# Julia

## Motivation

- frei
- offen
- leicht
- schnell
- dynamisch



## Getting started

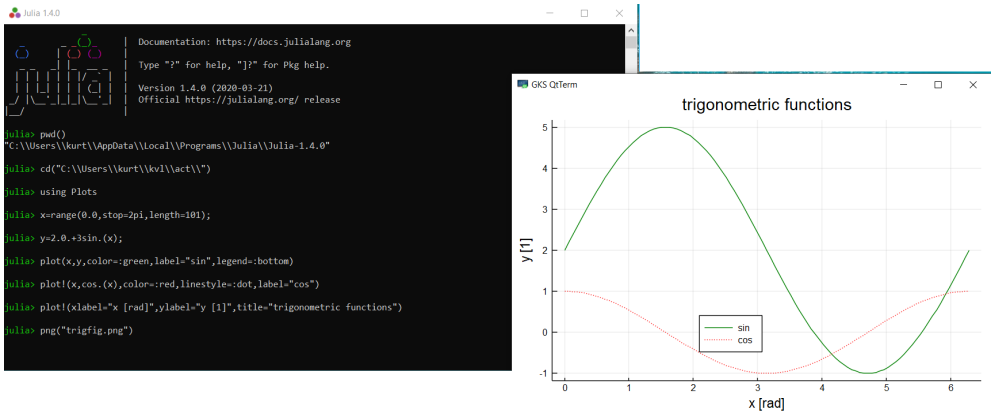
- windows, linux, macOS
  download installer/archive from julialang.org
  unter ubuntu: `sudo apt-get julia` gibt UraltVersion!
  aktuelle stabile Version: 1.9.0 (Mai 2023)
- packages
  Pkg.add(name), using name
  Beispiel: `add Plots; using Plots; plot(sin,-pi,pi)`
  inzwischen wird nach `using Plots;`
  automatisch Installation angeboten – sofern nötig



## Arbeitsweisen

- REPL-Modus (inklusive help und pkg)
- Skripte mit `include`
- GUI dank VisualStudioCode, vorher: atom (hackable editor)
- Notebooks unter Jupyter mit IJulia

# REPL

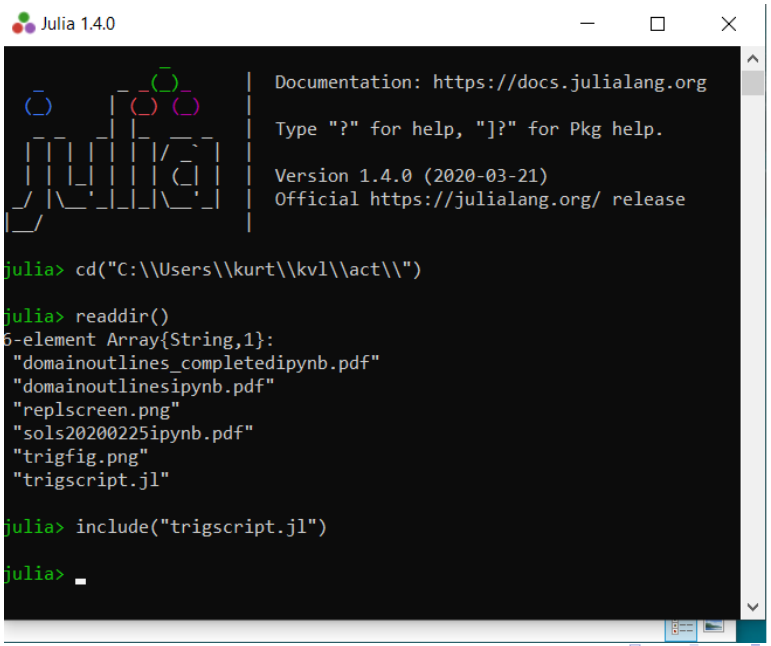



# Remarks zu REPL

- REPL steht für: read-eval-print loop
- Ausführung mit Enter
- Hilfe mit ?
- package-mode mit ] (add Plots etc.)
- copy und paste ggfs. mit rechter Mausetaste




