Advanced Topics from Scientific Computing
TU Berlin Winter 2023/24
Notebook 02
(cc) BY-SA **Jürgen Fuhrmann**

Julia - first contact

Julia - first contact

General information

Resources

- Homepage
- Documentation
- Cheat Sheet
- WikiBook

This notebook tries to give a first introduction to Julia, featuring standard features which can be found in other languages as well. More Julia-specific things come later.

Open Source

- Julia is an Open Source project started at MIT
- Julia itself is distributed under an MIT license
    - packages often have different licenses
- Development takes place on github
- The Open Source paradigm corresponds well to the fundamental requirement that scientific research should be transparent and reproducible

How to install and run Julia

- Installation:
  - Download from julialang.org (recommended by Julia creators)
  - Long term support (LTS) Version: 1.6
  - Current stable version: 1.9.3 (recommended for this course)
- Workflows:
  - Pluto notebooks in the browser
  - From command line: edit source code in any editor
  - Julia plugin of Visual Studio code editor
  - Jupyter notebooks in the browser

Standard number types

- Julia is a strongly typed language, so any variable has a type.
- Standard number types allow fast execution because they are supported in the instruction set of the processors
- Default types are autodected from expression
- The `typeof` function allows to detect the type of a variable
- The `sizeof` function detects the size in bytes of a variable (1 byte = 8 bit)

Integers

Integer variables are used to represent integer numbers.

```
i = 10
```

```
1 i=10
```

```
Int64
```

```
1 typeof(i)
```

```
8
```

```
1 sizeof(i)
```

Besides of the default `Int64` type, Julia knows `Int8`, `Int16`, `Int32`

```
10
```

```
1 j::Int32=10
```

```
Int32
```

```
1 typeof(j)
```

```
4
```

```
1 sizeof(j)
```

Floating point numbers

Floating point variables are used to represent real numbers up to some precision.

```
x = 10.0
```

```
1 x=10.0
```

```
Float64
```

```
1 typeof(x)
```

```
8
```

```
1 sizeof(x)
```

There is also `Float16`, `Float32`

```
20.614285714285717
```

```
1 y::BigFloat=144.3/7
```

```
40
```

```
1 sizeof(y)
```

Vectors

- Elements of a given type stored contiguously in memory
- Vectors and 1-dimensional arrays are the same
- Vectors can be created for any element type
- Element type can be determined by `eltype` method
- Indices count from 1!

Construction by explicit list of elements:

```
v1 =  [1, 2, 3, 4, 5, 6]
```

```
1 v1=[1,2,3,4,5,6]
```

```
Int64
```

```
1 eltype(v1)
```

```
Vector{Int64} (alias for Array{Int64, 1})
```

```
1 typeof(v1)
```

```
48
```

```
1 sizeof(v1)
```

We can create a vector of floats:

```
v1f =  [1.0, 2.0, 3.0, 4.0]
```

```
1 v1f=Float64[1,2,3,4]
```

If one element in the initializer is float, the vector becomes float:

```
v2 =  [1.0, 2.0, 3.0, 4.0]
```

```
1 v2=[1.0,2,3,4,]
```

```
Vector{Float64} (alias for Array{Float64, 1})
```

```
1 typeof(v2)
```

Other Vector constructors

Create vectors of zeros, ones, constant, random or uninitialized values:

```
[0.0, 0.0, 0.0, 0.0, 0.0]
```

```
1 zeros(Float32,5)
```

```
[1.0, 1.0, 1.0, 1.0, 1.0]
```

```
1 ones(5)
```

```
[17.0, 17.0, 17.0, 17.0, 17.0, 17.0, 17.0, 17.0, 17.0, 17.0]
```

```
1 fill(17.0,10)
```

```
[0.9316, 0.843, 0.5864, 0.1265, 0.798, 0.856, 0.1621, 0.4028, 0.29, 0.08496, 0.4336, 0.72
```

```
1 rand(Float16,20)
```

```
[1.0f-45, 0.0, 3.0f-45, 0.0, NaN, NaN, NaN, NaN, 1.0f-45, 0.0]
```

```
1 Vector{Float32}(undef,10)
```

Ranges

- Ranges describe sequences of numbers and can be used in loops, array constructors etc.
- They contain the recipe for the sequences, not the full data.

```
r1 = 1:10
```

```
1 r1=1:10
```

```
UnitRange{Int64}
```

```
1 typeof(r1)
```

Collect the sequence from a range into a vector:

```
w1 =  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
1 w1=collect(r1)
```

```
Vector{Int64} (alias for Array{Int64, 1})
```

```
1 typeof(w1)
```

