



CSCI-6221

Julia Group Presentation

Alan(Changjia) Yang
Yvonne(Youwen) Liu
Robert(Bo) Liu
Waad Algorashy
Tiffany Nguyen

What is the Julia language?



- Julia is interactive.
- Julia has a straightforward syntax.
- Julia combines the benefits of dynamic typing and static typing.
- Julia can call Python, C, and Fortran libraries.
- Julia supports metaprogramming.
- Julia has a full-featured debugger.

Why are we interested in Julia?



- Parametric polymorphism
- Multiple dispatch
- Concurrent computing
- Direct calling of C and Fortran libraries
- JIT(Just-in-Time) compilation



Julia vs. Python: Julia's Advantages



- Julia has a math-friendly syntax.
- Julia has automatic memory management.
- Julia offers superior parallelism.
- Julia is developing its own native machine learning libraries.

Julia vs. Python: Python's Advantages



- Python uses zero-based array indexing.
- Python has less startup overhead.
- Python is mature.
- Python has far more third-party packages.
- Python has more users.
- Python is getting faster.

Readability



- Overall Simplicity

Julia is the quite simple language and common language that easy to read and easy to use

- Orthogonality

It given vector normalized and randomly

- Data Types

Abstract Types

Primitive Types (data consists of plain old bits)

Composite Types (records, structs, or objects in various languages)

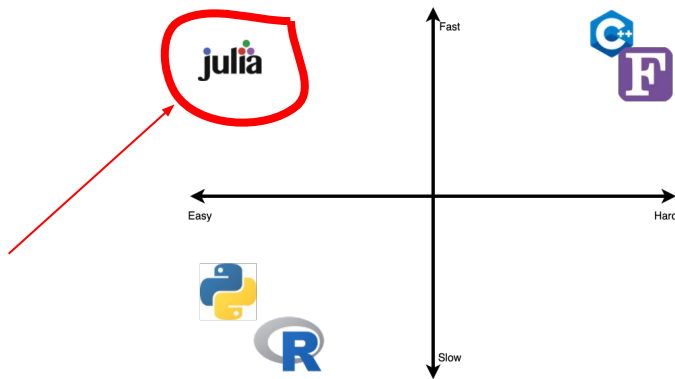
```
abstract type «name» end
abstract type «name» <: «supertype» end
```

- Syntax Design

Julia is an open-source language that combines the interactivity and syntax of 'scripting' languages, such as Python

Writability

If you want to... **code fast and easy** ...use



- Simplicity and Orthogonality

Simplicity, there are no template metaprogramming errors and everyone ... Julia and Go share some sense for the orthogonality of features.

- Support for Abstraction

Abstract types in Julia are structs that have no clear definition. They can be used to allow different types of structs to use the same function.

- Expressivity

Julia is likely to be the most expressive dynamic programming language.

Reliability

- Type Checking

Julia has type checking. But note that Julia is a dynamic language, so it is impossible to actually be 100% sure something is an error.