# Include




# Remarks zu include

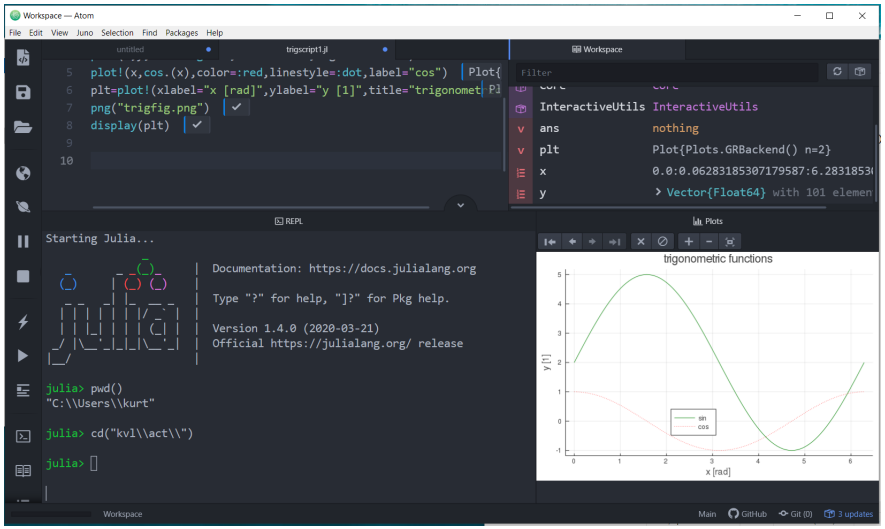
- im REPL Skripte mit include ausführen
- Ausgaben, u.a. Plots, werden nur auf Wunsch gezeigt
- Verzeichnis wird nicht automatisch gewechselt

## Julia – Example

```
using Plots

x=range(0.0,stop=2pi,length=101);
y=2.0.+3sin.(x);

plot(x,y,color=:green,label="sin",legend=:bottom)
plot!(x,cos.(x),color=:red,linestyle=:dot,label="cos")

plt=plot!(xlabel="x [rad]",ylabel="y [1]",
title="trigonometric functions")

png("trigfig.png")
display(plt)
```



## atom






## Remarks zu atom

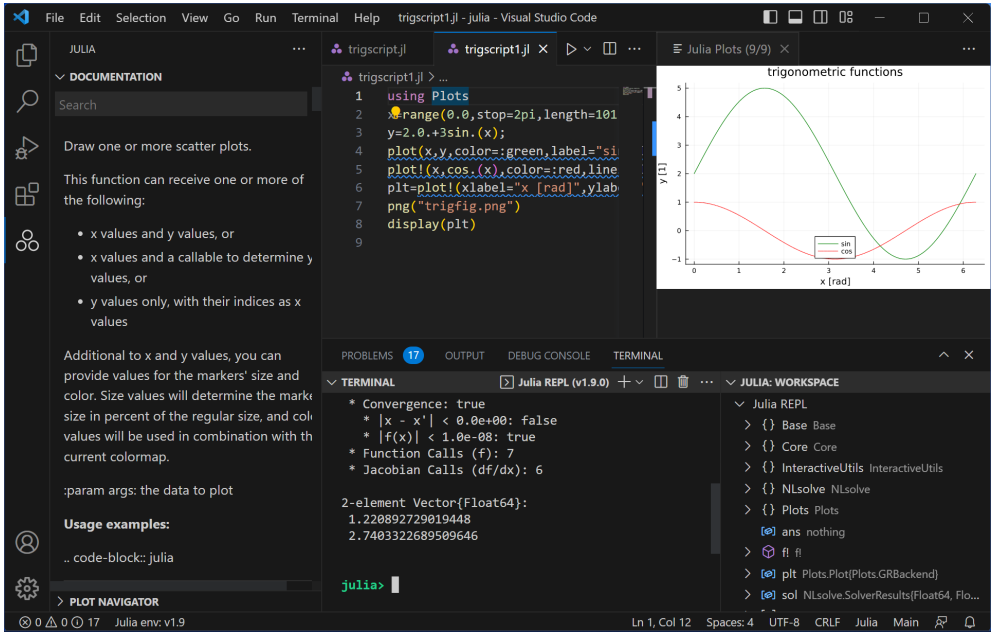
- unter atom wird Ausführung von Skript blockweise (Shift+Enter) oder gesamt (Crtl+Shift+Enter)
- Ausgaben ggfs. hinter jedem Block
- Plots docked in speziellem Unterfenster, zum durchblättern
- Workspace einsehbar
- es gibt keinen clear-Befehl



