Julia and optimization: Part 1

15.S60: Computing in Optimization and Statistics

Session #6: Tuesday, January 22

Goals of this session

- Share and take away some tips and tricks for coding in Julia
- Build familiarity with Julia's capabilities in optimization and methods for addressing scalability issues
- Learn how to manage data and structure optimization code bases in Julia

Agenda for today

Part 1: Julia intro + tips/tricks

- Why do we use Julia?
- Issues with Julia
- Tips and tricks

Part 2: Intro to JuMP

- Coding LPs in Julia
- Data management
- Coding best practices

Part 3: Integer optimization

- Branch-and-bound
- Interpreting Gurobi
- Speeding up the solver

Part 1: Introduction to Julia + tips and tricks

History

- Julia was founded at MIT by 4 co-founders:


Stefan Karpinski


Alan Edelman


Jeff Bezanson


Viral Shah

- First developed in 2009, with the first stable release in 2012
- Goals were to combine the benefits and iron out the disadvantages of each programming language into one open- sourced, liberally-licensed language

Why Julia?

High-level and easy to use

Julia is fast!

Good for optimization

Why Julia?

High-level and easy to use

Julia is fast!

Good for optimization

High-level and easy to use

- High-level, with readable, math-like syntax similar to Python/Matlab
- Possible to do complex computations quickly, e.g., matrix multiplication:

```
A = [1 2 3
     2 1 2
     3 2 1]

b = [1,1,1]
A \ b
```

```
3-element Vector{Float64}:
 0.24999999999999997
 8.326672684688673e-17
 0.25
```

- Julia is also interactive, which makes it possible to run Julia in Jupyter notebooks or in the REPL

Why Julia?

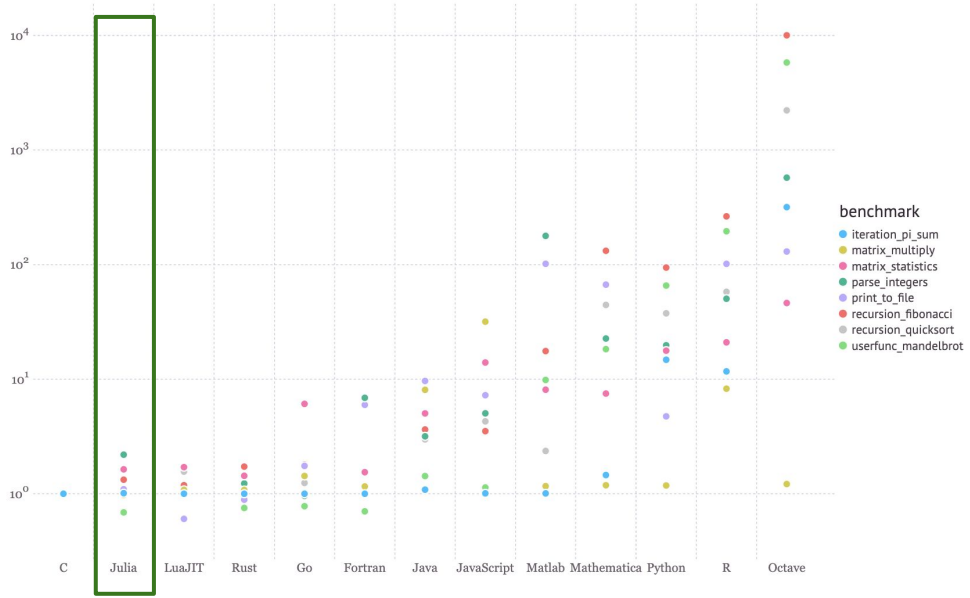
High-level and easy to use

Julia is fast!

Good for optimization

Julia is fast!