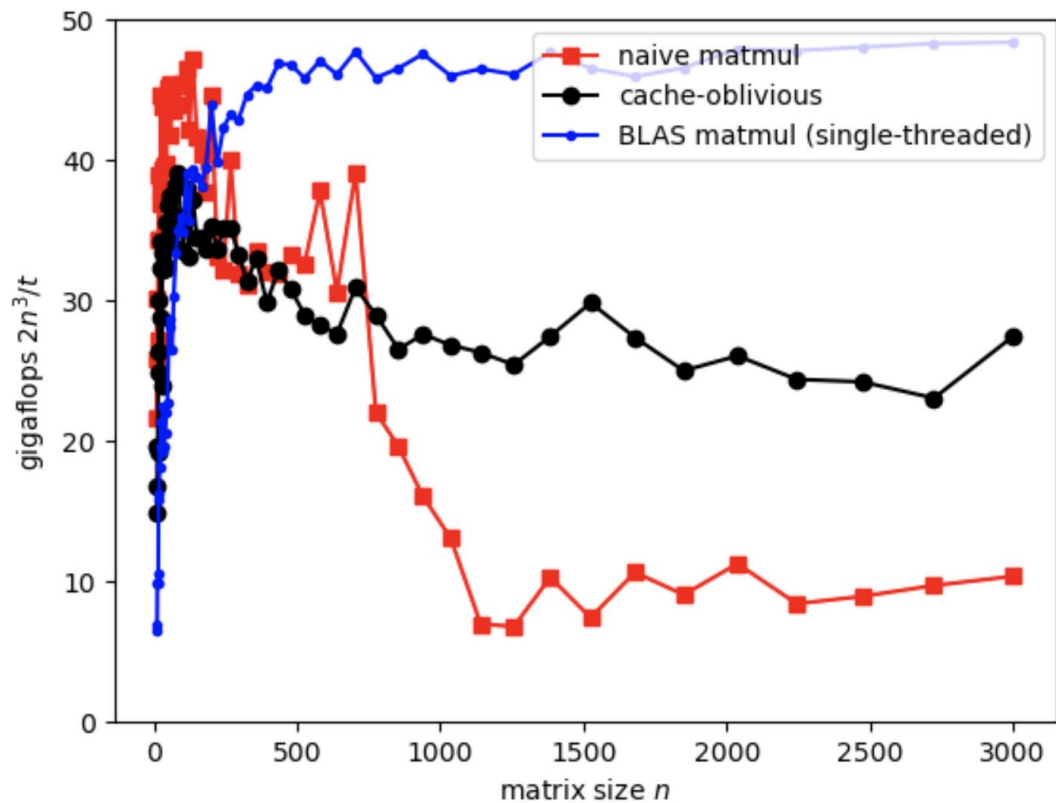
- Exception Handling

Julia allows exception handling through the use of a try-catch block. The block of code that can possibly throw an exception is placed in the try block and the catch block handles the exception thrown.

- Aliasing

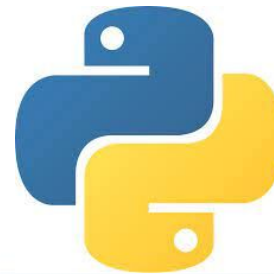
Sometimes it is convenient to introduce a new name for an already expressible type. This can be done with a simple assignment statement.

Cost



Evaluation Criteria

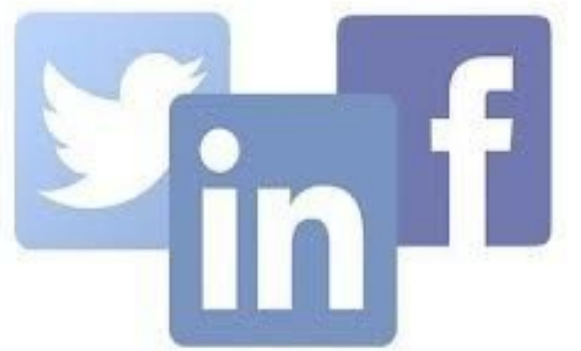
- Community
- Package Ecosystem
- Learning Materials
- Employability
- Programming Paradigms
- Performance
- Concurrency



Community



The Julia community has a presence across multiple platforms such as GitHub
Twitter, LinkedIn, Facebook ..etc.



Community



Measured by:

- # projects actively developed in total on GitHub
- # Conferences in history
- # Hacker News Posts in total
- “Programming, scripting, and markup languages” question of Stack Overflow survey
- “Most Loved” question of Stack Overflow survey
- “Top Paying Technology” question of Stack Overflow survey

Note: The results of “Most Loved” question , “Programming, scripting, and markup languages” question and “Top Paying Technology” question of Stack Overflow survey is based on 2021 survey results. The result of “Top Paying Technology” is median number of salary.