## Remarks zu Visual Studio Code

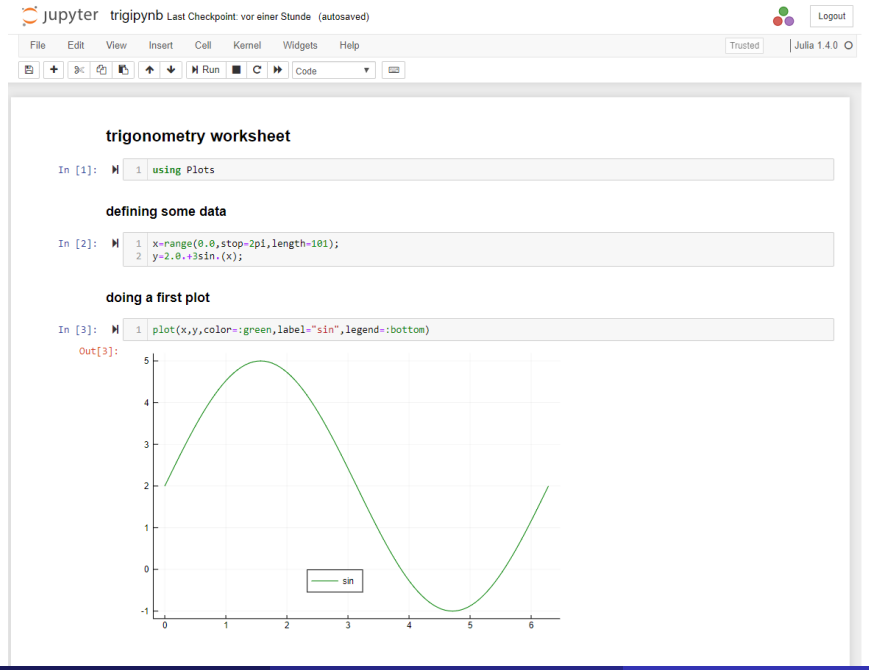
- löst atom ab
- nicht zu verwechseln mit C++Entwicklungsumgebung von MS
- `code.visualstudio.com`
- Plugins für julia und jupyter holen
- Workspace, PlotHistorie, Probleme, Doku ...
- vielfältige Nutzung, u.a. python, LaTeX...

