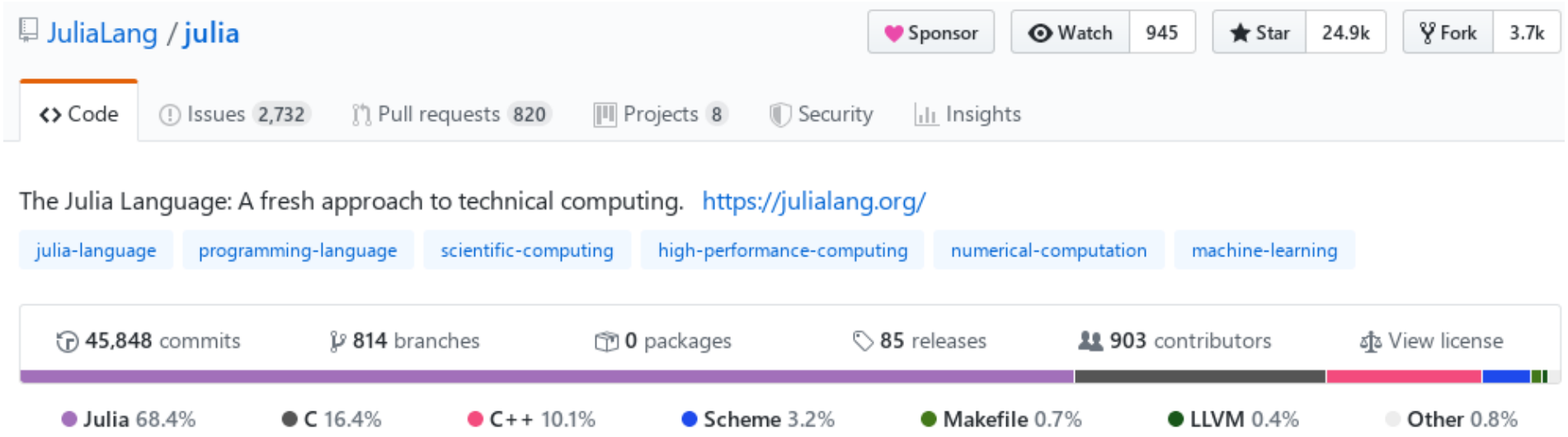


- C and Fortran: almost the entire stdlib written in them
- R: datasets, high-level functions/data structures, constants, documentation, tests; not used for intense computing/math functions?

Language source code - (C)Python

- C/C++: almost the entire stdlib written in them, especially everything performance critical
- Python: high-level functions/classes/data structures, constants, some core libraries, documentation, tests; not used for intense computing/math functions

Language source code - Julia

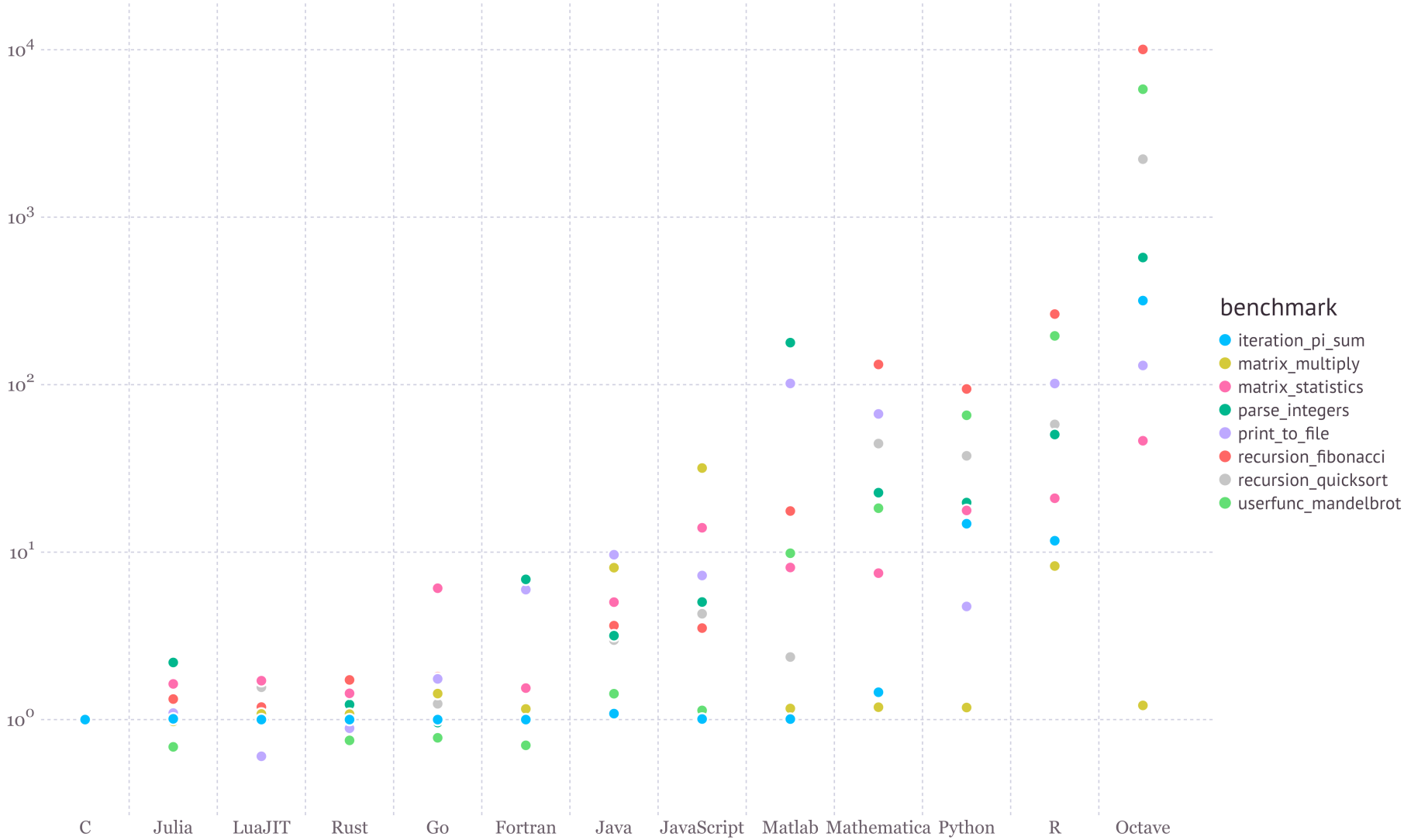


- C/C++: core functions (system level, e.g. OS support), LLVM backend, FFI (Foreign Function Interface)
- Julia: almost everything else (incl. stdlib)

