Julia & Pluto: First Contact

Julia & Pluto: First Contact

Advanced Topics from Scientific Computing
TU Berlin Winter 2024/25
Notebook 01
Jürgen Fuhrmann

What is Julia ?

- Computer language for Scientific computing
- Just-Ahead-Of-Time (AOT) compilation of code
- Accessible syntax (on the level of Matlab, Python)
- Multidimensional arrays as first-class objects
- Efficient generic code (like C++ generic lambdas and templates)
- Package ecosystem, best-in-class package manager

Open Source

- Julia is an Open Source project started at MIT
- Julia itself is distributed under an MIT license
    - packages sometimes have different licenses
    - all packages installable by default have open source licenses
- Development takes place on github
- Open Source paradigm & package management support transparency and reproducibility in scientific research

Resources

- Homepage
- Documentation
- Cheat Sheet
- WikiBook

Installation

- Long term support (LTS) Version: 1.10
    - Recommended as the minimal version for new projects
- Current stable version: 1.11.0
    - Recommended for this course
- Download from julialang.org with the help of the `juliaup` installation manager
    - Nowadays recommended by Julia creators
    - Avoid installations via homebrew, yast etc.
- Linux + Mac:

```
$ curl -fsSL https://install.julialang.org | sh
```

- Windows:

```
> winget install julia -s msstore
```

Running Julia: the REPL

- Read-Evaluation-Print-Loop when starting `julia` in terminal
- REPL modes:
  - **Default mode:** `julia>` prompt. Type backspace in other modes to enter default mode.
  - **Help mode:** `help?>` prompt. Type `?` to enter help mode. Search via `?search_term`
  - **Shell mode:** `shell>` prompt. Type `;` to enter shell mode.
  - **Package mode:** `Pkg>` prompt. Type `]` to enter package mode.
- Helpful commands in REPL:
  - `quit()` or `Ctrl+D`: exit Julia.
  - `Ctrl+C`: interrupt execution.
  - `Ctrl+L`: clear screen.
  - Append `;` to suppress displaying output from a command
  - `include("filename.jl")`: source a Julia code file.

Running Julia: VSCode & Jupyter notebooks

- Visual Studio Code has a Julia extension
  - A Julia REPL session can be started from within vscode
  - Julia graphics is integrated with the vscode graphics pane
- Jupyter notebooks
  - Julia is the "Ju" in Jupyter
  - Jupyter can have a Julia kernel
  - Two disadvantages of Jupyter notebooks
    - they store computational results in the notebook
    - no accounting for cell dependencies

Running Julia: Pluto notebooks

Pluto is a browser based notebook interface for the Julia language. It allows to present Julia code and computational results in a tightly linked fashion.

- **Pluto is like Google Sheets or Excel but with Julia code in its cells.** Pluto cells are arranged in one broad column. Communication of data between cells works via variables defined in the cells instead of cell references like `A5` etc. With Excel and other spreadsheets, Pluto shares the idea of **reactivity**: If a variable value is changed in the cell where it is defined, the code in all dependent cells (cells using this variable) is executed.
- **Pluto is like Jupyter for Julia, but without the hidden state** created by the unlimited possibility to execute cells in arbitrary sequence. Instead, it enhances the notebook concept by reactivity.
- Pluto is implemented in a combination of Julia and javascript, and can be installed like any other Julia package.
- During this course, Pluto notebooks will be used to present numerical methods implemented in Julia.

Installing and starting Pluto

Install Julia on your computer and run it. Issue:

```
julia> using Pkg
julia> Pkg.add("Pluto")
julia> Pluto.run()
```

or instead of the last statement

```
julia> Pluto.run(notebook="MyNotebook.jl")
```

Pluto resources

- plutojl.org homepage with installation hints
- Pluto repository at Github
- How to install Pluto (straight from the main author Fons van der Plas)

- Sample notebooks are available via the index page after starting Pluto.

Pluto structure

- Pluto notebooks consist of a sequence of cells which contain valid Julia code.
- The result of execution of the code in a cell is its return value which is displayed on top of the cell.

Markdown cells

Cells can consist of a string with text in Markdown format given as a Julia string prefixed by `md`. This single text string is valid Julia code and thus returned, formatted and shown as text.

This is a markdown cell. We can **highlight** or *emphasize* text.