# VSC-Screenschot



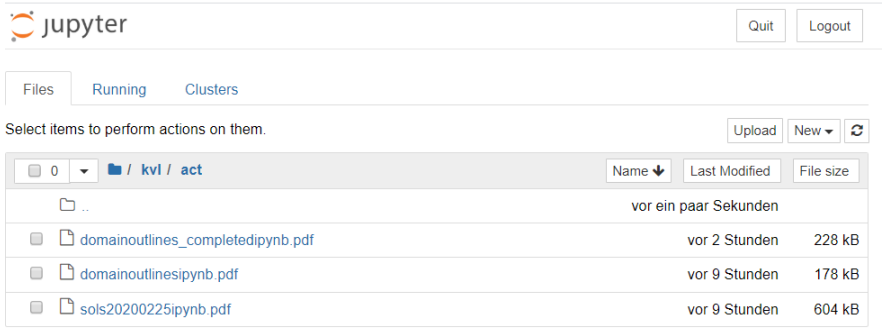



# Jupyter






# Jupyter – FileManager

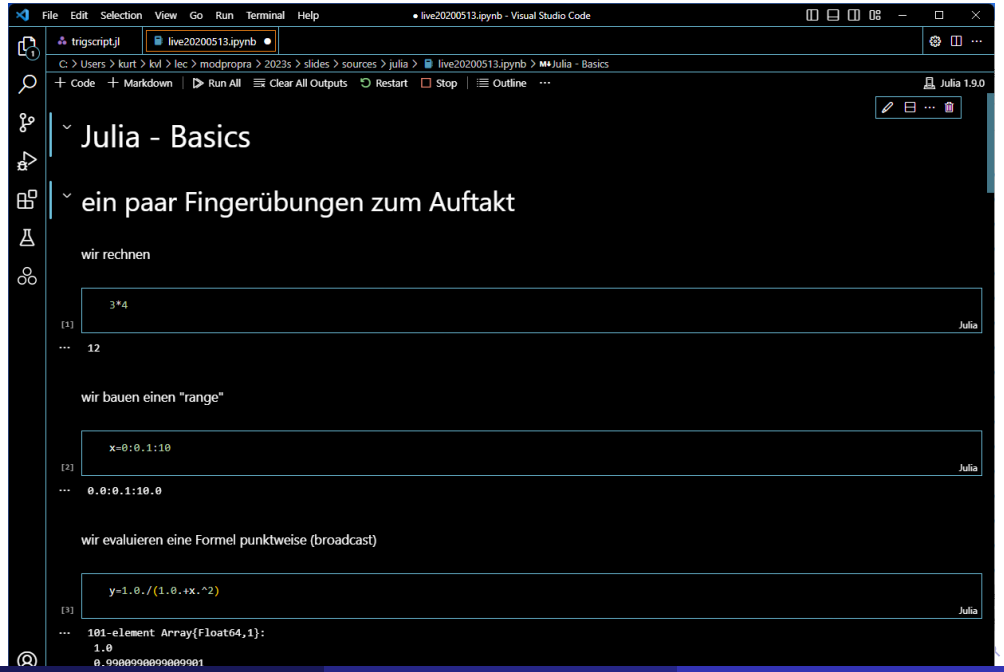





# Remarks zu IJulia

- Start mit: `using IJulia; notebook()`
- Maple-ähnliche Oberfläche im Standardbrowser
- neues oder existierendes notebook öffnen
- Text (inklusive LaTeX) und Input eingeben
- Output im Worksheet
- Ctrl+P zwecks Ausdruck

## Jupyter in VSC






## Unterschiede zu Matlab

- Kommentare mit `#`
- Klammern
  `x=[1;2;3], x[2]+=7`
- Ranges sind keine Vektoren
  `y=1:3; y[3]`
  `y[3]=9`
- Vektoren sind keine Matrizen
- Arrays zuweisen mit `x=copy(y);` sonst zeigen `x` und `y` auf das selbe Object
- Operatoren müssen broadcasted werden
  `x.+y`
- Pakete müssen geladen werden
  `eigvals([1 2; 3 4])` erst nach: `using LinearAlgebra`
- Funktionen sehr user-friendly defined
  `f(x)=sqrt(x)-2x^3`
- Deklaration von Variablen-Typen
- Befehl *meshgrid* fehlt (anfangs)



## Erste Fingerübungen

- Plots
  (2d, contour, 3d, Animation)
- Nichtlineare Gleichungen lösen
  (Gleichgewichtszustand)
- LGS Lösen
  (zum Beispiel Kräftegleichgewicht)
- Minimum finden (stationäre Lösung)
- ODE lösen
  (Oszillator)
- Bewegungsgleichungen aufstellen
  (symbolisches Rechnen, Formelmanipulation)



