Scientific Computing TU Berlin Winter 2021/22 © Jürgen Fuhrmann
Notebook 01

Julia: First Contact - Basic Pluto
- What is Pluto ?
- Pluto resources:
- Pluto structure
  - Markdown cells
  - Cell content visibility
  - LATEX in markdown cells
  - HTML cells
  - Embedded javascript
  - Julia code cells
- Variables and reactivity
  - Only one statement per cell
  - Suppression of return values
- Interactivity
  - Deactivating code
  - Live docs
  - Showing output of print statements
- Loading packages
- Conclusion

Julia: First Contact - Basic Pluto

What is Pluto ?

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.

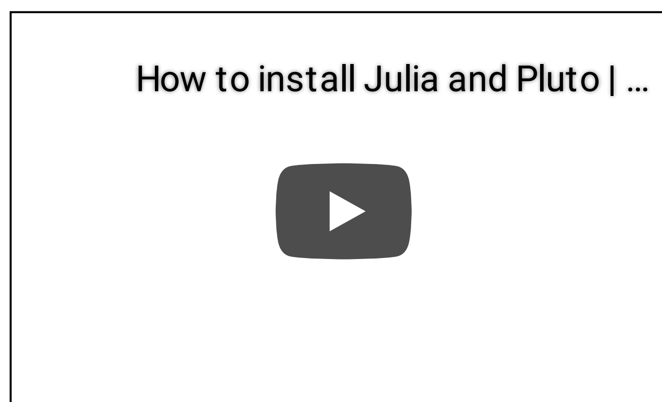
- For those familiar with spreadsheets: **Pluto is like Google Sheets or Excel but with Julia code in its cells.** Pluto cells are arranged in one broad column. Communication of data between cells works via variables defined in the cells instead of cell references like `A5` etc. With Excel and other 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.
- For those familiar with Jupyter notebooks: **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.

Pluto resources:

- **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.

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: $\int_0^1 sin(\pi\xi) d\xi$. Just surround it by $ symbols as in usual $\LaTeX$ texts or by double backtics: ``\int_0^1 sin(π ξ) dξ``. The later method is safer as it does not collide with string interpolation (explained below).

HTML cells

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

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

Embedded javascript

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

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.

Variables and reactivity

We can define a variable in a cell. The assignment has a return value like any other Julia statement which is shown on top of the cell.

x = 100

```
x=100
```

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 *reactive* behaviour typical for a spreadsheet.

101

```
x+1
```

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

(101, 102, 103)

```
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. If multiple expressions are desired, they can made into one by surrounding them by `begin` and `end`. The return value will be the return value of the last expression in the statement.

-0.5063656411097588

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

An alternative way is to have all statements on one line, separated by `;` :

```
0.8623188722876839
```

```
z1=x+v; cos(z1)
```

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.

```
f (generic function with 1 method)
```

```
function f(x,v)
    z=x+v
    cos(z)
end
```

```
0.8623188722876839
```

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

Suppression of return values

```
md"""
Display of the return value can be suppressed by ending the last statement with `;`
""";
```

Interactivity

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

0

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

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

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

Deactivating code

We occasionally will use the possibility to deactivate cells before running their code. This 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: ☐

```
if allow_run
    a=rand(2000,2000)
    ainv=inv(a)
    sum(eigvals(ainv)),tr(ainv)
end
```

Cells can be deactivated using the corresponding button in the cell menu, however, this state cannot be stored in the notebook file:

```
(-0.045231-4.02456e-15im, -0.045231)
```

```
begin
    b=rand(2000,2000)
    binv=inv(b)
    sum(eigvals(binv)),tr(binv)
end

```

Live docs

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

```
cos (generic function with 24 methods)
```

```
cos
```

Showing output of print statements

Normally text output from statements in a cell is shown in the console window where Pluto was started, and not in the notebook, as Pluto focuses on the presentation of the results. Sometimes it is however desirable to inspect this output. Instead of looking for this output in the console window where the browser has been started. One can use the function `with_terminal` from `PlutoUI.jl`.

```
10
```

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

```
PlutoUI.with_terminal() do
    for i=1:10
        println("i=$(i)")
    end
    10
end
```

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.

```
begin
    using PlutoUI
    using LinearAlgebra
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