Julia for Scientific Programming

Seminar 1: Basics 1

Kristian Soltesz

Dept. of Automatic Control
Lund University

Practical Stuff

- Homework (getting started, finding documentation)
- Course web page (navigate from main page)
- Forum for questions/discussions at Piazza (e-mail sent out)
- All slides on course home page (and git repo)

Today’s Goal

...is to figure out:

- Where (among languages) does Julia belong?
- How do we use variables
- How to use numerics
- How to define and use functions
- Learn how to alter control flow

This seminar will roughly follow the chapters of `docs.julialang.org` until *Types*.

We will also talk a bit about how to structure the remainder of the course.

What is Julia?

- A new (2012 release) language for scientific computing
- Open source and free (MIT) license
- Just In Time (JIT) compilation to *native* machine code allows for C-speed execution
- Small core, most functionality implemented in the language itself
- Multi-paradigm: procedural / functional / object-oriented / concurrent
- Multi-dispatch allows dynamic binding at runtime

Executing Code

Just like Matlab and Python, there are two execution modes:

- interactive (prompt / read-eval-print loop)
- non-interactive mode

Variables

Variable can be (almost) any unicode sequence. For instance $\alpha = 5$ is a valid variable declaration. (If you don't know the unicode for α, julia accepts the LaTeX code followed by a tab.

Conventions:

- Variable names start lower-case
- Type names start uppercase
- camelCasing is used in favor of under_score
- Functions that modify their input in-place have names ending in `!`

Type `whos()` to see defined variables and `typeof(myVariable)` to get type info. The `ans` variable works like in Matlab.

Primitive Numeric Types

- Normal integers: `IntX` where `X` is either of $8, 16, 32, 64, 128$ and corresponding (unsigned) `UintX`.
- Boolean: `Bool` internally represented as 8 bit int with values 0 or 1.
- Character: `Char` internally represented as 32 bit int.
- Floats: `Float16` (half), `Float32` (single) and `Float64` (double).

The type is decided at construction, and type casting is *implicit*. Try for instance:

```
julia> a=1;b=1.;c=a*b; typeof(c)
```

Explicit casting is possible: `convert(Int8,3.)` Unlike Matlab, overflow is cyclic. Try adding 1 to `intmax` in Matlab, and to `typemin(Int64)` in Julia.

Composite Numeric Types

- There is a rational type associated with each integer type, constructed as `7//8`.
- Arbitrary precision arithmetics are available through the `BigInt` and `BigFloat` types (constructors with same names).
- Complex numbers can be constructed like `1+3im` or `complex(1,3)`. Note that no `*` is required between `3` and `im`.

Casting to complex numbers is not implicit. `sqrt(-1)` throws an error.

Almost like Matlab

Operation on numbers resembles Matlab, with some improvements:

- Several operations can be combined with assignment: `a+=3` instead of `a=a+3`.
- Coefficients can be more compactly expressed: `a^2b` instead of `a^(3*b)`.i
- The float variable `x=1.` prints as `1.0`, not `1`.
- Chaining comparisons is possible (like in Python). Try `x=1.5; 1<x<2`
- If the type is associated with a zero or one, these are explicitly available: `one(Int32)`, `zero(1.)`

Strings

If you plan to use Julia to process data from files or streams, read the chapter on Strings. We will skip it in favor of...

Functions 101

Vanilla function definition:

```
function f(x)
  x+1
end
```

- Calling the function: `f(3)`
- Note how output is not defined at the function declaration (more on this later).
- Parenthesis required even for functions without arguments, such as `eps()`.
- Last evaluated expression in function body is returned. Julia also implements the `return` keyword, just like Matlab.

More on Functions

- Operators are functions. Try `+(1,3,4)`
- Anonymous functions are created using `->`. Example of use: `map(x->x^2,[1 2 3])` produces `[1 4 9]`. Mapping functions is familiar for Python users as a powerful tool.
- Functions return a single object. However, it can be a tuple with the comma as constructor:

  ```
  function foo(a,b)
    a+b, a-b
  end
  ```

  There is built-in support for destructuring tuples, making it look like the function returns several objects, for example try: `x,y=foo(1,1)`
- Named functions can be defined in-line without the function keyword: `square(x)=x^2`.

Passing Arguments

- Variable number of input arguments (vararg) functions can be defined using ellipsis: `f(first,rest...)=(first,rest)`. The `rest` list of input argument is accessible inside the function as a single tuple `rest`..
- The ellipsis can also be used to splice an iterable collection into a list: used in an argument:

  ```
  x=(2,7,3);
  max(x...)
  ```
- It is possible to define functions with optional arguments: `grow(x,y=2)=x*y` can be called either by `grow(3)` or `grow(3,4)`.
- There are also keyword (named) arguments, following semicolon in declaration: `function f(x;y=0)` can be invoked `f(3,y=4)`.

Brief Note on Functional Programming

Functions are first class citizens in Julia. They can be passed around like any value:

```
f(x,y)=x+y
g(x)=f(x,3)
```

As we have seen, there is also support for anonymous (aka lambda) functions. These two facts make it possible to use Julia as a *functional* lanaguage. Aha experience for those who know Haskell or Lisp.

Control Flow 101

Very similar to Matlab with familiar keywords: `if`, `else`, `elseif`, `for`, `while`, `continue`, `throw` `error`, `warn` (`warning` in Matlab) `try`, `catch`, `end`.

Short circuit "lazy" evaluation as in Matlab and C.

Some new friends:

- `begin-end`. Lumps several expressions into one. However, no scoping.
- `finally` makes code in block after `finally` keyword run regardless if the block exits clean or through exception.

Note on Light Threading

Julia provides another type of control flow control through tasks (aka light treads). Tasks can take on states *runnable, waiting, queued, done, failed*. They are a convenient tool for dealing with external events, such as I/O. Read more in the documentation!

Practical

- Homework 1. Go to `projecteuler.net`, select 1-3 problems and solve by coding in Julia.
- Next meeting. Suggested Friday August 28. Someone volunteering to go though *methods, constructors, conversions, modules?*