Add a step size to a range:

```
r2 = 1.0:0.8:9.8
```

```
1 r2=1:0.8:10
```

```
StepRangeLen{Float64, Base.TwicePrecision{Float64}, Base.TwicePrecision{Float64}, Int64}
```

Create a vector from a list comprehension containing a range:

```
[0.841471, 0.891207, 0.932039, 0.963558, 0.98545, 0.997495, 0.999574, 0.991665, 0.973848,
```

```
1 [sin(i) for i=1:0.1:5]
```

Create a random vector of given size:

```
[0.802005, 0.811877, 0.0531582, 0.314559, 0.0110408, 0.0734351, 0.429833, 0.121725, 0.907
```

```
1 rand(10)
```

Julia version of "linspace":

```
[0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0, 20.0]
```

```
collect(range(0,20,length=11))
```

Vector dimensions

```
v6 = [1, 3, 5, 7, 9]
```

```
v6=collect(1:2:10)
```

`size` is a tuple of dimensions

```
(5)
```

```
size(v6)
```

`length` describes the overall length:

```
5
```

```
length(v6)
```

Subvectors

Copies of parts of vectors:

```
v7 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
v7=collect(1:10)
```

```
subv7 = [2, 3, 4]
```

```
subv7=v7[2:4]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
subv7[1]=17;v7
```

Views

```
v8 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
v8=collect(1:10)
```

```
subv8 =  view(::Vector{Int64}, 2:4): [2, 3, 4]
```

```
1 subv8=view(v8,2:4)
```

```
[1, 20, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
1 subv8[1]=20;v8
```

The `@views` macro can turn a copy statement into a view

```
v9 =  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
1 v9=collect(1:10)
```

```
@views subv9 =  view(::Vector{Int64}, 2:4): [2, 3, 4]
```

```
1 @views subv9=v9[2:4]
```

```
[1, 29, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
1 subv9[1]=29; v9
```

Dot operations

- Element-wise operations on vectors

```
v10 =
[0.0, 0.314159, 0.628319, 0.942478, 1.25664, 1.5708, 1.88496, 2.19911, 2.51327, 2.82743,
```

```
1 v10=collect(0:0.1π:2π)
```

```
[0.0, 0.309017, 0.587785, 0.809017, 0.951057, 1.0, 0.951057, 0.809017, 0.587785, 0.309017
```

```
1 sin.(v10)
```

```
[100.0, 100.314, 100.628, 100.942, 101.257, 101.571, 101.885, 102.199, 102.513, 102.827,
```

```
1 v10.+100
```

Matrices

- Elements of a given type stored contiguously in memory, with two-dimensional access
- Matrices and 2-dimensional arrays are the same
- Julia matrices are stored column-major

Zero initialization:

```
(5×6 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0, 5×6 Matrix{Float64}:
 1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0, 5×6 Matrix{Int64}:
 17  17  17  17  17  17
 17  17  17  17  17  17
 17  17  17  17  17  17
 17  17  17  17  17  17
 17  17  17  17  17  17, 5×6 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0)
```

```
1 zeros(5,6), ones(5,6), fill(17,5,6), rand(5,6)
```

`undef` initialization:

```
3×3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0
```

```
1 Matrix{Float64}(undef,3,3)
```

List comprehension:

```
m3 = 6×5 Matrix{Float64}:
       0.367879    0.606531    1.0        1.64872    2.71828
      -0.153092   -0.252406   -0.416147  -0.68611   -1.1312
      -0.240462   -0.396455   -0.653644  -1.07768   -1.77679
       0.353227    0.582373    0.96017    1.58305    2.61001
      -0.0535265  -0.0882502  -0.1455    -0.239889  -0.39551
      -0.308677   -0.508923   -0.839072  -1.3834    -2.28083
```

```
1 m3=[cos(x)*exp(y) for x=0:2:10, y=-1:0.5:1]
```

The size of a matrix is the tuple of the two matrix dimensions:

```
(6, 5)
```

```
1 size(m3)
```

The length of a matrix is the length of the contiguous storage array in memory:

```
30
```

```
1 length(m3)
```

Linear Algebra

```
1 using LinearAlgebra
```

Let us create some linear algebra objects:

```
n = 10
```

```
1 n=10
```

w =

```
[0.79103, 0.074211, 0.260048, 0.635886, 0.583295, 0.582967, 0.828953, 0.408032, 0.324305
```

```
w=rand(n)
```

u =

```
[0.970766, 0.23551, 0.580186, 0.306903, 0.941777, 0.238796, 0.315838, 0.841754, 0.869148
```

```
u=rand(n)
```