Community



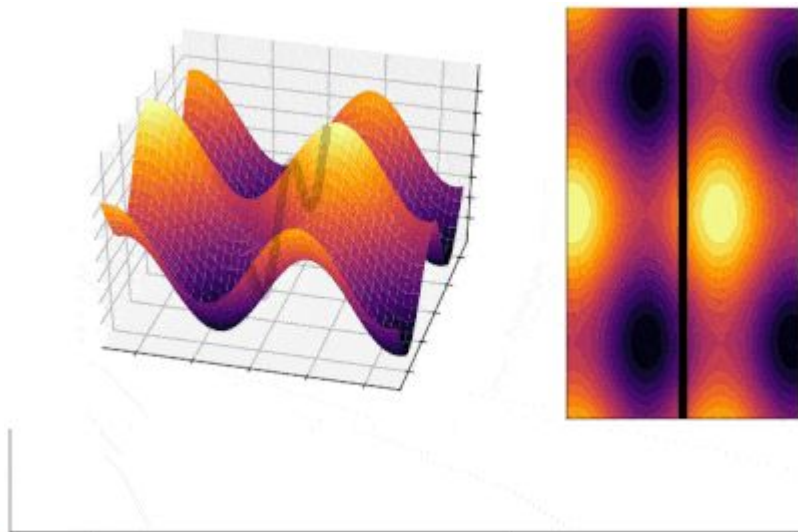
	Java	JavaScript	C++	Python	Julia
GitHub Projects	2,295,514	1,254,871	878,819	2,499,338	38,149
Conferences	2097	22	4	22	5
Hacker News Search	17,162	152,358	11,368	161,253	6,262
Commonly used languages	35.35%	64.96%	24.31%	48.24%	1.29%
StackOverflow "Most Loved" Rating	47.15%	72.73%	49.24%	67.83%	70.69%
StackOverflow "Top Paying Technology"	\$51,888	\$54,049	\$54,049	\$59,454	\$65,228



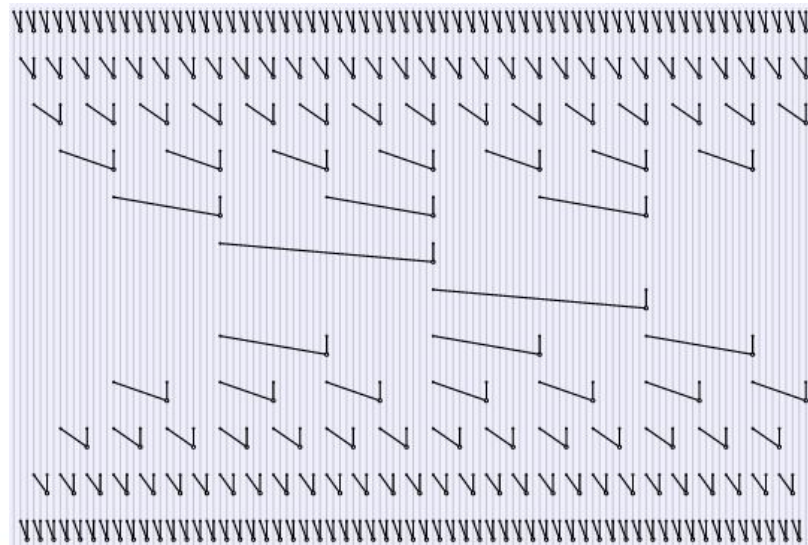
Package Ecosystem



— Visualization:
Data Visualization and Plotting
[Plots.jl](#)



