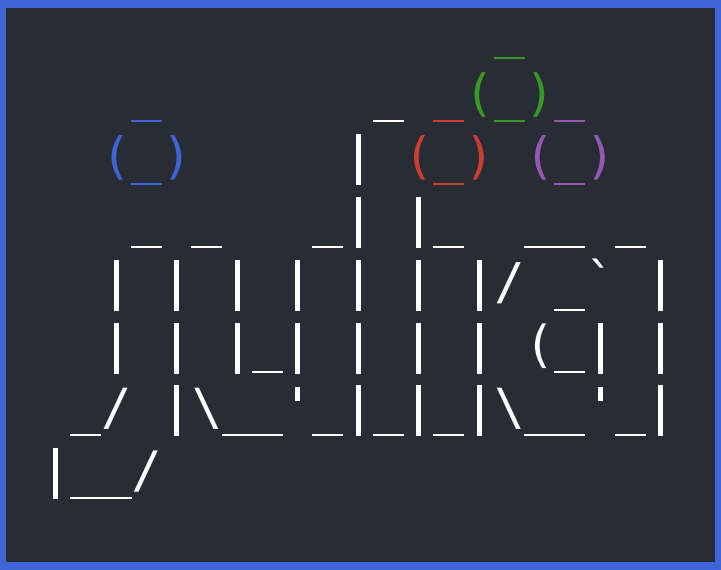

julia cheatsheet

A handy reference for the lecture Advanced Statistical Physics

Philipp Hintz

2020-12-14

Contents

Introduction

This is a (not so) small and elective collection of important code-snippets for the Advanced Statistical Physics lecture. For convenience of julia-beginners there is a small basic tutorial, which is directed to people who have experience in programming but are new to julia. Another good start is julia express. Other cheat cheets are from juliadocs the comparison-cheatsheet from [QuantEcon]{https://cheatsheets.quantecon.org/}, which is also great and lists many useful snippets. More comprehensive examples can be found on rip-tutorials.

Specifics of julia

julia is a high-performance, dynamic programming language and very convenient for numerical calculations. Especially in comparison to Python the speed is sometimes magnitudes better and close to C.

Differences to other programming languages

- julia is quite strict with the definition of so-called *scopes*. This is the region in which variables are defined. E.g. you cannot call variable from outside within a for-loop. More on this later in the section *pitfalls*.
- no semicolons needed at the end of line (but they can be used in the interactive REPL-promt to supress output)
- `if`, `for`, `while`, … have to be terminated by `end`
- strings have to be placed of double quotation marks: `"foo"`
- indexing of arrays etc. is 1-based not 0-based, end is given by `end`, e.g. `f[1], f[2], ..., f[end-1], f[end]`. Be aware of this when looping (out-of-bonds-error)!
- a range is given by `start:step:stop`
- Julia saves arrays "column major", whereas languages like C and Python use "row major", so use loops along columns for best speed. Details: link
- Since julia is very fast, in most cases you do not have to write "vectorized" code. Instead just use loops.

Recommended IDEs

Although you can use any plaintext-editor for writing code, I can recommend Jupyter if you are used to it or like the use of the so-called notebooks (IJulia). If not or you like more plaintext based IDEs, try Atom or Visual Studio Code, which have nice features like displaying plots, a built-in profiler (shows which lines of code take much computation time), workspace (view of all variables and their values) and progressbars. Read the respective section for more details.

Important code-snippets

The following commands and fragments can be useful for the course.

General syntax

task	code
comments	`#`
comment block	`#= [...multiple lines...] =#`

task	code
null value	`nothing`
use last result from interactive REPL	`ans`
every command ending with an exclamation mark mutates its argument	`push!(A,x)` appends x to A
dispatch the type of an argument with `::`	`f(x::Integer)`
define a function the short way	`square(x) = x^2`
the long way (x integer, y float and z any with default value 0 if not set)	`function f(x::Int,y::Float64,z=0)`

Types of variables

Convert variable-type of variable x to *TYPE*: `convert(TYPE, x)`

Find the type of a variable: `typeof(x)`

`Int8`, `Int16`, `Int32`, `Int64`, `Int128`, `BigInt`, `UInt8`, `UInt16`, `UInt32`, `UInt64`, `UInt128`, `Float16` (half precision), `Float32` (single), `Float64` (double), `BigFloat`, `Bool`, `Complex` (like `1 + 2im`), `Rational` (like `1//2`).

The type of `Int`, `UInt` and `Float` without a number depends on the operating systems architecture, so usually a word size of 64. Rationals allow exact calculations with fractions. Use syntax `1//2` to define them. More details and the range of supported values can be found in the docs.

