universität innsbruck

igs

Einführung in `Julia`

Proseminar Angewandte Mathematik für die Informatik

Stephanie Maria Autherith, BSc MSc
Hassam Ahmed Malik, BSc MSc
Juliette Louise Opdenplatz, BSc MSc
Markus Walzthöni, BSc MSc
Stefano Fogarollo, BSc MSc
Adéla Moravová, BSc MSc
Nikolaus Rauch, BSc MSc

Übersicht

- Was ist `Julia`?
- Installation
- Verwendung von `Julia`

Was ist `Julia`?

Generelles:

- REPL (read–eval–print loop)
- Dynamisches Typsystem
- Entwickelt am MIT
- Hohe Performance
- Numerisches und wissenschaftliches Rechnen

Einflüsse:

- Matlab
- Scheme
- LISP

Was ist `Julia`?

Features:

- Direkte Aufrufe von `Python`, `C/C++` und `Fortran`-Funktionen
- Integration von `Julia` auch in `Python` und `C/C++` Anwendungen
- Paralleles und verteiltes Rechnen
- Integrierte Paketverwaltung

Installation

Linux:

1. Julia von `https://julialang.org/downloads/` herunterladen
2. Entpacken und in den gewünschten Installationsordner verschieben
3. Symbolischen Link mit
   `sudo ln -s pfad/zu/julia/bin/julia /usr/local/bin/julia`
   erstellen
4. Mit dem Kommando `julia` starten

Windows:

1. Installer für Julia von `https://julialang.org/downloads/` downloaden
2. Installer ausführen
3. Mit dem Kommando `julia` starten

IDEs

Vorschläge:

- Juno `https://junolab.org/`
  (basierend auf dem Atom Editor)
- VS Codium `https://vscodium.com/`
  “Julia” und “Judy” Erweiterung
- Jupyter `https://jupyter.org/`
- Vim `https://github.com/vim/vim`
  Plugin `https://github.com/JuliaEditorSupport/julia-vim`
- ...

Variablen und Typen

`Julia` ist eine dynamisch typisierte Programmiersprache:

```
x = 5
ξ = 5 # unicode variable \xi tab
```

Explizite Typdeklaration ist für nicht-globale Variablen immernoch möglich:

```
x::Int8 = 5
y::Float64 = 3.14
z::String = "Hello World!"
```

https://docs.julialang.org/en/v1/manual/types/

Mathematische Operatoren und Konstanten

Einfache mathematische Operationen:

```
c_squared = a^2 + b^2
log2(4)
log10(100)
sin(0)
cos(0)
```

Wichtige mathematische Konstanten sind verfügbar:

```
c = 2*pi*radius
```

https://docs.julialang.org/en/v1/base/math/

Bool’sche und Gleichhheitsausdrücke

AND:

```
true && false # false
true && true  # true
```

OR:

```
true || false # true
false || false # false
```

Negation:

```
!true # false
```

Gleichheit:

```
1 > 0 # true
1 < 0 # false
1 >= 1 # true
1 <= 1 # true
1 == 0 # false
1 != 0 # true
```

Arrays und Vektoren

Es gibt viele Wege ein Array zu erzeugen:

```
a13 = [1 2 3]
a31 = [1,2,3]
```

Vektoren sind ein-dimensionale Arrays:

```
Vector{Int} == Array{Int64,1} # true
Vector{Int} == Array{Int64,2} # false
```

Vereinfachte Initialiserung:

```
ones(3)
zeros(3)
```

https://docs.julialang.org/en/v1/base/arrays/

Arrays und Matrizen

Es gibt viele Wege ein mehrdimensionales Array zu erzeugen:

```
a22 = [1 2; 3 4]
a22 = [[1, 2] [3, 4]]
a22 = [1 3; 2 4]
```

Matrizen sind zweidimensionale Arrays:

```
Matrix{Int} == Array{Int64,1} # false
Matrix{Int} == Array{Int64,2} # true
```

Die Größe eines Arrays ist ein Tupel, welches die Dimensionen des Arrays enthält:

```
size(a31) # (1,3)
size(a22) # (2,2)
```

Vereinfachte Initialisierung funktioniert auch für mehrdimensionale Arrays:

```
ones(2,2)
```

Vektoren und Matrizen (Operationen)

Transponieren oder `reshape` um Dimensionen zu ändern:

```
size(ones(3,2))   # (3,2)
size(ones(3,2)') # (2,3)
size(reshape(ones(3,2), (2,3))) # (2,3)
```

Mehr mathematische Operationen:

```
2*ones(3)               # scalar by vector
2*ones(3,3)             # scalar by matrix
zeros(3) * ones(3,3)    # vector by matrix - dimension mismatch
zeros(3)' * ones(3,3)   # vector by matrix
zeros(3,3) * ones(3,3) # matrix by matrix
ones(3,3)/0.5
```

Das `LinearAlgebra` package enthält mehr und fortgeschrittenere Operationen.

https://docs.julialang.org/en/v1/base/arrays/
https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/

Reihen (`range`) und Operationen

Definieren einer `range` von Werten:

```
range(1, stop=100)           # 1:100
range(1, step=10, stop=100) # 1:10:91
1:10:100                     # shorthand
-5.2:0.4:3.2                 # also possible in floats
```

Reihen sind keine Arrays aber man kann Arrays aus Reihen erzeugen:

```
collect(1:10)
```

