# Introduction to Julia

Toivo Henningsson

April 11, 2023

# About me

- Involved in the Julia community for the first few years, been using it ever since
- Wrote the first Julia debugger in 2012
- Currently using Julia to model/explore hardware acceleration of Simultaneous Localization and Mapping (SLAM) algorithms at Ericsson

# Overview

Julia:

- Dynamic language
- Aimed at technical computing (but can do many things!)
- Expressive
- Can be fast

# Initial example

Let's define a simple function:

```
f(x, y) = y / (y + exp(-2x))
```

What can we do with it?

Evaluate:

```
julia> f(1, 2)
0.9366210616669624
```

Evaluate with complex arguments:

```
julia> f(1 + 2im, 2 - 1im)
1.058531467476806 - 0.026097049754484028im
```

# Initial example

Function:

```
f(x, y) = y / (y + exp(-2x))
```

Matrix math:

```
julia> using LinearAlgebra  # for I
julia> f([1 2
          0 3], I)

2x2 MatrixFloat64:
 0.880797  0.11673
 0.0       0.997527
```

# Initial example

Function:

```
f(x, y) = y / (y + exp(-2x))
```

Interval arithmetic:

```
julia> using IntervalArithmetic: Interval

julia> f(Interval(1, 2), Interval(1, 1.5))
[0.611495, 1.47303]
```

# Initial example

Function, with wrappers:

```
f(x, y) = y / (y + exp(-2x))
g(x)    = f(x, 1)
h(v)    = f(v[1], v[2])
```

Derivatives:

```
julia> using ForwardDiff: derivative, gradient

julia> derivative(g, 2)
0.03532541242658223

julia> gradient(h, [2, 1])
2-element VectorFloat64:
 0.03532541242658223
 0.017662706213291135
```

# Initial example

Function:

```
f(x, y) = y / (y + exp(-2x))
```

Error propagation:

```
julia> using Measurements

julia> f(1 ± 2, 1)
0.88 ± 0.42
```

Uses automatic differentiation to propagate the uncertainty.

# A word about typing

- `x::T` is a *type assertion*
  $\Longrightarrow$ throw an error if `x` is not of type `T`
- Can also specify argument types:
  `f(x::Int, s::String) = ...`
- Typing is optional in general
- Julia's *type inference* will try to figure out which types are used, even when none are specified

# Multiple dispatch

Functions can be overloaded based on all argument types:

```
f(x, y)           = x + y
f(x::Int, y::Int) = x - y
```

The most speficic method that applies is called:

```
julia> f(10,   1)
9
julia> f(10.0, 1)
11.0
julia> f(10,   1.0)
11.0
julia> f(10.0, 1.0)
11.0
```

# Multiple dispatch

Ambiguous overloading:

```
f(x, y) = x + y
f(x::Int, y) = x - y
f(x, y::Int) = y - x
```

No most specific method $\Longrightarrow$ error:

```
julia> f(10, 1)
ERROR: MethodError: f(::Int64, ::Int64) is ambiguous. Candidates:
  f(x::Int64, y) in Main at example.jl:2
  f(x, y::Int64) in Main at example.jl:3
Possible fix, define
  f(::Int64, ::Int64)
Stacktrace:
 [1] top-level scope
   @ REPL[6]:1
```

# Plugging in your own type

A simple type:

```
struct MyType
        x::Int
        y::Int
end
```

Make addition work for `MyType`:

```
julia> import Base: +

julia> +(a::MyType, b::MyType) = MyType(a.x + b.x, a.y + b.y)

julia> MyType(1, 2) + MyType(10, 100)
MyType(11, 102)
```

# JIT compilation and function specialization

- Julia is just-in-time (JIT) compiled
- Call function with new argument types
  $\Longrightarrow$ JIT compile specialized version
- Specialized code can be fast
  $\Longrightarrow$ don't have to call another language like C/C++ for speed
  $\Longrightarrow$ most of Julia is written in Julia

# Comparison to object orientation

In an object oriented language:

```
result = x.f(y, z)
```

In Julia:

```
result = f(x, y, z)
```

- Most OO idioms can be translated to Julia, syntax looks a bit different
- Behavior can be inherited
- Fields can not – avoids fragile base class problem
- Functions outside of types
  $\Longrightarrow$ can add behavior after type has been created
  $\Longrightarrow$ creates lots of extensibility

# Some more nice things

- Capable terminal interface (REPL)
- Friendly syntax for matrix and array operations
  - Matrix literals

```
A = [1 0 2
     0 1 3]
```

  - `A * B` for matrix product, `A .* B` for elementwise, etc
  - ...
- Integrated package manager
  - Records used package versions for reproducibility

# Some drawbacks

- JIT compilation can take a little time
- Not as many libraries compared to older languages (but still many!)

# Summary

- Expressive
  - Easy to make different codes work together
- Friendly syntax for matrix and array operations
- Can be fast
  - (JIT) Just-in-time compiled
  - Designed to allow the JIT to produce good code