Micro-Benchmarks

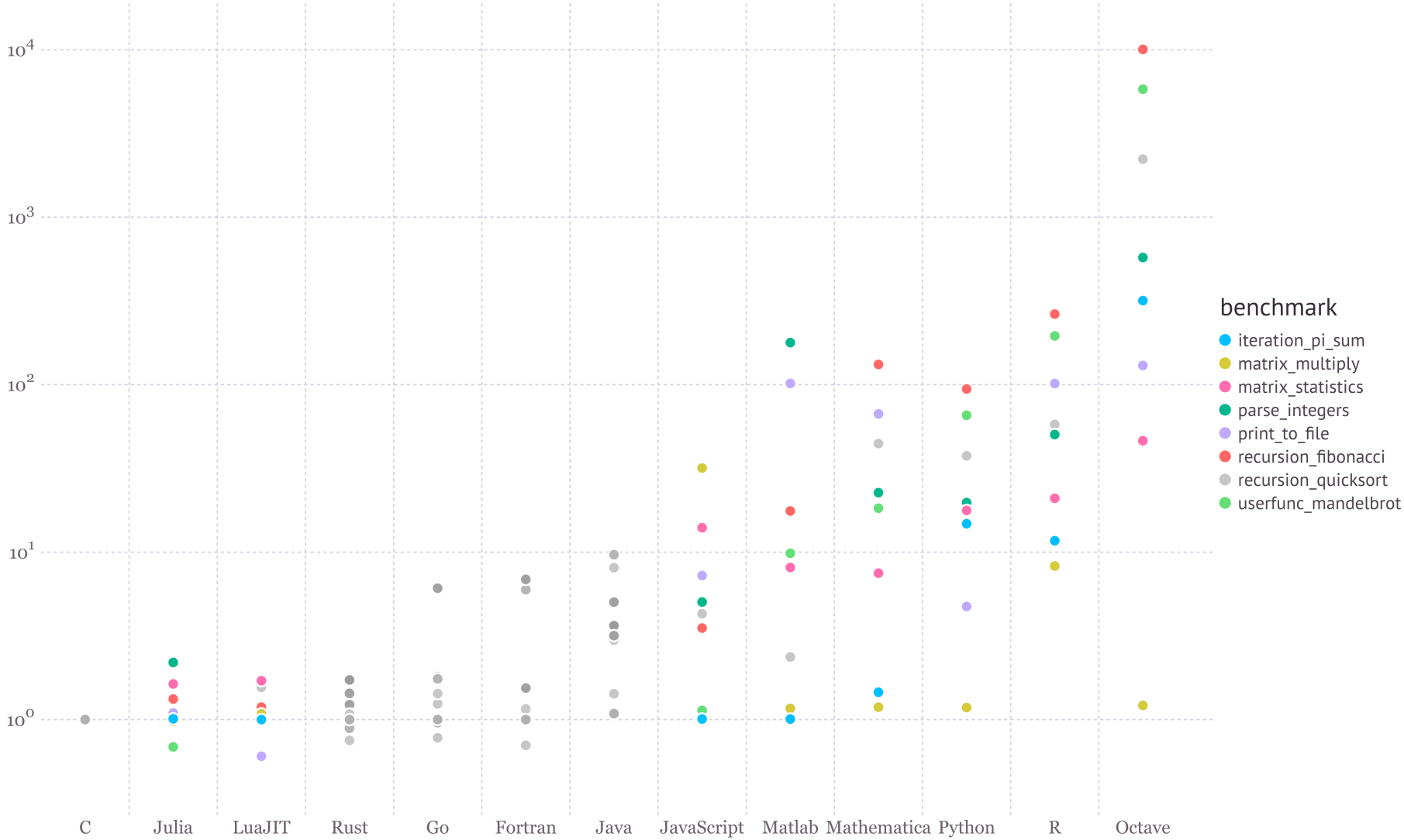
source: `https://julialang.org/benchmarks/` [1]

Micro-Benchmarks



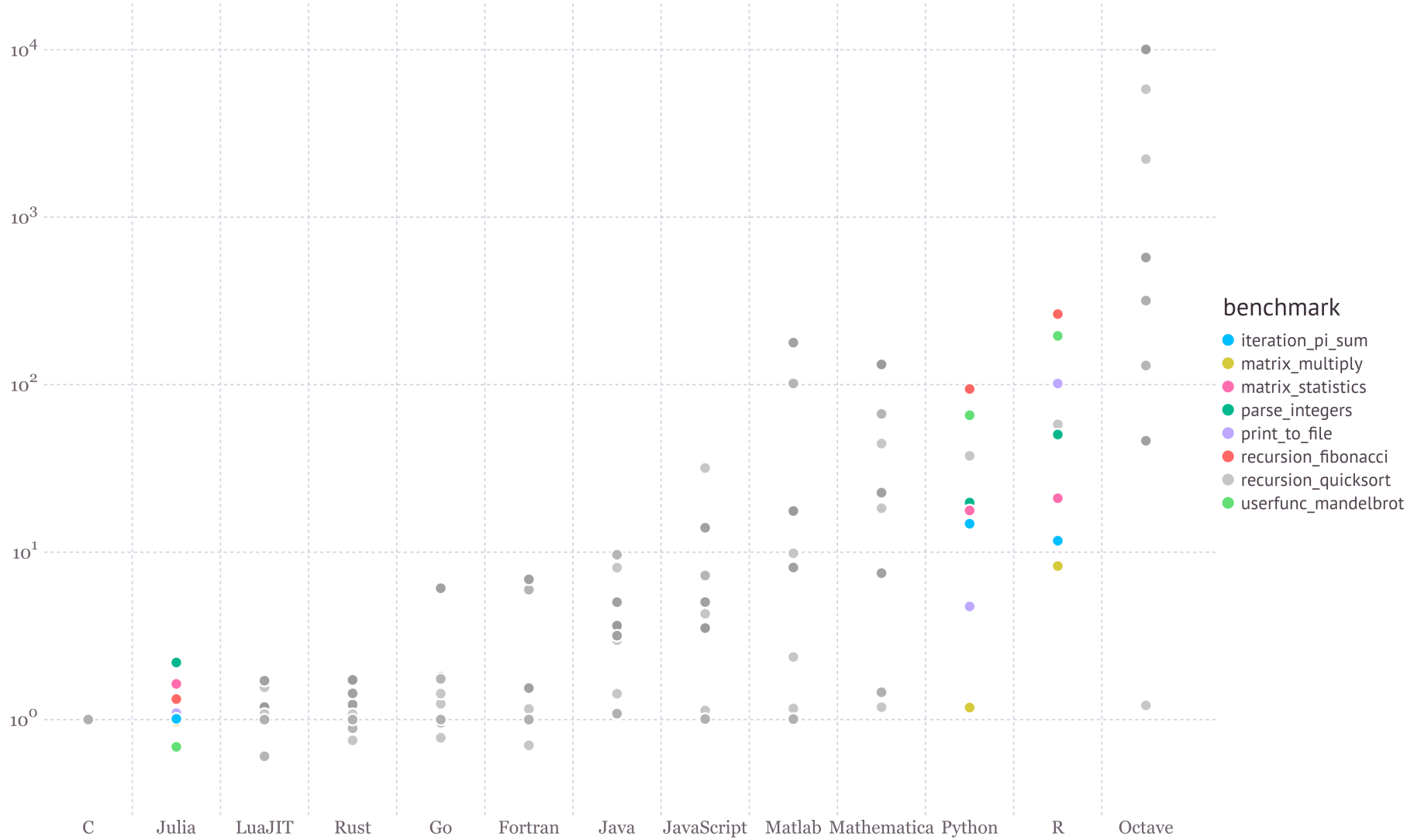
https://julialang.org/benchmarks/ [1]

Micro-Benchmarks



https://julialang.org/benchmarks/ [1]

Micro-Benchmarks



https://julialang.org/benchmarks/ [1]