## Alternativen bei Graphik

many packages – As you like it

- Plots
  Pkg.add(Plots)
  using Plots
- PyPlot
  best for Python fans
- Gaston
  based on Gnuplot
- Winston
  looks pretty poor
- GR
  ist der Standard bei Plots

Bemerkung: Man kann Frontend wie Backend wechseln, also etwa Plots so nutzen, als hätte man PyPlot geladen.

## Alternativen bei ODEs

- DifferentialEquations
  Maß der Dinge
- ODE
  wishlist, still growing project
- OrdinaryDifferentialEqs
- Sundials
  wraps libraries od Lawrence Livermore National Laboratory
  SUite of Nonlinear and DIfferential/Algebraic equation Solvers



## Angestaubte Variante

```
using Sundials, Winston
# a well-known right-hand side
function f(t,y,ydot)
   mu = 2.5
   ydot[1] = y[2]
   ydot[2] = mu*(1-y[1]^2)*y[2]-y[1]
ydot
end
# time range
t = [0:.01:10.0;]
# initial state
y0 = [1.0, 3.0]
# call of solver
res = Sundials.cvode(f, y0, t)
# plotting results
plot(res[:,1],res[:,2])
```



## Moderne Variante – skalare ODE (Zerfall)

```
using DifferentialEquations, Plots;

# Funktion mit Rueckgabe der Ableitung
f(y,p,t)=p*y;

# Zusammenstellen des Cauchyproblems (AWP)
deqprbl=ODEProblem(f,3.0,(0.0,12.0),-0.33)

# Loesung des Problems
solu=solve(deqprbl)

# Visualisierung
plot(t->solu(t),0,12,legend=false,title="Loesung des AWP")
scatter!(solu.t,solu.u,xlabel="Zeit t",ylabel="Menge y(t)")
```

Bemerkung: Übergabe des Parameters, Markierung der vom Solver gewählten Schritte.

Achtung: Reihenfolge der Argumente bei `ODEProblem` und der rechten Seite



## Moderne Variante – System (Pendel)

```
function f!(yp,y,p,t)
   g=p[1]; l=p[2];
   yp[1]=y[2];
   yp[2]=-g*sin(y[1])/l;
   return;
end;

y0=[0.0,6.26]; tspan=(0.0,12.0); p=(9.81,1.0);
pendprob=ODEProblem(f!,y0,tspan,p)

pendsol=solve(pendprob,reltol=1.0e-4)

plot(t->pendsol(t)[1],0,12,label="Winkel",
xlabel="Zeit t")
plot!(t->pendsol(t)[2],0,12,label="Spin",
ylabel="Winkel und Spin")
plot(t->pendsol(t)[1],t->pendsol(t)[2],0,12,leg=false,
title="Phasenportrait",xlabel="Winkel",ylabel="Spin")
```

Bemerkung: Hauptaufwand ist Aufstellen der rechten Seite, zwecks Lösung reicht ein knappes `solve`, der meiste Text dient der grafischen Darstellung der Lösung.

# Optimierungsbeispiel

```
using Optim, Plots;

rosenbrock(x) = (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2;

result = optimize(rosenbrock, zeros(2), BFGS());
println("the candidate minimizer is: $(result.minimizer)")
println("the minimum is: $(result.minimum)")
println("the number of steps was: $(result.iterations)")

result.f_calls, result.g_calls, result.time_run,
   result.x_relchange, result.f_abschange

surface(-1:0.1:1.2,-1:0.1:1.2,(x,y)->rosenbrock([x,y]),
        colorbar=false,
        color=color=cgrad([:blue,:red]),
        camera=(130,80),
        xlabel="\\xi_1", ylabel="\\xi_2")
```