Why Julia?

There are traditionally two categories of computer languages:

Compiled - script file translated to machine code and linked to libraries once
- the executable program file is static, data types static
- examples: C/C++, Fortran
- fast, suitable for demanding high-performance computing
- not user-friendly handling of external packages, e.g., graphics

Interpreted - the script file is translated line-by-line at run time
- there is no static executable, allows more flexible functionality
- examples: Python, Perl, R
- slow; most time is spent translating the script over and over again
- more flexible handling of data (dynamic, automatic data typing)
- friendly integration of packages, graphics, notebooks,…
- not user-friendly for improving efficiency (e.g., precompiled parts)

Julia: first successful “best of both worlds” language

- v0 launched in 2012, v1.0 in 2018, now v1.10.5

Key: Just-in-time (just-ahead-of-time) compilation

- goes through the script line-by-line, but saves compiled machine code for efficiency-critical parts (loops, entire functions)

https://julialang.org

Almost as fast as C/C++ and Fortran (within ~10%)

- designed specifically for high-performance scientific computing

As dynamic as Python

- data types can change dynamically, but can also be declared

Good mechanism for incorporating external packages/libraries

- C/C++ and Fortran codes can also be incorporated easily

Library module “Base” is automatically included, extensive functionality

Other modules can easily be imported and used

- growing user community, many packages available in different fields

Introduction to Julia

The language has many features; here we just cover the basics

- PY502 is not a software engineering course
- We will not cover advanced programming
- We will (later) pay attention to code performance (execution speed)

Teaching method: brief general principles + code examples

- commented codes available on the course web site

http://physics.bu.edu/py502/lect1/examples/

Variable types and elements to get started

[int1.jl] Integer declaration and wrap-around (mod) behavior
[int2.jl] Integer declarations; modified version of int1, run-time error due to type mismatch
[randomarray.jl] Function with two methods; generates array of Float32 or Float64 random numbers
[matrix.jl] Matrices and matrix operations

There are not yet any good Julia books (?)

Documentation on the Julia site is quite good https://julialang.org

- please read and practice elements we do not cover here!

Three ways to run Julia

1) Code written in file, run from terminal command line

    $ julia yourcode.jl (list of arguments may follow)

    This is the way for serious work

2) Using interactiv REPL (read-execute-print-loop) session

    $ julia (opens interactive session)

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.10.5 (2024-08-27)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

- Useful for learning and testing (small code pieces)
- Package manager (import modules with specific functionality)

3) Run in Jupyter notebook
   - Install the Julia kernel first

Examples with animations:
http://docs.juliaplots.org

Bit representation of integers

A "word" representing a number in a computer consists of B bits

- normally B=32 or 64, also in some cases 16 or 128
- a group of 8 bits is called a "byte" (normally a word is 4 or 8 bytes)

B-1				3	2	1	0
0	0	1	· · · ·	0	1	0	1

bit index i = 0,1,…,B-1

bit values b(i) = 0/1

For signed integers, the last bit (B-1) is called the "sign bit"

- b_{B-1} = 0 for positive (or zero) values, b_{B-1} = 1 for negative values

For positive (or 0) integer I, the value corresponding to the bits is

$$I = \sum_{i=0}^{B-1} b(i)2^i$$

00 …. 0000 = 0
00 …. 0001 = 1
00 …. 0010 = 2,….

For I < 0, "two's complement" representation:

$$I = \sum_{i=0}^{B-2} b(i)2^i - b(B-1)2^{B-1}$$

Positive to negative:

- reverse all bits
- add 1 (ignore overflow)

11 …. 1111 = -1
11 …. 1110 = -2
11 …. 1101 = -3,….

- most practical way for computer algebra
- integer operations have"wrap around" behavior (mod 2^B for unsigned)

Example: integer declarations and operations [int1.jl]

```
function integertest()
   a::UInt32=typemax(UInt32)
   b::UInt32=1
   c=a+b
   return a,c
end
x,y=integertest()
println(x)
println(y)
```

Base function typemax gives largest value - typemin gives smallest

function "integertest" with no arguments is declared

variables a, b declared as unsigned 32-bit integers and given values

two integers are returned by the function

Base function println writes a line to standard output

Output:

```
$ 4294967295
$ 0
```

$2^{32} - 1$

$(2^{32} - 1 + 1) \bmod 2^{32}$

Try also with "Int32" instead of "UInt32"!

Example with an error [int2.jl]

Changing the function to (keep the rest of the previous example)

```
function integertest()
   a::UInt32=typemax(UInt32)
   b::UInt32=1
   b=a+1
   return a,b
end
```

Running gives this error message (+ more):

```
ERROR: LoadError: InexactError: trunc(UInt32, 4294967296)
```

Reason: My computer (and likely yours) is based on 64-bit architecture

- the constant "1" is then of type Integer64
- a+1 also is of type Integer64 (the "larger" of the two types involved)
- b is declared as UInt32 and cannot represent the value pf a+1

Integer types in Julia

`Int8, Int16, Int32, Int64, Int128`
`UInt8, UInt16, UInt32, UInt64, UInt128`

`Int` is the default integer type
- normally same as `Int64`