How to make things fast and efficient

- without re-writing our DS/ML pipelines in C/Fortran/CUDA/Rust/etc.
- at almost no development costs (time)
- aiming at orders of magnitude speed-ups (not a few percent improvement)
- **NB!: not going to focus on multi-threading or GPGPU**

Making things fast and efficient - Math Libraries

- BLAS (Basic Linear Algebra Subprograms)
  - written in Fortran
- ATLAS (Automatically Tuned Linear Algebra Software)
  - written in C, Fortran, Pascal, Assembly
  - faster than BLAS
- OpenBLAS
  - optimized BLAS library written in Fortran, Assembly, C
  - much faster than BLAS and faster than ATLAS
  - OpenBLAS leads to **2-10x faster** matrix computation in **R!**(as of 2013 ;))
- Intel MKL (Math Kernel Library)
  - hand-optimized for Intel CPUs in C, C++, Fortran (+ Assembly?)
  - a bit/a lot faster than OpenBLAS depending on application and platform

```
tensorflow/core/platform/cpu_feature_guard.cc:145] This TensorFlow binary
is optimized with Intel(R) MKL-DNN to use the following CPU instructions
in performance critical operations:  SSE4.1 SSE4.2 AVX
```

Other libs: ARPACK-NG, Eigen, LAPACK (with BLAS/ATLAS), cuBLAS, clBLAS, Armadillo, Apple accelerate, ...

Making things fast and efficient - Language level

			julia [1]
best practices	use correct libraries, avoid loops, use functions, use vectorization, don't use dplyr[2] [3, 4, 5, 6]	use correct libraries, avoid loops, use vectorization, don't use pandas	(type definitions), use functions, think twice beforce using vectorization
JIT	R-compiler [3], R-JIT (deprecated?), RIR	PyPy, Numba	part of Julia
(C, C++, etc.) extension **generator**	N/A?	Cython	not necessary

[1] contains more powerful optimization than Numba+Cython [2]

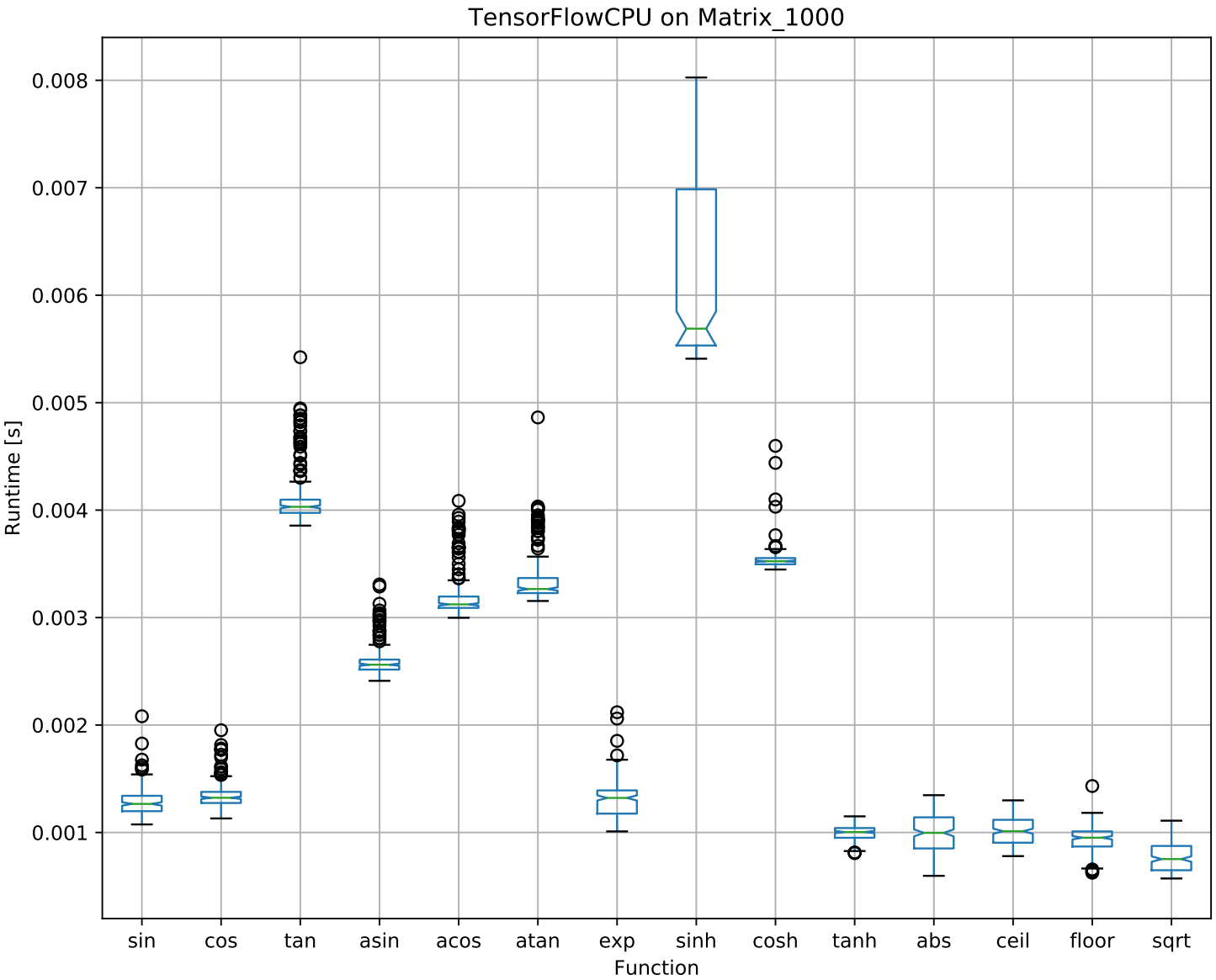
[2] according to it's description: "A fast, consistent tool for working with data frame like objects, both in memory and out of memory."

[3] enabled since R 3.4.0, runs after the 1st or 2nd time a function is used [7]

Choosing the correct library - "basic math"