Parallel Computing:
Parallel and Heterogeneous Computing,

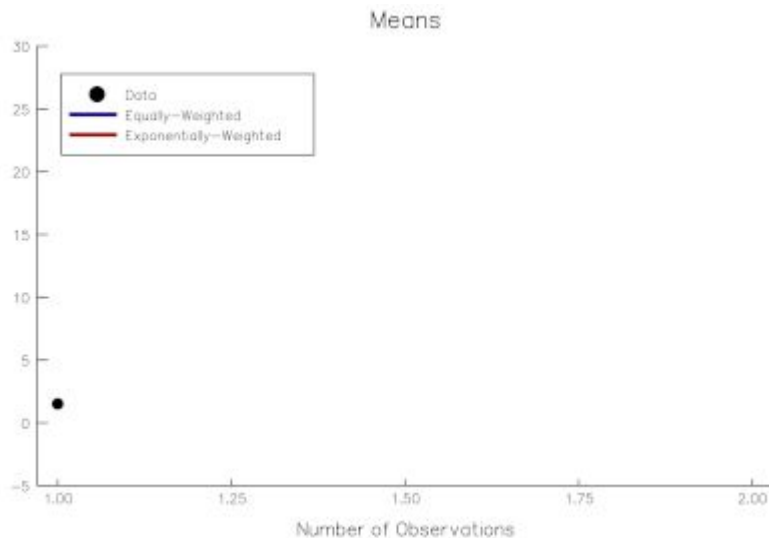


Package Ecosystem



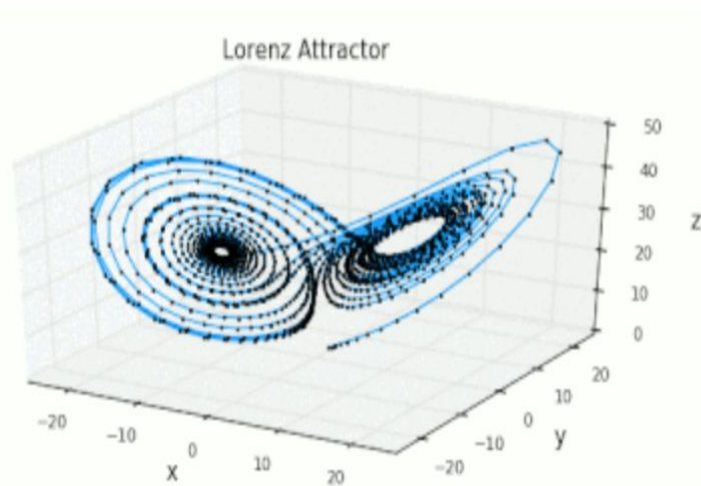
Data science:

Interact with your Data, [DataFrames.jl](#)



Scientific domain:

Scientific Computing, [DifferentialEquations.jl](#)



Package Ecosystem

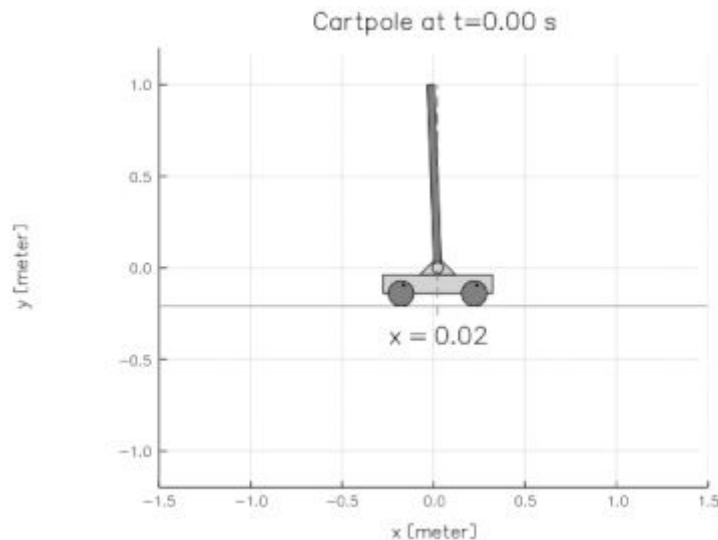


Machine learning :

Scalable Machine Learning ;

MLJ.jl package, which include generalized linear models, decision trees, and clustering.

Flux.jl and Knet.jl packages for Deep Learning.



Package Ecosystem



Measured by:

- Number of packages in the dominant package manager as scraped by modulecounts.com

Package Ecosystem



	Java	JavaScript	C++	Python	Julia
Modules Count	467,604	1,927,358	329	368,258	4,385

Learning Materials



- **Official documents**

Ex: julialang.org

- **Online video tutorials**

Ex: Youtube

- **Online courses:**

Ex: Udemy

- **Online projects:**

Ex: Github, Stackflow

- **Books.**

Learning Materials



	Java	JavaScript	C++	Python	Julia
Amazon Books	10000+	8000+	9000+	10000+	498
Udemy Courses	970	371	355	1,499	19

Note: Amazon book store does not show the specific number over 1000 results

Employability



- **Data Analyst & Data scientist :**

Advanced understanding of a statistical programming language such as R, Python, or Julia.