- Bullet lists can be nested
    - like this
- Web links are easy to write. Markdown is easy!

```julia
md"""
This is a markdown cell. We can __highlight__ or *emphasize* text.

- Bullet lists can be nested
   - like this
- Web links are easy to write. [Markdown](https://www.markdownguide.org/getting-started/) is easy!
"""
```

Cell content visibility

Cell content can be visible...

```
md"""
Cell content can be visible...
"""
```

... or hidden, but their return value is visible nevertheless. Content visibility can be toggled via the eye symbol on the top left of the cell.

$\LaTeX$ in markdown cells

Markdown cells can contain $\LaTeX$ math code: $\int_0^1 sin(\pi\xi)$. Just surround it by $ symbols as in usual $\LaTeX$ texts or better by double backticks: ``\int_o^1 sin(π ξ) dξ``. The later method is safer as it does not collide with string interpolation (explained below).

Math in display mode is written like this:

$$\int_0^1 sin(\pi\xi) d\xi$$

~~~
md"""
Math in display mode is written like this:
```math
\int_0^1 sin(π ξ) dξ
```
"""
~~~

HTML cells

Instead of a markdown string, cells also can return a string prefixed by `html` containing HTML code.

deutsch	english	Українська	中文
Bier	beer	пиво	啤酒
Tee	tea	чай	茶

```
html"""<table>
  <tr>
    <th>deutsch</th>    <th>english</th> <th>Українська</th>    <th> 中文 </th>
  </tr>
  <tr>
    <td>Bier</td>    <td>beer</td>  <td>пиво</td>   <td> 啤酒 </td>
   </tr>
   <tr>
     <td>Tee</td> <td>tea</td>  <td>чай</td> <td> 茶</td>
   </tr>
</table>
"""
```

Embedded javascript

A html string can contain javascript code. This allows to make use of the vast amount of Javascript libraries designed for interactive use in the browser.

+
−
Cauerstraße
Abbestraße
Landwehrkanal
Einsteinufer
Salzufer
Gutenbergstraße
Englische Straße
Siegmunds Hof
Bachstraße
Tiergarten
Charlottenburger Brücke
B 2
B 5
Bismarckstraße
Ernst-Reuter-
Technische
Leaflet

```
html"""
<link rel="stylesheet" href="https://unpkg.com/leaflet@1.7.1/dist/leaflet.css"/>
<script src="https://unpkg.com/leaflet@1.7.1/dist/leaflet.js"></script>
<div id="mapid" style="width: 600px; height: 200px;"></div>
<script>
  var map = L.map('mapid');
  map.setView([52.514, 13.3257], 15);
  var layer = new L.TileLayer('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png');
  map.addLayer(layer);
</script>
"""
```

Julia code cells

Code cells are cells which just contain "normal" Julia code. Running the code in the cell is triggered by the `Shift-Enter` keyboard combination or clicking on the triangle symbol on the right below the cell.

```
2
```

```julia
1+1
```

```
[5, 7, 9]
```

```julia
[1, 2, 3] + [4,5,6]
```

Showing the return value can be suspended by adding a `;` after the cell statement.

```julia
17+19;
```

Variables and reactivity

Define a variable in a cell. The assignment has a return value which is shown on top of the cell.

```
x = 102
```

```julia
x=102
```

A variable defined in one cell can be used in another cell. Moreover, if the value is changed, the other cell reacts and the code contained in that cell is executed with the new value of the variable. This is *reactive* behaviour typical for a spreadsheet.

```
103
```

```julia
x+1
```

One can return several results by stating them separated by `,` . The returned value then is a tuple.

```
(103, 104, 105)
```

```julia
x+1,x+2,x+3
```

The dependency of one cell from another is defined via the involved variables and not by the sequence in the notebook. In order to achieve this, Pluto makes extensive use of **reflexivity**, the possibility to inspect the variables defined in a running Julia instance using Julia itself.

Only one statement per cell

Each cell can contain only exactly one Julia statement. Multiple expressions can be grouped into one by surrounding them by `begin` and `end` or by having the in one line, separated by `;` . The return value will be the return value of the last expression.

```
0.9948267913584063
```

```julia
begin
    z=x+v
    sin(z)
end
```

```
0.9948267913584063
```

```julia
z1=x+v; sin(z)
```

However, in this situation the better structural decision would be to combine the statements into a function defined in one cell and to call it in another cell.

```julia
function f(a,b)
    c=a+b
    sin(c)
end;
```

```
0.9948267913584063
```

```julia
f(x,v)
```

Loading packages

In Julia, packages provide additional functionality on top of the standard functionality of Julia.