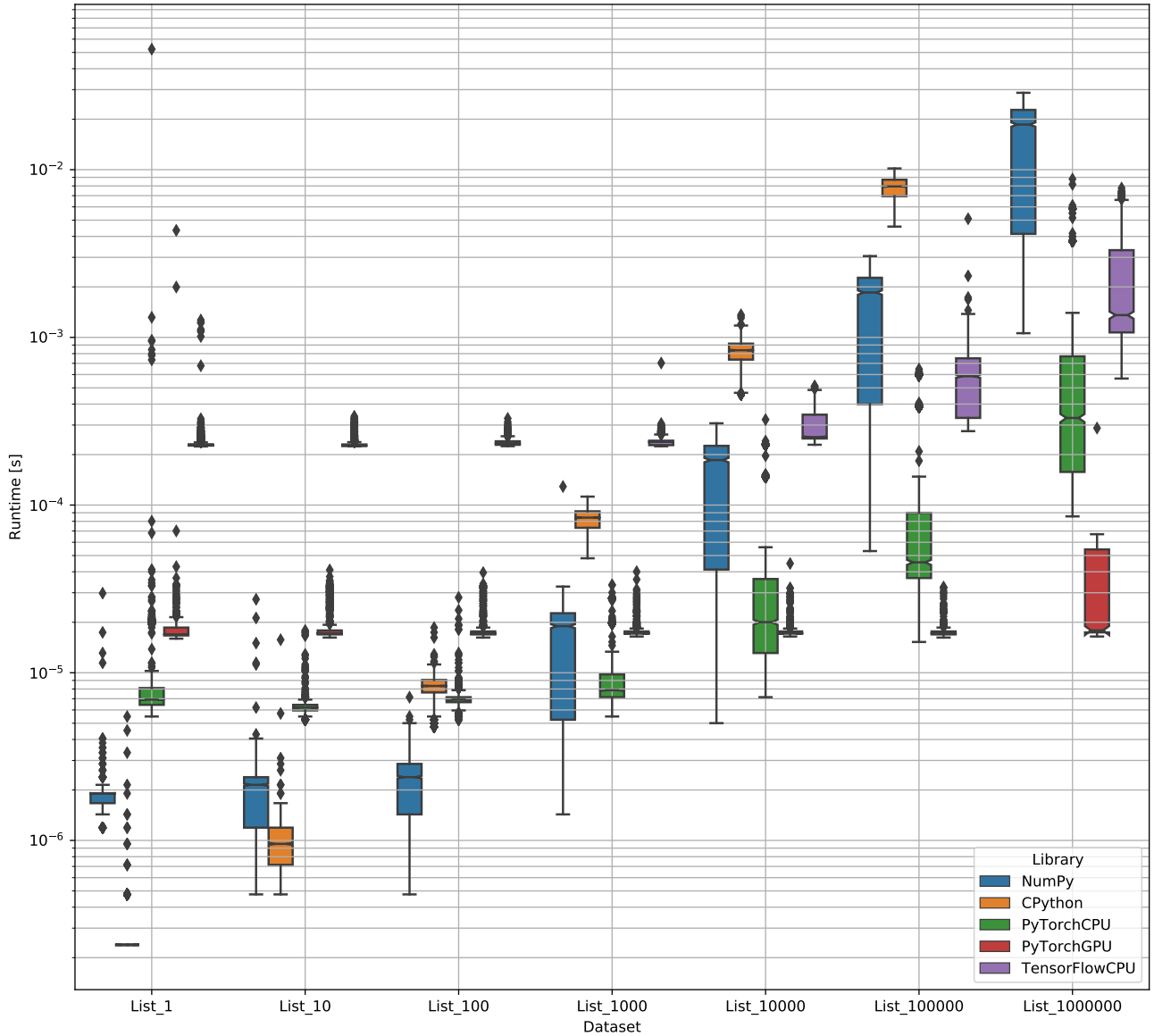
- running a mathematical function (e.g. sin(x)) on:
  - lists of various lengths
  - matrices (square) of various sizes
  - no return/output, only input and calculations
  - both reach sizes - should be split into smaller chunks for parallel processing
- 200 iterations per function and dataset
- *not a trivial benchmark that indicates advantages/disadvantages of loops*
- original idea: best Python library for different list/array size
- **NB!: Garbage Collection!**
- **NB!: log-log plots!**

Choosing the correct library - Python “basic math” (1)



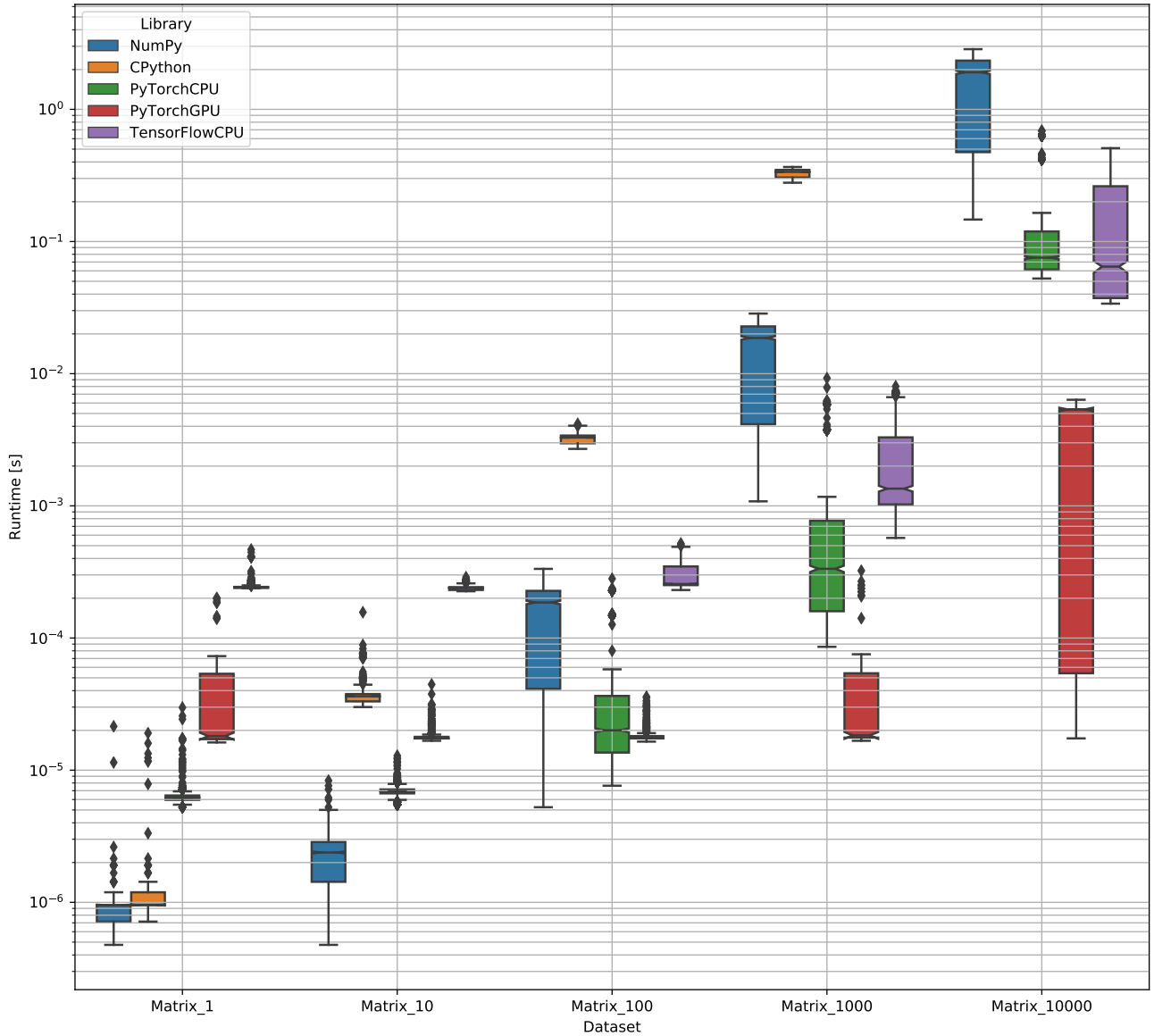
https://www.simonwenkel.com/2020/01/05/python-math-benchmarks.html

Choosing the correct library - Python “basic math” (2)