- **Systems Programmer**

Julia Computing.

Employability



Measured by number of impressions of “xxx developer” on popular job sites located in United States:

- Indeed
- ZipRecruiter
- LinkedIn



BlackRock uses Julia for time-series analysis while British insurer Aviva uses it to calculate risks. Beyond that, the Federal Reserve Bank of New York has previously used Julia in modeling the USA's economy. Other key users of Julia include NVIDIA, CISCO, the Climate Modeling Alliance, Cancer Research UK, QuantEcon, etc



Employability



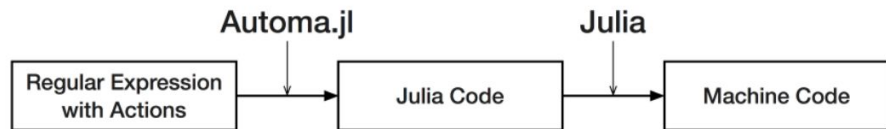
	Java	JavaScript	C++	Python	Julia
Indeed	103,526	90,924	25,990	82,648	110
ZipRecruiter	201,640	145,415	340,675	296,068	2,423
LinkedIn	870,000	809,000	782,000	902,000	1,000

Note: Statistics collected on April 8, 2022

Programming Paradigms 1

Packages and resources that support various programming styles, Software Architecture and CS paradigms.

A Julia package for text validation, parsing, and tokenizing based on state machine compiler.



Examples:

- **Automa.jl :: A julia code generator for regular expressions - this package can do text validation, parsing, and tokenizing based on a state machine compiler.**

A tokenizer of octal, decimal, hexadecimal and floating point numbers

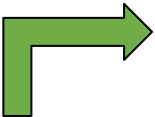
```
import Automa
import Automa.RegExp: @re_str
const re = Automa.RegExp

# Describe patterns in regular expression.
oct      = re"0o[0-7]+"
dec      = re"[-+]?[0-9]+"
hex      = re"0x[0-9A-Fa-f]+"
prefloat = re"[-+]?([0-9]+\.[0-9]*|[0-9]*\.[0-9]+)"
float    = prefloat | re.cat(prefloat | re"[-+]?[0-9]+", re"[eE]([-+]?[0-9]+)")
number   = oct | dec | hex | float
numbers  = re.cat(re.opt(number), re.rep(re" + " * number), re" *")
```

- Automata
 - Control Flow
 - Declarative Programming
 - Functional Programming
 - DSL
 - Grammatical Evolution
 - Interpreters
 - Language Comparison
 - Macro
 - Metaprogramming
 - Automatic Programming
 - Multi Threading
 - Polymorphism
 - Double Dispatch
 - Multiple Dispatch
 - Program Analysis
 - Reactive Programming
 - STATIC ANALYSIS
 - Turnaround Time
 - Style Guidelines

Programming Paradigms 2

Julia uses multiple dispatch as a paradigm, making it easy to express many object-oriented and functional programming patterns.



```
# Use package and import desired positive/negative trait type aliases
using BinaryTraits
using BinaryTraits.Prefix: Can

# Define a trait and its interface contracts
@trait Fly
@implement Can{Fly} by fly(_, destination::Location, speed::Float64)


# Define your data type and implementation
struct Bird end
fly(::Bird, destination::Location, speed::Float64) = "Wohoo! Arrived! 🐦"

# Assign your data type to a trait
@assign Bird with Can{Fly}

# Verify that your implementation is correct
@check(Bird)

# Dispatch for all flying things
@traitfn flap(::Can{Fly}, freq::Float64) = "Flapping wings at $freq Hz"
```

Examples:

- **BinaryTraits.jl** :: easy-to-use trait library with formal interface specification support.
 - **WhereTraits.jl** :: This package exports one powerful macro `@traits` with which you can extend Julia's where syntax.
- 