- Julia is fast thanks to:
  - Multiple dispatch
  - Just-in-time compilation
  - No two-language problem



Julia micro-benchmarks test performance

Multiple dispatch

- **Dispatch:** which “version” of a function (i.e., method) to execute when the function is called

Multiple dispatch

- **Dispatch:** which “version” of a function (i.e., method) to execute when the function is called
- **Single dispatch:** for other high-level languages (e.g. Python), method execution is based on the type of one argument

Multiple dispatch

- **Dispatch:** which “version” of a function (i.e., method) to execute when the function is called
- **Single dispatch:** for other high-level languages (e.g. Python), method execution is based on the type of one argument
- **Multiple dispatch**: for Julia, method execution is based on the types of all inputs
  - Julia creates methods that are specialized to the exact argument types
  - Leads to extensible code, where you can add new types to a function without modifying existing code

Multiple dispatch

2 methods for a generic function **my_function:**

```
function my_function(x)
    println("Default output")
end
```

```
function my_function(x::Int) # only called when x is an integer
    println("You gave me an integer!")
end
```

What will: **my_function(1.0)** return?

Multiple dispatch

2 methods for a generic function **my_function:**

```
function my_function(x)
    println("Default output")
end
```

```
function my_function(x::Int) # only called when x is an integer
    println("You gave me an integer!")
end
```

What will: **my_function(1.0)** return?

An error?

“Default output”

“You gave me an integer!”

Multiple dispatch

2 methods for a generic function **my_function**

```
function my_function(x)
    println("Default output")
end
```

```
function my_function(x::Int) # only called when x is an integer
    println("You gave me an integer!")
end
```

What will: **my_function(1)** return?

Multiple dispatch

2 methods for a generic function **my_function:**

```
function my_function(x)
    println("Default output")
end
```

```
function my_function(x::Int) # only called when x is an integer
    println("You gave me an integer!")
end
```

What will: **my_function(1)** return?

What about: **my_function(“ORC”)**?

Just-in-time compilation

- Multiple dispatch: when functions are called with specific argument types, Julia creates specialized methods for those types
- **Just-in-time compilation** is when code is compiled into machine code at runtime, rather than ahead of time
- These methods are cached, so future calls with the same argument types reuse the compiled code and will run fast

No two-language problem

- **Two language problem** is the workflow where two different programming languages are used for the same task:
  - A high-level language (e.g. Python, Matlab, R) for prototyping and quick development
  - A low-level language (e.g., C, C++) for performance-sensitive code

No two-language problem

- **Two language problem** is the workflow where two different programming languages are used for the same task:
  - A high-level language (e.g. Python, Matlab, R) for prototyping and quick development
  - A low-level language (e.g., C, C++) for performance-sensitive code
- This requires you to transfer / re-write code in a faster language if performance is an issue
  - Leads to more implementation, more complex codebases, etc.

No two-language problem

- **Two language problem** is the workflow where two different programming languages are used for the same task:
  - A high-level language (e.g. Python, Matlab, R) for prototyping and quick development
  - A low-level language (e.g., C, C++) for performance-sensitive code
- This requires you to transfer / re-write code in a faster language if performance is an issue
  - Leads to more implementation, more complex codebases, etc.
- In Julia, high-level code compiles to efficient machine code
  - You can use Julia for prototyping, and *stay in Julia* while optimizing it for fast performance
  - More readable code, faster iterations, and fewer bugs

Why Julia?

High-level and easy to use

Julia is fast!

Good for optimization

Performance in optimization

- Optimization in Julia is built around **JuMP**, an easy-to-read language that is similar to mathematical notation

Performance in optimization

- Optimization in Julia is built around **JuMP**, an easy-to-read language that is similar to mathematical notation
  - JuMP is an API that allows you to build and solve optimization problems with many different solvers, such as Gurobi, CPLEX, Mosek, and GLPK
  - Define all your constraints, variables, and objectives through JuMP syntax, which enables flexibility

```
@variable(model, x >= 0)
@constraint(model, A * x == b)
@objective(model, Min, c' * x)
```