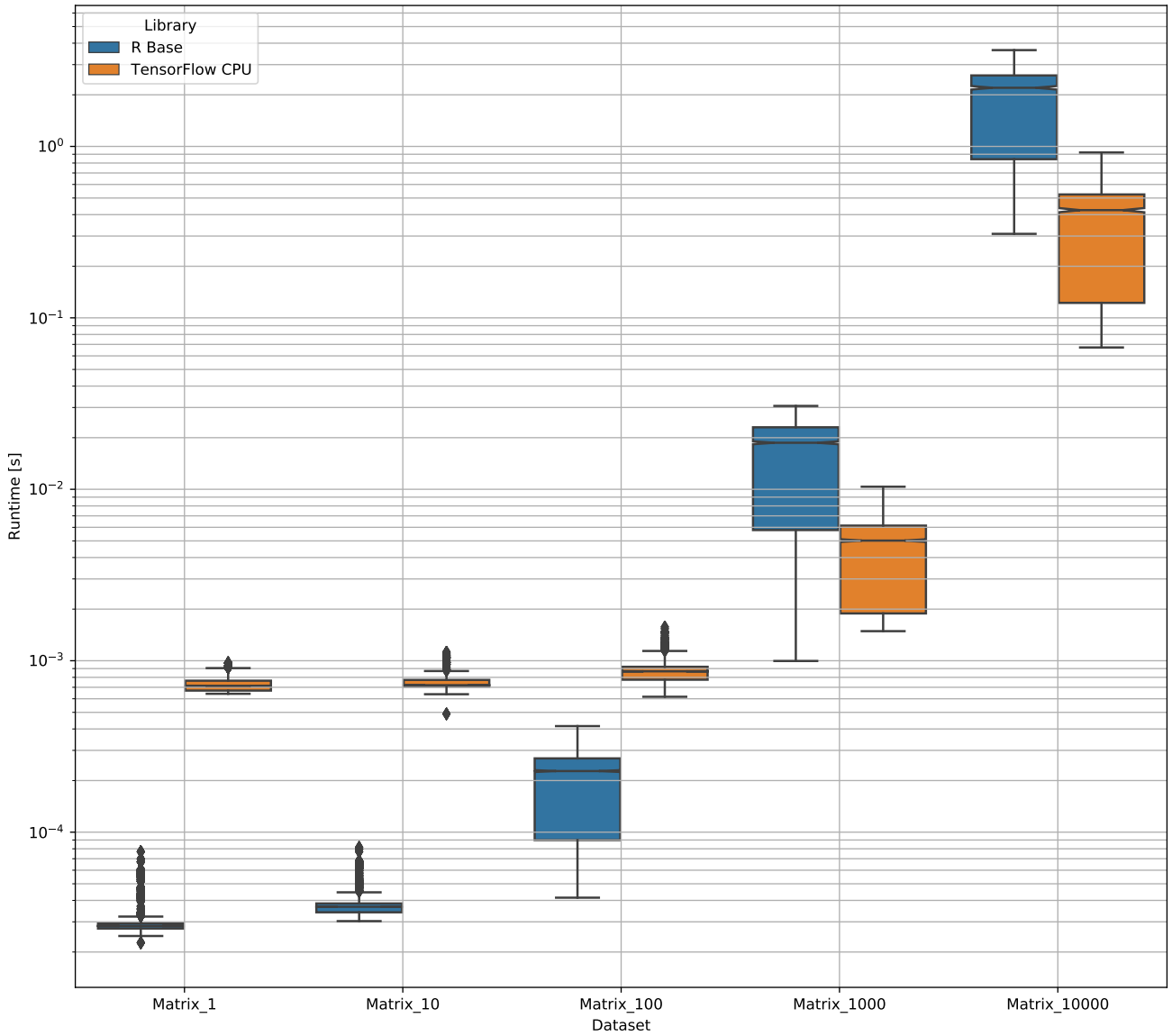
https://www.simonwenkel.com/2020/01/05/python-math-benchmarks.html

Choosing the correct library - Python "basic math" (3)



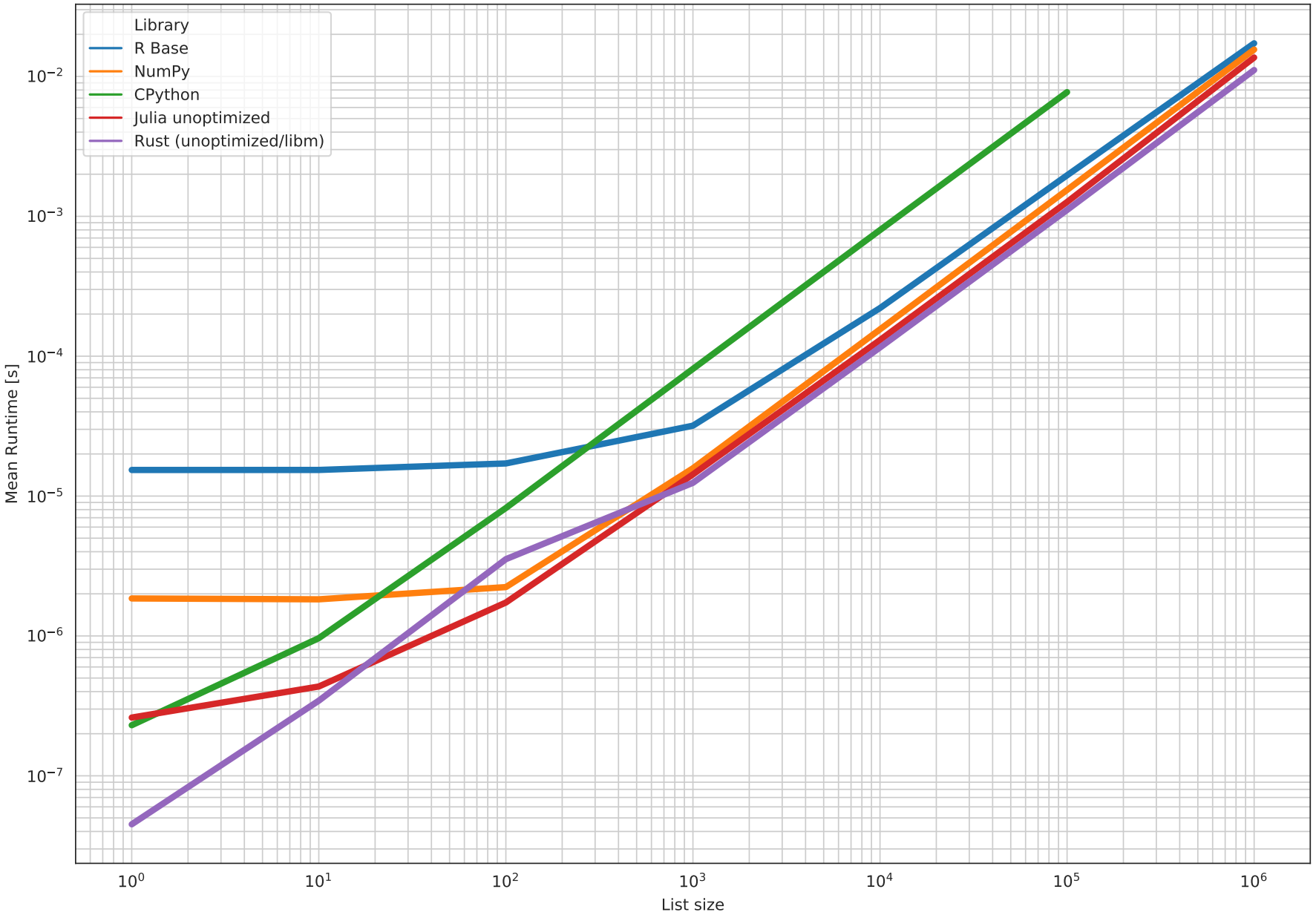
https://www.simonwenkel.com/2020/01/05/python-math-benchmarks.html