A =

```
10×10 Matrix{Float64}:
 0.356184  0.155497  0.905029   0.766034     …  0.175292  0.145217  0.411488  0.930428
 0.50167   0.418461  0.927313   0.759872        0.712825  0.767963  0.508279  0.538989
 0.690751  0.628531  0.194641   0.572231        0.119289  0.843545  0.747226  0.620329
 0.108621  0.308078  0.0374938  0.739576        0.416549  0.303715  0.8404    0.83686
 0.520857  0.472584  0.0901283  0.195999        0.28531   0.658128  0.274633  0.907745
 0.889918  0.875447  0.0680537  0.594424     …  0.767033  0.784089  0.842741  0.297943
 0.916277  0.765296  0.129485   0.404422        0.851457  0.390191  0.899804  0.936364
 0.682388  0.724524  0.996825   0.00581933      0.702147  0.388026  0.476688  0.788852
 0.851466  0.719339  0.48402    0.762046        0.811837  0.130149  0.205123  0.582437
 0.123726  0.474875  0.770313   0.496023        0.86314   0.139631  0.4732    0.588388
```

```
A=rand(n,n)
```

Mean square norm $||u||_2 = \sqrt{\sum_{i=1}^{n} u_i^2}$:

```
1.995226339836786
```

```
norm(u)
```

Dot product: $(u, w) = \sum_{i=1}^{n} u_i w_i$:

```
2.745693207176765
```

```
dot(u,w)
```

Matrix-vector product Au:

```
[2.78393, 3.67494, 3.61481, 1.94355, 2.12192, 3.51126, 3.79428, 3.19695, 3.27446, 2.58659
```

```
A*u
```

Inverse:

```
10×10 Matrix{Float64}:
 -0.0406678   1.0837    -0.567468    0.363366   …   0.703455   0.660656  -2.48433
 -1.72463    -1.40957    1.78806     1.68256        1.78904    1.75238    0.262666
  1.30324    -0.432413  -0.519769   -0.693221       0.345331  -0.680638   0.431768
  0.708955   -0.262831  -0.324016    0.375979      -0.550781   0.535498  -0.0110013
 -1.79552     1.13957    1.51155     0.709386       0.353635   0.951844  -0.517988
  2.39696    -1.56257   -0.856349   -2.42225    …  -1.69382   -1.60821    2.1948
 -0.534024    1.02373   -1.0993     -0.233987      -0.585775  -0.156158   0.34928
  0.802437    0.168868  -0.333929   -1.54871       -1.25221   -1.35159    1.19936
  0.660793   -0.289653   0.0362941   0.0436068      0.117375  -1.1486     0.0595613
 -0.254865    0.046398  -0.0575594   0.687141       0.267167   0.297528  -0.329378
```

```
1 inv(A)
```

```
b =
 [0.489415, 0.187954, 0.818997, 0.300858, 0.178752, 0.766853, 0.384298, 0.945711, 0.180011
```

```
1 b=rand(10)
```

Solve $Ax = b$:

```
 [-0.527119, 1.58663, 0.964242, 0.0481085, -0.689001, 0.657979, -1.13877, -0.0026514, 0.8
```

```
1 A\(b)
```

```
 [-0.527119, 1.58663, 0.964242, 0.0481085, -0.689001, 0.657979, -1.13877, -0.0026514, 0.8
```

```
1 inv(A)*b
```

Control structures

Conditional execution:

```
cond1 = true
```

```
1 cond1=true
```

```
cond2 = true
```

```
1 cond2=true
```

```
"cond1"
```

```
1 if cond1
2     "cond1"
3 elseif cond2
4     "cond2"
5 else
6     "nothing"
7 end
```

'?' operator for writing shorter code (borrowed from C):

"cond1"

```
cond1 ? "cond1" : "nothing"
```

For loop:

```
for i ∈ 1:5
        println(i)
end
```

```
1
2
3
4
5
```

Preliminary exit of a loop:

```
for i in 1:10
        println(i)
        if i>5
            break
        end
end

```

```
1
2
3
4
5
6
```

Skipping iterations:

```
    for i in 1:10
      if i==5
        continue
      end
      println(i)
    end
```

```
1
2
3
4
6
7
8
9
10
```

Functions

- All arguments to functions are passed by reference
- Function name ending with ! indicates that the function mutates at least one argument, typically the first. This is a convention, not a syntax rule.
- Function objects can be assigned to variables

Structure of function definition

```
function func(req1, req2, opt1=dflt1, opt2=dflt2; kw1=dflt3, kw2=dflt4)
   # do stuff
    return out1, out2, out3
end
```