Performance in optimization

- Optimization in Julia is built around **JuMP**, an easy-to-read language that is similar to mathematical notation
  - JuMP is an API that allows you to build and solve optimization problems with many different solvers, such as Gurobi, CPLEX, Mosek, and GLPK
  - Define all your constraints, variables, and objectives through JuMP syntax, which enables flexibility

```
@variable(model, x >= 0)
@constraint(model, A * x == b)
@objective(model, Min, c' * x)
```

- Because of the no two-language problem, we don't need to re-write our code in low-level languages for performance improvements

Issues with Julia

Debugging can be tricky

Slow startup time

Immature package system

Issues with Julia

Debugging can be tricky

Slow startup time

Immature package system

Debugging

- Improvements in syntax error messages since Julia 1.10:

  Unclear where the error is :/

```
julia> [[], [], [,] [], []]
ERROR: syntax: unexpected ","
```

  Pinpoints where the error occurred :)

```
julia> [[], [], [,] [], []]
ERROR: ParseError:
# Error @ REPL[1]:1:11
[[], [], [,] [], []]
#          └ ── unexpected `,`
```

Debugging

- Improvements in syntax error messages since Julia 1.10:

  Unclear where the error is :/

```
julia> [[], [], [,] [], []]
ERROR: syntax: unexpected ","
```

  Pinpoints where the error occurred :)

```
julia> [[], [], [,] [], []]
ERROR: ParseError:
# Error @ REPL[1]:1:11
[[], [], [,] [], []]
#          └ ── unexpected `,`
```

- Improvements in the stack trace since Julia 1.10 to make it more concise

  Very verbose and hard to read :/

```
Stacktrace:
 [1] error()
   @ Base ./error.jl:44
 [2] f(g::Function, a::Int64; kw::Base.Pairs{Symbol, Union{}, Tuple{}, NamedTuple{(), Tuple{}}})
   @ Main ./REPL[1]:1
 [3] f(g::Function, a::Int64)
   @ Main ./REPL[1]:1
 [4] ##f#16
   @ ./REPL[2]:1 [inlined]
 [5] f(a::Int64)
   @ Main ./REPL[2]:1
 [6] top-level scope
   @ REPL[6]:1
```

  More concise :)

```
Stacktrace:
 [1] error()
   @ Base ./error.jl:44
 [2] f(g::Function, a::Int64; kw::@Kwargs{})
   @ Main ./REPL[1]:1
 [3] f(a::Int64)
   @ Main ./REPL[2]:1
 [4] top-level scope
   @ REPL[3]:1
```

Debugging

- Unfortunately, even in more recent versions of Julia, error messages can still be ambiguous:

```
f(x::Int, y::Int) = x + y
f(x::Int, y::String) = parse(Int, y) + x

f(3, missing)
```

```
MethodError: no method matching f(::Int64, ::Missing)
The function `f` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  f(::Int64, ::String)
   @ Main In[2]:2
  f(::Int64, ::Int64)
   @ Main In[2]:1


Stacktrace:
 [1] top-level scope
   @ In[2]:4
```

- Important to test code frequently to catch bugs!

Issues with Julia

Debugging can be tricky

Slow startup time

Immature package system

Slow startup time

- **Packages**: compiled when using them, so it might take a really long time to set up your working environment
  - Known as the “Time to first ___” problem (TTFX)
  - Julia versions 1.10 and onwards have sped up the performance of loading packages

Slow startup time

- **Packages**: compiled when using them, so it might take a really long time to set up your working environment
  - Known as the “Time to first ___” problem (TTFX)
  - Julia versions 1.10 and onwards have sped up the performance of loading packages
- **Functions**: because compilation happens at runtime (just-in-time compilation), Julia can be slower on the first call to the function

Slow startup time

- **Packages**: compiled when using them, so it might take a really long time to set up your working environment
  - Known as the “Time to first ___” problem (TTFX)
  - Julia versions 1.10 and onwards have sped up the performance of loading packages
- **Functions**: because compilation happens at runtime (just-in-time compilation), Julia can be slower on the first call to the function
- When benchmarking code, it’s important to ignore compilation time for packages and functions