Choosing the correct library - R "basic math"

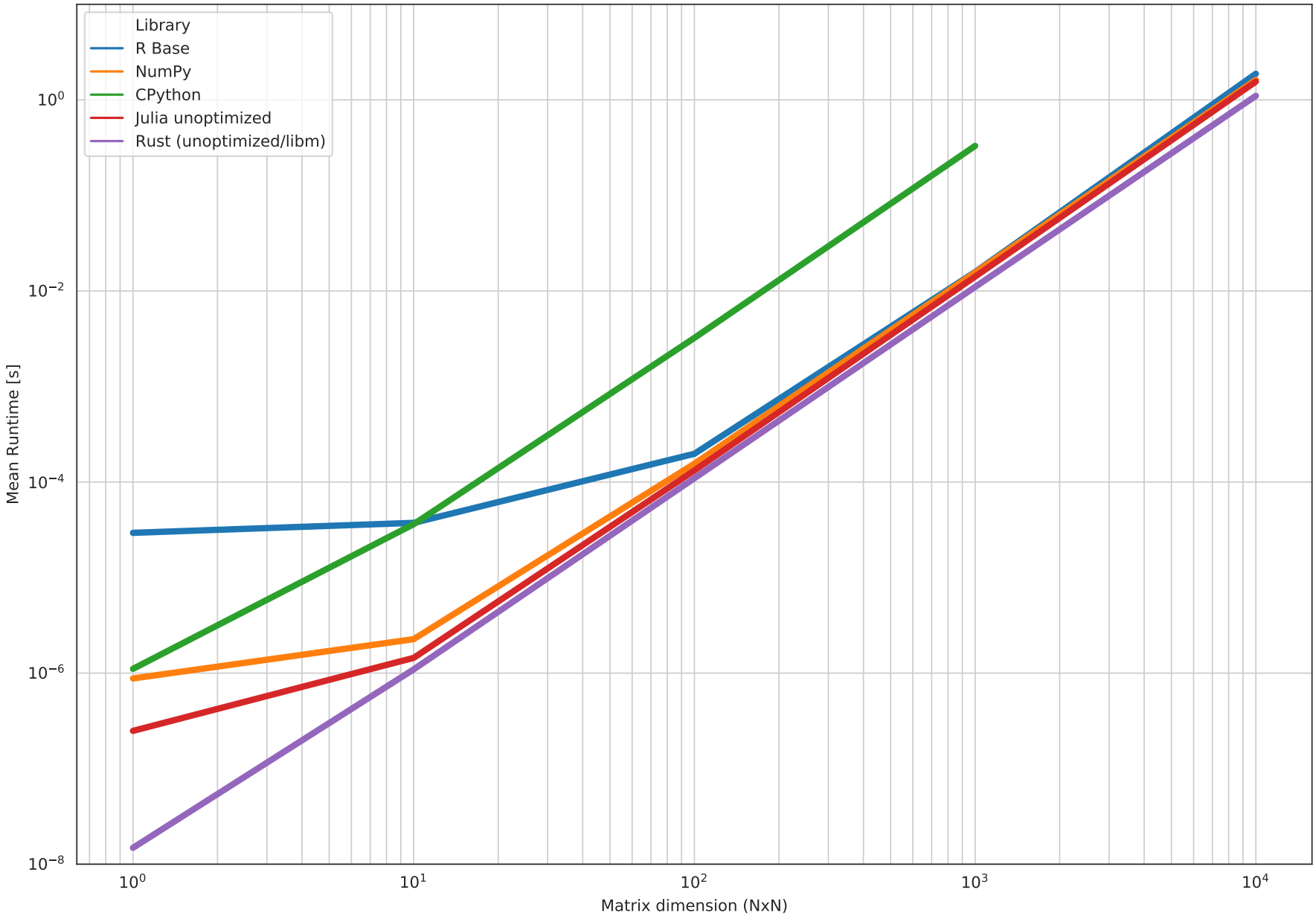


- single thread only
- via py_call?
- no direct C++ API usage?
- GPU seems to work

“Basic math” language comparison (1)



“Basic math” language comparison (2)



Cython - toy example

```python
def add_one(x):
    y = 0
    for i in range(x):
        y+=1
```

```
%%timeit
add_one(100000)
```

```
4.67 ms ± 19 µs per loop (mean ± std. dev. of 7 runs,
```

```
%load_ext Cython
import Cython
```

```
%%cython
cpdef add_one_cython(int x):
    cdef int y = 0
    for i in range(x):
            y+=1
```

```
%%timeit
add_one_cython(100000)
```

```
39.4 ns ± 0.0832 ns per loop (mean ± std. dev. of 7 r
```

```python
def add(x):
    y = 0
    for i in range(x):
        y+=i
```

```
%%timeit
add(10000)
```

```
467 µs ± 4.19 µs per loop (mean ± std. dev. of 7 runs,
```

```
%load_ext Cython
import Cython
```

```
%%cython
cpdef add_cython(int x):
    cdef int y = 0
    cdef int i
    for i in range(x):
            y+=i
```

```
%%timeit
add_cython(10000)
```

```
40.4 ns ± 0.136 ns per loop (mean ± std. dev. of 7 runs
```

Cython - it's not that easy

```
%%cython
cpdef int add_cython(int x):
    cdef int y = 0
    cdef int i
    for i in range(x):
            y+=i
    return y
```

```
%%timeit
add_cython(10000)
```