- dispatch on functions returning Bool

```
@traits f(a) where {isodd(a)} = (a+1)/2
@traits f(a) where {!isodd(a)} = a/2
f(4) # 2.0
f(5) # 3.0
```

- dispatch on functions returning anything

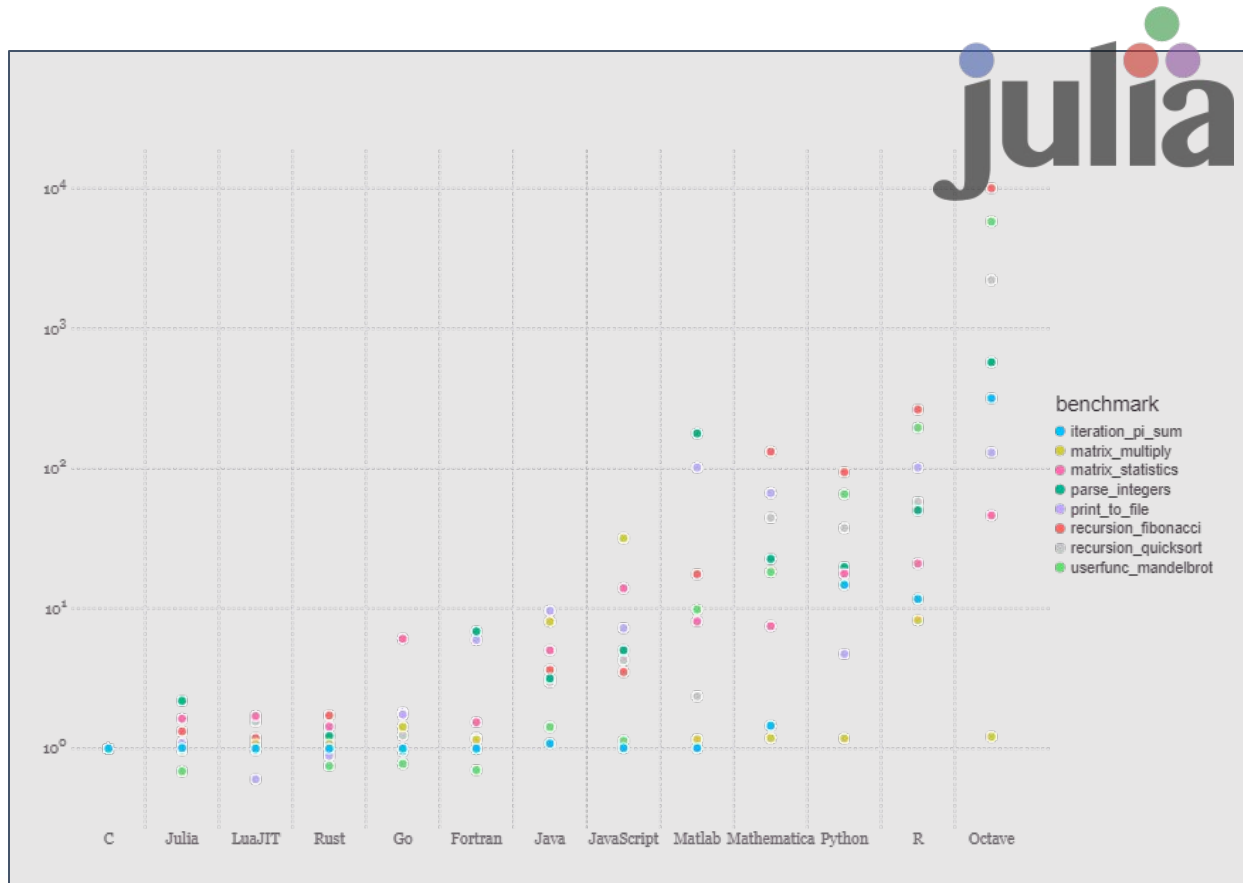
```
@traits g(a) where {Base.IteratorSize(a)::Base.HasShape} = 43
@traits g(a) = 1
g([1,2,3]) # 43
g(Iterators.repeated(1)) # 1
```

- dispatch on bounds on functions returning Types

```
@traits h(a) where {eltype(a) <: Number} = true
@traits h(a) = false
h([1.0]) # true
h([""]) # false
```