Issues with Julia

Debugging can be tricky

Slow startup time

Immature package system

Immature package system

- “Mega packages” in Python like `numpy` and `pandas` don’t exist to the same scale in Julia
- However, Julia is a new and fast-growing community, and packages are being created at a fast pace

Immature package system

- “Mega packages” in Python like `numpy` and `pandas` don’t exist to the same scale in Julia
- However, Julia is a new and fast-growing community, and packages are being created at a fast pace
- Some comprehensive packages for scientific computing:
  - Statistics: `StatsBase.jl` and `Statistics.jl`
  - Machine learning: `MLJ.jl`, `Flux.jl`, and `Knet.jl` for deep learning
  - Data tools: `DataFrames.jl`, `CSV.jl`, `Arrow.jl`, and `Spark.jl` for big data
  - Data visualization: `Plots.jl`, `Makie.jl`
  - Optimization: `JuMP.jl` and `Optim.jl`
  - Differential equations: `DifferentialEquations.jl`

Julia tips and tricks

Type stability

- A function is **type stable** if the compiler can infer the type of its return value from the types of its input arguments

Type stability

- A function is **type stable** if the compiler can infer the type of its return value from the types of its input arguments
- If a function is type stable…
  - Julia can create efficient machine code
- If a function is not type stable…
  - Compiler will produce machine code full of “ifs,” covering all options of variable types
  - This slows down machine code :/

Type stability

- `Number` is an abstract type which contains many concrete types, such as `Int8` and `Float64`
- Is this function type stable?

```
function add(x::Number, y::Number)
    z = x + y
    return z
end
```

Type stability

- `Number` is an abstract type which contains many concrete types, such as `Int8` and `Float64`
- Is this function type stable?

```
function add(x::Number, y::Number)
    z = x + y
    return z
end
```

- What about this function?

```
function largest(a::Float64, b::Int64)
    if a > b
        c = a
    else
        c = b
    end
    return c
end
```

Type stability

- Make functions type stable:

```
function largest_stable(a::Float64, b::Int64)
    b = b*1.0
    if a > b
        c = a
    else
        c = b
    end
    return c
end
```

Cast `b` to a float

```
function largest_stable_new(a::Number, b::Number)
    a, b = promote(a, b)
    if a > b
        return a
    else
        return b
    end
end
```

Use `promote`

- Useful to check the type stability of functions (especially for more complex ones) using `@code_warntype <your_function>`

Broadcasting

- **Broadcasting** is a way to apply operations element-wise over arrays or other objects using the . (dot) syntax

```
x = [1, 2, 3]
y = [4, 5, 6]
x .+ y          # elementwise addition
x .* y          # elementwise multiplication
```

- Can also be applied to functions, avoiding the need for loops

```
y = [1,2,3]
g(x) = x^2 + 1
g.(y)
```

- Benefits are in readability, as performance is the same as a `for` loop

Dictionaries

- Dictionaries are a type of data structure for when you need fast lookups of *values* using *keys*
  - Order is not preserved, unless using an ordered dictionary explicitly
- When type-specified, dictionaries are fast, for example:

```
typedict = Dict{String, Int}()
```

- Iteration is also fast, and can be done by:

```
for (k, v) in typedict
    …
end
```

- Useful operations: `haskey` (to check existence of keys), `get` (to safely access values), `delete!` (to delete entries)

Data structures

- A `struct` is similar to a Python class, which is a user-defined data structure to store related data with concrete types
  - Immutable structs: cannot change the values of fields after creating an instance of the struct
  - Mutable structs: field can change after creation of instance, but this is less memory efficient

Data structures

- A `struct` is similar to a Python class, which is a user-defined data structure to store related data with concrete types
  - Immutable structs: cannot change the values of fields after creating an instance of the struct
  - Mutable structs: field can change after creation of instance, but this is less memory efficient
- Naturally results in type stable code, which leads to efficient machine code and optimized loops
- Useful in large-scale data-oriented projects, where you need to manage different kinds of data
- We will see this in action in Part 2!

Any other tips and tricks from you all?