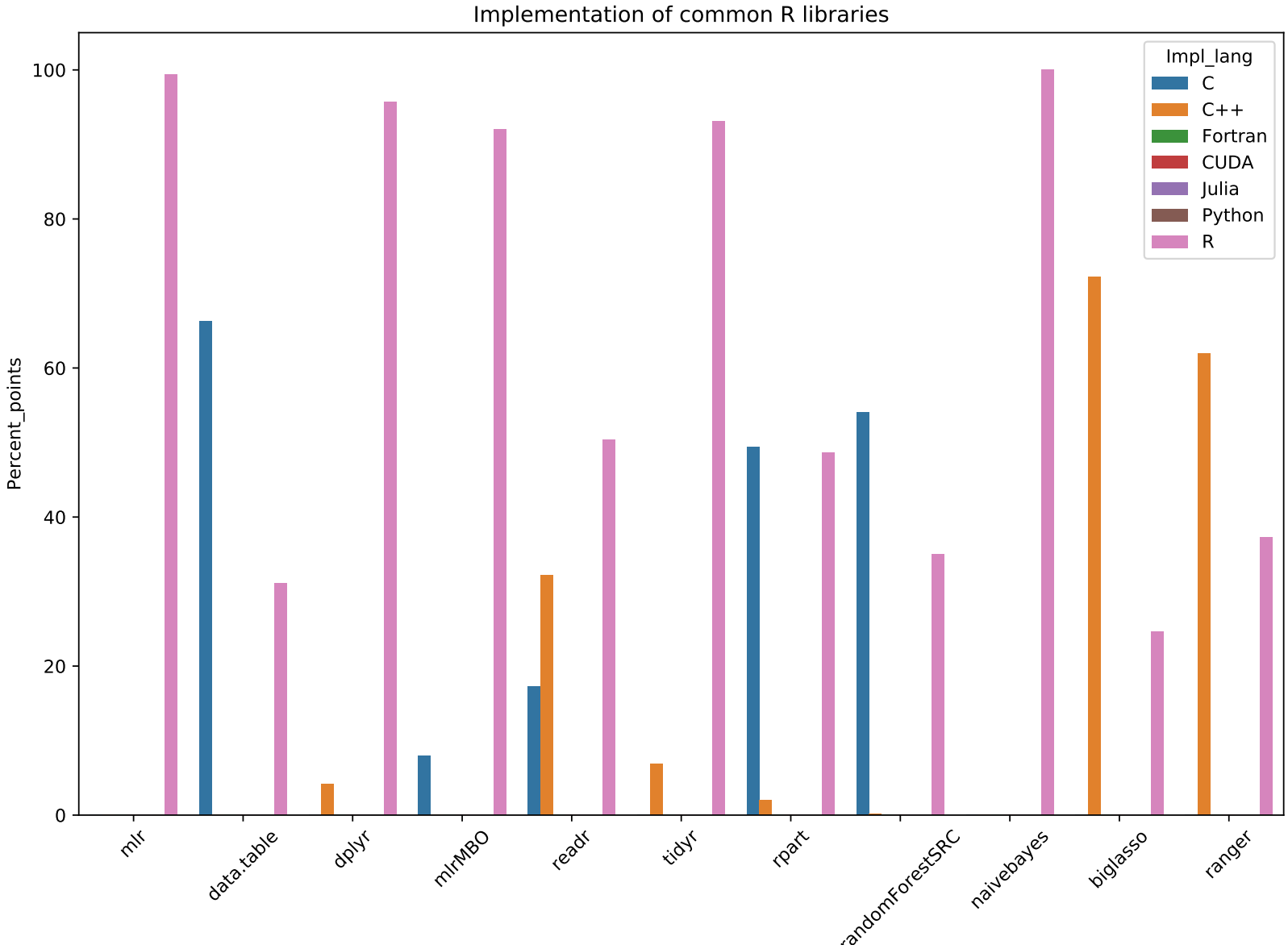
```
1.21 µs ± 10.7 ns per loop (mean ± std. dev. of 7
```

- outside Jupyter notebooks: precompilation with "setup.py"
- high performance: aim at "pure cython"
- usage with NumPy slightly more complicated
- direct integration of C/C++ libraries
- expect to spend a week to learn and understand it