- Required arguments are separated with a comma and use the positional notation
- Optional arguments have a default value in the signature and are positional
- Keyword arguments follow after the `;` and have default values as well, they can be invoked in arbitrary sequence
- Return statement is optional, by default, the result of the last statement is returned
- Multiple outputs can be returned as a tuple, e.g., return out1, out2, out3.
- Return `nothing` if you would like to avoid returning data

A function with one requiered and two optional arguments:

```
func_with_optional_args (generic function with 3 methods)
```

```
function func_with_optional_args(x,y=9,z=9)
    100*x+10*y+z
end
```

```
199
```

```
func_with_optional_args(1)
```

```
159
```

```
func_with_optional_args(1,5)
```

```
178
```

```
func_with_optional_args(1,7,8)

```

A function with one required and two keyword arguments:

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

```
function func_with_keyword_args(x;y=9,z=9)
    100*x+10*y+z
end
```

```
199
```

```
func_with_keyword_args(1)
```

```
190
```

```
func_with_keyword_args(1; z=0)
```

```
109
```

```
func_with_keyword_args(1; y=0)
```

One line function definition:

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

```
g(x)=exp(sin(x))
```

```
1.151562836514535
```

```
g(3)
```

Nested function definitions:

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

```
function outerfunction(n)
    function innerfunction(i)
        println(i)
    end
    for i=1:n
        innerfunction(i)
    end
end
```

```
outerfunction(13)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

Functions are variables, too:

```
1.151562836514535
```

```
h=g; h(3)
```

Functions as function parameters:

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

```
F(f,x)= f(x)
```

```
1.151562836514535
```

```
F(g,3)
```

Anonymous functions (convenient in function parameters):

```
0.1411200080598672
```

```
F(x -> sin(x),3)
```

Do-block syntax: the body of first parameter is in the `do ... end` block:

```
1.151562836514535
```

```
F(3) do x
    exp(sin(x))
end
```

Functions and vectors

Dot syntax can be used to make any function work on vectors:

```
v11 =  [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
```

```
v11=collect(0:0.1:1)
```

```
[1.0, 1.10499, 1.21978, 1.34383, 1.47612, 1.61515, 1.75882, 1.9045, 2.04901, 2.18874, 2.3
```

```
h.(v11)
```

Map function on vector:

```
[1.0, 1.10499, 1.21978, 1.34383, 1.47612, 1.61515, 1.75882, 1.9045, 2.04901, 2.18874, 2.3
```

```
map(h,v11)
```

`mapreduce`: apply operator to each element and collect data

```
5.5
```

```
mapreduce(x->x,+,v11)
```

```
0.0
```

```
prod(v11)
```

```
5.5
```

```
mapreduce(x->x,+,v11)
```

```
5.5
```

```
sum(v11)
```

Macros

Julia allows to define macros which allow to modify Julia statements before they are compiled and executed. This capability is similar to the preprocessor in C or C++. Macro names start with `@`. Occasionally we will use predefined macros, e.g. `@elapsed` for returning the time used by some statement.

```
0.000634101
```

```
1 @elapsed inv(rand(100,100))
```

The @time macro prints time and the number of allocation used for a statement:

```
100×100 Matrix{Float64}:
   8.9436    -0.957307    -0.341852   …  -13.2524     2.34709     10.5528
   3.05956   -0.251765    -0.0883467      -4.28415    0.327861     3.23383
  -0.322667   0.0026661    0.505704        0.390092   0.491958    -0.0462207
  -0.653106   0.00707719   0.27518         0.818434   0.0520537   -0.730591
  -6.95121    0.467085    -0.0841561      10.3954    -1.48744     -8.02673
   6.77827   -0.236859    -0.327985   …   -9.22187    1.20271      6.43721
  17.9164    -1.47837     -0.612436      -25.9654     4.01356     19.6516
   ⋮                                   ⋱
  12.543     -1.03071      0.1486        -18.5143     2.92247     14.4935
 -15.6652     1.27814      0.397847   …   22.6484    -3.23655    -17.1897
 -11.0877     1.06031      0.440809       16.4171    -2.65381    -13.0272
  22.9891    -1.8501      -0.458551      -33.4489     4.97949     25.2399
   4.47995   -0.451711    -0.597444       -6.49577    0.467855     4.74648
 -12.5375     1.00937     -0.26765        18.144     -2.68861    -13.7367
```

```
1 @time inv(rand(100,100))
```

```
  0.000511 seconds (7 allocations: 207.344 KiB)
```