Anwenden von Operationen auf jedes Element einer Reihe oder eines Arrays:

```
(1:100).+1   # apply an operation
log2.(1:100) # apply a function
```

https://docs.julialang.org/en/v1/base/math/#Base.range

Auf Elemente Zugreifen

Index startet ab `1`!
Zugriff auf Array durch direktes Indizieren von Elementen oder Indizieren ganzer Reihen/Spalten:

```
A = [1 2; 3 4]
A[1,1]    # 1
A[end]    # access last element
A[end-2]  # access second last element
A[2:end]  # access sub-array by range
A[:,1]    # access column
A[1,:]    # access row
```

Schleifen

`for`-Schleifen Syntax:

```
for iterator [in|=] [range|array]
  # do something
end
```

Verschachtelte `for`-Schleifen Syntax:

```
for i = [range|array], j = [range|array]
  # do something
end
```

`while`-Schleifen Syntax:

```
while boolean-expression == true
  # do something
end
```

https://docs.julialang.org/en/v1/manual/control-flow/#man-loops

Schleifen und Iteratoren

`for`-Schleife simultan über zwei Arrays:

```
for (i, j) in zip(a,b)
  # do something
end
```

Iterieren über jeden Index eines Arrays:

```
for i in eachindex([1 2; 3 4])
  # do something
end
```

Ausgeben von Werten als Einzeiler:

```
foreach(println, [1, 2, 3, 4])
```

https://docs.julialang.org/en/v1/base/iterators/

If(-else) Konstrukte

`if-else` Konstrukte sehen folgendermaßen aus:

```
if x < 0
  # do something
elseif x > y
  # do something else if
else
  # do something else
end
```

Inline Konditionen sind auch möglich:

```
println(x < y ? "less than" : "not less than")
```

https://docs.julialang.org/en/v1/manual/control-flow/#man-conditional-evaluation

Funktionen 1

Es gibt mehrere Wege eine Funktion zu definieren:

```
function f(a, b, ...)
  #do something
end
f(a, b, ...) = ... # a named one-liner
(a, b, ...) -> ... # an anonymous one-liner
```

Der Rückgabewert einer Funktion ist der Wert ihres letzten Ausdrucks:

```
function add(a,b)
  a+b
end
add(1,2) # a usual function call
+(1,2) # note: operators can also be used using function call s
```

https://docs.julialang.org/en/v1/manual/functions/

Funktionen 2

Explizite Rückgabe ist auch möglich und manchmal nötig:

```
function divide(a,b)
  if (b == 0)
    return NaN
  end
  a/b
end
```

Manchmal ist es nötig Parameter und Rückgabe Typen zu definieren:

```
function add(a::Float64,b::Float64)::Float64
  a+b
end
```

Map/Reduce und Broadcasting

Anwenden einer Funktion über ein Array oder `range` mit `map`:

```
map(sqrt, 1:100)
map(x->x*x, 1:100)
```

Ähnlich aber nicht equivalent zur "dot notation" oder `broadcast` Funktion:

```
sqrt.(1:100)      # broadcast(sqrt, 1:100)
(x->x*x).(1:100)  # broadcast(x->x*x, 1:100)
```

Broadcasting funktioniert auch über mehrere Arrays/Reihen:

```
(1:10) .* (1:10)           # broadcast(*, 1:10, 1:10)
((x,y)->x*y).(1:10, 1:10)  # broadcast((x,y) -> x*y, 1:10, 1:10)
```

`reduce` zum Aggregieren von Werten über ein Array oder einer Reihe:

```
reduce(+, 1:100)/100
```

Arbeiten mit `julia`

Starten der `julia`-Shell:

```
$julia
```

Ausführen einer `julia`-Datei:

```
$julia main.jl
```

Einbinden von einer `julia`-Datei:

```
include("my-source.jl")
```

Packages

Installeren und Laden eines `package` im Code:

```
import Pkg
Pkg.add("GR")
import GR
using GR
```

Hinzufügen eines `package` aus der `julia`-Shell:

```
julia> ] # you will automatically go to pkg mode
(@v(x.y) pkg)>add GR
(@v(x.y) pkg)> # backspace will get you back to julia mode
```

Plotting mit `Plots` und `GR`

```
using Plots
Plots.GRBackend() # not necessary as GR is default backend

x = 0 : π / 10 : 3 * π;
y1 = x .+ 1.5*sin.( x );
y2 = x .+ 1.5*sin.( π / 2 .+ x );
y3 = x .+ 1.5*sin.( 3 * π / 2 .+ x );

plt = Plots.plot(x, y1, color=:green, label="base",
        title="First function plot (sin)");
plot!(plt, x, y2, color=:blue, label="Pi/2");
plot!(plt, x, y3, color=:yellow, label="3Pi/2")
```

https://gr-framework.org/julia.html

Tutorials - Videos - ...

- Julia Youtube channel
  `https://www.youtube.com/user/JuliaLanguage`
- MIT Julia tutorial
  `https://www.youtube.com/playlist?list=PLP8iPy9hna6Si2sjMkrPY-wt2mEouZgaZ`
- Think Julia: How to Think Like a Computer Scientist
  `https://benlauwens.github.io/ThinkJulia.jl/latest/book.html`
- Julia Online Training
  `https://julialang.org/learning/`

Vielen Dank für Ihre Aufmerksamkeit!

Stephanie Maria Autherith, BSc MSc
Hassam Ahmed Malik, BSc MSc
Juliette Louise Opdenplatz, BSc MSc
Markus Walzthöni, BSc MSc
Stefano Fogarollo, BSc MSc
Adéla Moravová, BSc MSc
Nikolaus Rauch, BSc MSc