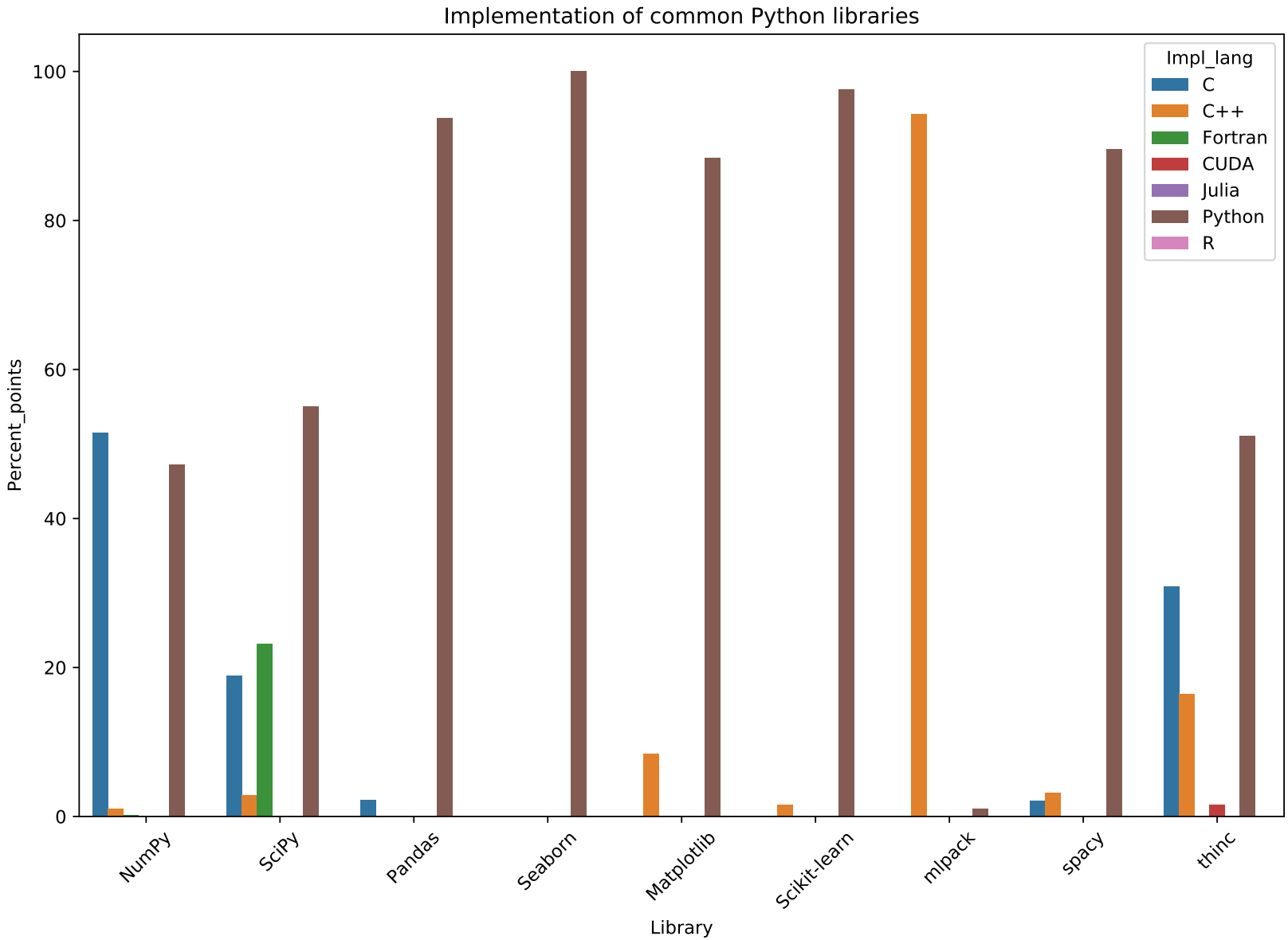
Contents

Are we really using what we think we are using? - Part 2

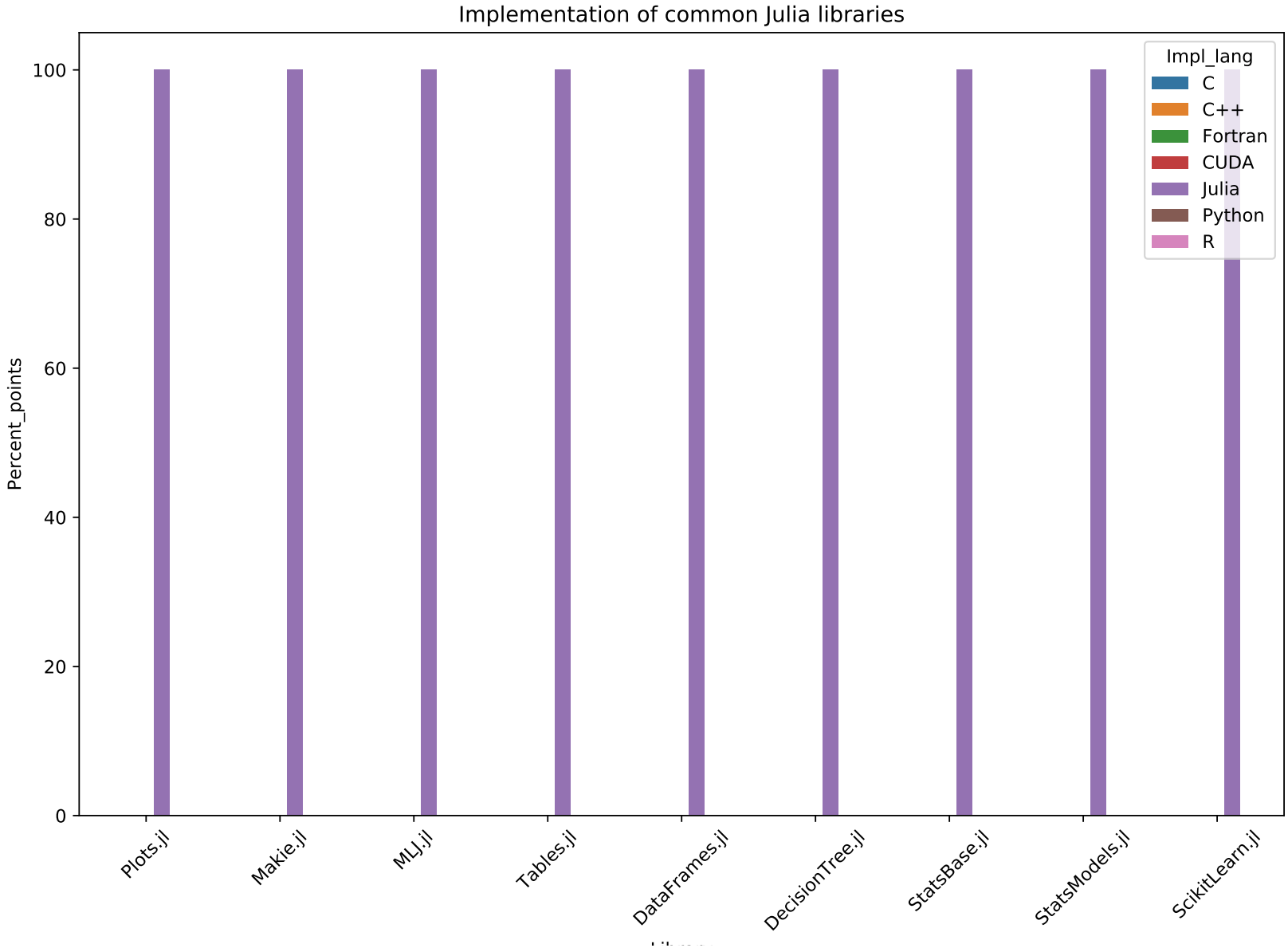
Common R libraries



Common Python libraries



Common Julia libraries



Geospatial/Geostats - R

	C	C++	CUDA	Julia	Python	R
rGDAL	6 %	45 %	0 %	0 %	0 %	36 %
sp	21 %	0 %	0 %	0 %	0 %	21 %
gstat	62 %	1 %	0 %	0 %	0 %	37 %
maptools	41 %	0 %	0 %	0 %	0 %	59 %
geoR	4 %	0 %	0 %	0 %	0 %	96 %

Geospatial/Geostats - Python

Until 1-3 years ago, Python was used (almost) only as a gluetool for various geospatial packages/GIS.

- PostGIS: PostgreSQL and C
- QGIS: written in C++ (+ Python for API)
- GRASS GIS: written in C and C++ (+ Python for API)
- SAGA: written in C++ and C
- ESRI ArcGIS's script environment/API migrated from VBA to Python

Benchmarks

(there are so many benchmarks, and there is so much variance)

general

- Julia up to 400 times faster than R
- Python often 10 times faster than R

machine learning

- Python/Scikit-learn up to 10 times faster than R/caret and with better results
- Python/mlpack is between 5 and 50 times faster than Python/Scikit-learn
- Julia's ML packages are between 2 slower and 400 times faster than Python/Scikit-learn

Performance matters!

Gradient Boosting Libraries

	C	C++	CUDA	Julia	Python	R
CatBoost	0 %	84 %	4 %	0 %	10 %	1 %
LightGBM	6 %	60 %	0? %	0 %	22 %	11 %
XGBoost	0 %	41 %	14 %	0 %	14 %	10 %

Deep Learning Libraries

	C	C++	CUDA	Julia	Python	R
Caffe	0 %	80 %	6 %	0 %	9 %	0 %
Chainer	0 %	10 %	2 %	0 %	76 %	0 %
Darknet	90 %	0 %	8 %	0 %	0 %	0 %
Deeplearning4j	0 %	29 %	4 %	0 %	1 %	0 %
Flux	0 %	0 %	0? %	100 %	0 %	0 %
MXNet	0 %	31 %	4 %	0 %	32 %	0 %
PyTorch	5 %	51 %	8 %	0 %	32 %	0 %
TensorFlow	0 %	61 %	0? %	0 %	31 %	0 %
Theano	5 %	0 %	1 %	0 %	94 %	0 %

Contents

Considerations for production use (1)

General

- what packages are available
- define what you need in terms of performance
- remember infrastructure and development costs
- identify the skill set of team members
- avoid writing packages in C/C++ (safety+security)
- write benchmarks

Considerations for production use (2)

use R

- performance is not important
- many legacy stats packages are needed
- if performance is required: give tensorflow a chance
- if current products are built around it

Considerations for production use (3)

use Python

- unified backend (incl. webservices) is required
- machine learning is a key part (no way around Python(+C/C++/CUDA) yet)
- use PyTorch/TensorFlow or cython for heavy math

Considerations for production use (4)

use Julia

- if strong mathematical optimization across all packages is needed
- clean from scratch implementation is required
- pure performance is required

Some suggestions to the R community

- analysis of R: why is it so slow?
- cython allows to deploy Python in production - anything cython-like for R in development?
- R as gluetool only?
- benchmark packages (e.g. data.table vs DBMS)
- R API for mlpack

References I

[1] URL https://julialang.org/benchmarks/.

[2] Christopher Rackauckas. Why Numba and Cython are not substitutes for Julia, 2018. URL https://www.stochasticlifestyle.com/why-numba-and-cython-are-not-substitutes-for-julia/.

[3] URL https://h2oai.github.io/db-benchmark/.

[4] John Mount. Timing Working With a Row or a Column from a data.frame, 2019. URL http://www.win-vector.com/blog/2019/05/timing-working-with-a-row-or-a-column-from-a-data-frame/.

[5] John Mount. http://www.win-vector.com/blog/2019/06/data-table-is-much-better-than-you-have-been-told/, . URL http://www.win-vector.com/blog/2019/06/data-table-is-much-better-than-you-have-been-told/.

[6] John Mount. New Timings for a Grouped In-Place Aggregation Task, . URL http://www.win-vector.com/blog/2020/01/new-timings-for-a-grouped-in-place-aggregation-task/.

[7] Peter Dalgaard. R 3.4.0 is released, 2017. URL https://stat.ethz.ch/pipermail/r-announce/2017/000612.html.