Performance

Julia Micro-Benchmarks



Performance

- Julia: using `DataFrames.jl` - 0.4ms
- Python: using `Pandas` and `NumPy` - 1.76ms
- R: using `{dplyr}` - 3.22ms

```
import pandas as pd
import numpy as np
```

```
n = 10000
```

```
df = pd.DataFrame({'x': np.random.choice(['A', 'B', 'C', 'D'], n, replace=True),
                  'y': np.random.randn(n),
                  'z': np.random.rand(n)})
```

```
%timeit df.groupby('x').agg({'y': 'median', 'z': 'mean'})
```



```
library(dplyr)
```

```
n <- 10e3
df <- tibble(
  x = sample(c("A", "B", "C", "D"), n, replace = TRUE),
  y = runif(n),
  z = rnorm(n)
)
```

```
bench::mark(
  df %>%
    group_by(x) %>%
    summarize(
      median(y),
      mean(z)
    )
)
```

```
using Random, StatsBase, DataFrames, BenchmarkTools, Chain
Random.seed!(123)
```

```
n = 10_000
```

```
df = DataFrame(
  x=sample(["A", "B", "C", "D"], n, replace=true),
  y=rand(n),
  z=randn(n),
)
```

```
@btime @chain $df begin # passing `df` as reference so the compiler cannot optimize
  groupby(:x)
  combine(:y => median, :z => mean)
end
```



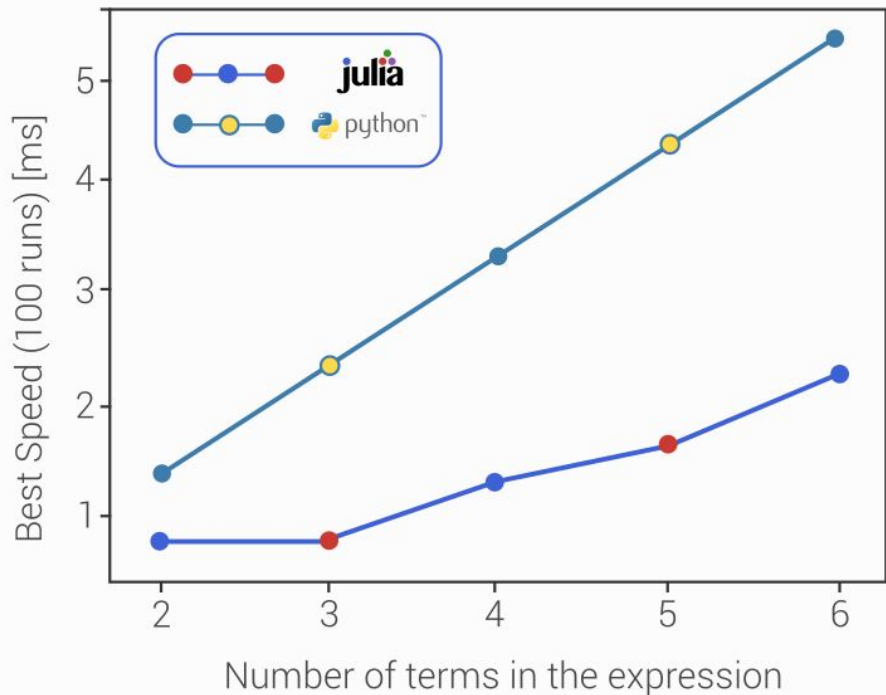
Performance



Language	Ease/readability	Lines of code	ST time	MT time
Numpy	Very good	15	6.8s	X
Basic Julia	Excellent	21	3.0s	0.6s
F2PY-Fortran	Very good	42	4.8s	1.3s

Performance

Projects from other languages can be written once and naively compiled in Julia making it ideal for machine learning and data science. The time taken by Julia to execute **big** and **complex** codes is lesser to Python's



Concurrency



- Concurrency means the ability for a program to be decomposed into parts that can be run independently.
- Julia supports a variety of styles of concurrent computation.