In Pluto notebooks, adding and loading packages is performed via the `using` statement. This will be discussed in more depth later.

```julia
begin
    using PlutoUI
    using LinearAlgebra
end
```

Interactivity

We can bind interactive HTML elements to variables. The Julia package PlutoUI.jl provides a nice API for this.

v = 0

```julia
@bindname v PlutoUI.Slider(0:20,show_value=true)
```

v=0 (This uses *string interpolation* to print the value of v into the Markdown string)

```julia
md"""v=$(v)  (This uses _string interpolation_ to print the value of v into the Markdown string)"""
```

Deactivating code

Deactivating cells before running their code can be useful for preventing long runnig code to start immediately after loading the notebook or for pedagogical reasons.

The preferred pattern for this uses a checkbox bound to a logical variable.

Run next cell: ☐

```
md"""
 __Run next cell__: $(@bind allow_run PlutoUI.CheckBox(default=false))
"""
```

```
if allow_run
    a=rand(1000,1000)
    ainv=inv(a)
    sum(eigvals(ainv)),tr(ainv)
end
```

Cells also can be deactivated using the corresponding button in the cell menu.

Live docs

The live docs pane in the lower right bottom allows to quickly obtain help information about documented Julia functions etc.

```
sin (generic function with 14 methods)
```

```
sin
```

With `@doc` one can also show documentation as cell output (however this interfers a bit with the headings)

Showing output during calculations

Due to the idea to show information as return results of notebook cells, text output via print statements is not the "Pluto way" to convey information.

Sometimes it is however desirable to inspect text output during calculations.

Pluto shows the terminal output below the cell.

```julia
for i=1:10
    println("i=$(i)")
end
```

```
i=1
i=2
i=3
i=4
i=5
i=6
i=7
i=8
i=9
i=10
```

Logging

It is possible to use the logging features of Julia. Instead of using `println`, you can use `@info`, `@warn` and `@error` to inspect code running in a cell. This feature is quite powerful. Besides of bare text strings you can log arrays and other data.

```julia
for i=1:5
    @info "i=$(i)"
end
```

```
i=1

i=2

i=3

i=4

i=5
```

Presentation mode

Pluto notebooks have a presentation mode which can be toggled by the cooresponding button in the top pane and left by `ESC`.

First and second level headers become slide headers, and we get some navigation controls.

Styling

Styling can be done by overwriting the standard CSS of Pluto in a html cell of the notebook.

Please be aware that this may interfer with the dark/light theme switch of your browser or your system.

```
  html"""
   <style>
    # headers
    h1{background-color:#dddddd;  padding: 10px;}
    h2{background-color:#e7e7e7;  padding: 10px;}
    h3{background-color:#eeeeee;  padding: 10px;}
    h4{background-color:#f7f7f7;  padding: 10px;}

    # terminal output
    pluto-log-dot-sizer { max-width: 655px;}
    pluto-log-dot.Stdout {
    background: #002000;
    color: #10f080;
    border: 6px solid #b7b7b7;
    min-width: 18em;
    max-height: 300px;
    width: 675px;
    overflow: auto;
   }
   </style>
"""

```

```
using HypertextLiteral: @htl, @htl_str
```

```
html"""<style>.dont-panic{ display: none }</style>"""
```

```
@statement_str (macro with 1 method)
```

```
begin

    # Highlighting some text with background color
    highlight(mdstring,color)= htl"""<blockquote style="padding: 10px;
background-color: $(color);">$(mdstring)</blockquote>"""

    macro important_str(s)  :(highlight(Markdown.parse($s),"#ffcccc")) end
    macro definition_str(s) :(highlight(Markdown.parse($s),"#ccccff")) end
    macro statement_str(s)  :(highlight(Markdown.parse($s),"#ccffcc")) end

 end
```

Conclusion

- Pluto notebooks provide a flexible, reproducible and lean possibility to convey algorithmic content in the Julia language.
- I will use Pluto notebooks for almost all code examples
- Coding assignments will be done in Pluto

- Your first homework will be the installation of Julia and Pluto on your computer

---