Basic math

task	code
exponentiation	`e^x`
square-root	`sqrt(x)`
imaginary unit	`im`
absolute value	`abs(x)`

task	code
modulo (remainder)	`x % y`
integer divide (truncated to an `Int`)	`x ÷ y`

Logic

task	code
Numeric comparisons	`==, !=, <, <=, >, >=, ===` (last one is true, if value *and* type are the same -> egality)
AND	`x < y && y < z` or (faster) `x < y < z`
OR	`x || y`

for-loop:

```
for i in 1:10
    x += i
end
```

multidimensional for-loop: `for i in 1:N, j in 1:N`

while-loop:

```
while x < 1
    ...
end
```

terminate loop prematurely: `break` terminates the loop, `continue` skips the current iteration and jumps to the next one, `return` ends the whole function

if-then-else:

```
if x == 1
    ...
elseif x <= 0
    ...
else
    ...
end
```

increase variables: `x += 1` or also `x *= 2`

Random numbers

task	code
return a random number between 0 and 1	`rand()`
return a random number between 0 and 10	`rand(0:10)`
return a m×n array with random numbers from 0 to 10	`rand(0:10, m, n)`
pick randomly 1 or 2	`rand((1, 2))`
generate an normally-distributed array with mean 0 and σ 1	`randn(m, n)` or just `randn()`
randomly shuffle (permutate) an array	`shuffle!(A)`
randomly cyclic permutate an array	`randcycle!(A)`

Arrays

task	code
access element (1 is first one!)	`A[i]`
access last element	`A[end]`
access n-last element	`A[end - n]`

task	code
access sub-array from m to n	`A[m:n]`
access a single row/column	`A[i, :]`, `A[:, i]`
sum/mean of all elements of an array	`sum(A)`, `mean(A)`
get the size or an matrix	`size(A)`
add element x to an array A (as last/first element)	`push!(A, x)`, `pushfirst!(A, x)`
combine (concatenate) arrays horizontally/vertically	`hcat(A, B)` or `[A B]` and `vcat(A, B)` or `[A ; B]` respectively
append elements from A to B	`append!(A, B)`
you can also combine them by using square bracket notation with spaces and semicolons or commas	`[A B ; C D]`
remove last/first element of array (it's gone in A then) and return it	`pop!(A)`, `popfirst!(A)`
delete element from A at index `i`	`deleteat!(A, i)`
sort an array	`sort!(A)`
check whether an value x is in array A	`in(x, A)` or simply `x in A`

Create and fill arrays

task	code
by hand	`[1 2 3, 4 5 6, 7 8 9]`
m×n filled with special values (type like `Int64` is optional)	`ones(Int64,m,n)`, `zeros(Int64,m,n)`, `rand(Int64,m,n)`, `trues(m,n)`, `falses(m,n)`

task	code
m×n filled with defined value 42	`fill(42, (m×n))`
fill existing array A with value 42	`fill!(A, 42)`
initialize an empty array (type *Any* -> worst performance)	`A = []`
with a specific type (better performance)	`A = Int[]`
with type and given dimensions (2×3, best performance)	`A = Array{Int,2}(undef, 2, 3)`
`Matrix{Type}` is shorthand for `Array{Type,2}`	`A = Matrix{Int}(undef, 2, 3)`
with formula	`A = [i^2 for i=1:100]`
with formula and condition	`A = [i for i=1:100 if i%2==0]`

element-wise-operations

task	code
To execute mathematical operations between the element of two arrays (element-by-element) just put a dot `.` before the operator	`A .* B`
scalar product	`sum(A .* B)` or `dot(A, B)`

Linear Algebra

task	code
identity matrix (dimensions fit automatically)	`I`
determinant, trace, inverse,	`det(A)`, `tr(A)`, `inv(A)`

task	code
euclidian norm, maximum, minimum	`norm(A)`, `maximum(A)`, `minimum(A)`
adjugate matrix	`A'` or `adjoint(A)`
rank, eigenvalues, eigenvectors	`rank(A)`, `eigvals(A)`, `eigvecs(A)`

Outputs

task	code
print (with or without linebreak)	`print(x)`, `println("hello")`

Inputs

task	code
get user-input (chomp removes possible linebreaks)	`chomp(readline())`

Handle text-files

task	code

Plot diagrams

task	code

Pitfalls

Out-of-bonds-error since the first element starts at 1, so `A[0]` is never valid.

`step` is a reserved expression in julia, so do not name your functions `step`.

scope

Work in progress

Missing:

1. element-wise operations
2. Global
3. Pre initialize
4. Plot
5. Ranges
6. concatenate strings
7. Vector array matrix performance
8. Inplace operations a=b
9. Int float
10. Map, filter
11. Foreach
12. Ceil, floor
13. Mean variance
14. collect …