A multithreaded computation(simultaneous work)

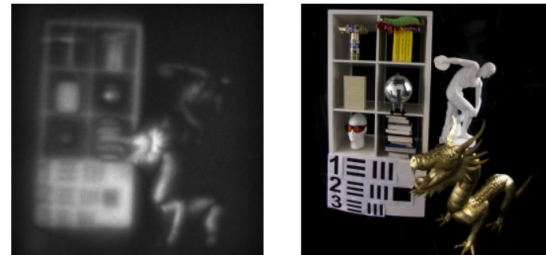
Distributed computing

GPU computing

Objective of Project



- Ultra high performance image processing system on backend with Julia:
 - image blurring
 - image sharpening
 - gradient computation
 - linear filtering operations
 - nonlinear filters like min/max
 - fast f-k migration algorithm



```
julia> tau, calib = loadDataset("../teaser", "meas_10min.mat") ;  
julia> calibrate!(tau, calib) ;  
  
julia> tau = downsampleAndCrop(tau, 64, 512) ;  
  
julia> tau = reconstruct(tau) ;
```


Software and Hardware Requirements



Software:

- VSCode / Juno
- Jupyter notebook

Hardware:

- Personal Computer



Code Snapshots



```
In [5]: function loadImgOnline(url)
        r = HTTP.get(url)
        buffer = IOBuffer(r.body)
        img = ImageMagick.load(buffer)
        img
    end
```

Out[5]: loadImgOnline (generic function with 1 method)

```
In [6]: function saveImg(type, path, img_source)
        if type == 1 #jpg
            # save file in JPG format
            save(string(path, "/saved_pic.jpg"), img_source)
        elseif type == 2 #png
            # save file in PNG format
            save(string(path, "/saved_pic.png"), img_source)
        else
            return "not supported file type"
        end
        return "file saved"
    end
```

Out[6]: saveImg (generic function with 1 method)

```
In [9]: function sharpImg(img)
        gaussian_smoothing = 1
        intensity = 1
        # Load an image and apply Gaussian smoothing filter
        imgb = imfilter(img, Kernel.gaussian(gaussian_smoothing))
        # convert images to Float to perform mathematical operations
        img_array = Float16.(channelview(img))
        imgb_array = Float16.(channelview(imgb))
        # create a sharpened version of our image and fix values from 0 to 1
        sharpened = img_array .* (1 + intensity) .+ imgb_array .* (-intensity)
        sharpened = max.(sharpened, 0)
        sharpened = min.(sharpened, 1)
        sharpened_image = colorview(RGB, sharpened)
        sharpened_image
    end
```

Out[9]: sharpImg (generic function with 1 method)

```
In [10]: #controlled by Red, Green, Blue
        function imgSaturate(img_source, r_par, g_par, b_par)
            img = copy(img_source)
            img_ch_view = channelview(img) # extract channels
            img_ch_view = permuteddimsview(img_ch_view, (2, 3, 1))
            x_coords = 1:size(img, 2)

            img_ch_view[:, x_coords, 1] = min.(img_ch_view[:, x_coords, 1] .* r_par, 1)
            img_ch_view[:, x_coords, 2] = min.(img_ch_view[:, x_coords, 2] .* g_par, 1)
            img_ch_view[:, x_coords, 3] = min.(img_ch_view[:, x_coords, 3] .* b_par, 1)
            img
        end
```

Out[10]: imgSaturate (generic function with 1 method)

Constrains



- Unlike some mainstream language such as Java or Python, Julia has a relatively limited libraries, which limited the writability, functionality and scalability of this language. In another word, It's hard to use Julia to construct the front end of our software. This led us to consider other languages for the frond end.
- Since julia is rather a young language and is mainly used by data science and machine learning. Some supporting packages may not have been tested too much in the industry, and there may be stability and compatibility issues.

LIVE DEMO



We will run everything through Jupyter Notebook.



Any Questions?

More details can be found through our Github link:

<https://github.com/yliu1268/CSCI6221-Group-Project-Julia>



Thank you for your time!

THE GEORGE
WASHINGTON
UNIVERSITY

WASHINGTON, DC