Advanced Topics from Scientific Computing
TU Berlin Winter 2022/23
Notebook 05
Jürgen Fuhrmann

Plotting & visualization in Julia

Human perception is much better adapted to visual representation than to numbers

Purposes of plotting:

- Visualization of research results for publications & presentations
- Debugging + developing algorithms
- "In-situ visualization" of evolving computations
- Investigation of data
- 1D, 2D, 3D, 4D data

PyPlot

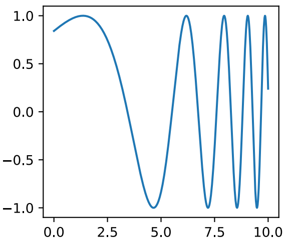
- PyPlot.jl: Interface to python/matplotlib
    - realization via PyCall.jl
    - Full functionality of matplotlib
    - also as backend for Plots.jl
    - Problem: slow - most code in python, no support for GPU acceleration
- Resources:
    - Julia package
    - Julia examples
    - Matplotlib documentation

```
import PyPlot
```

We can choose the way the plot is created: in the browser it can make sense to create it as a vector graphic in svg format. The alternatice is png, a pixel based format.

```
PyPlot.svg(true);
```

How to create a plot ?



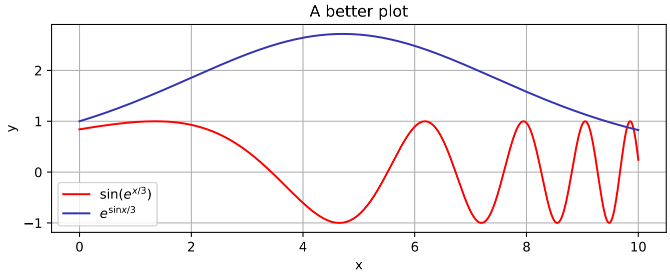
```
let
    X=collect(0:0.01:10)
    PyPlot.clf() # Clear the figure
    PyPlot.plot(X,sin.(exp.(X/3))) # call the plot function
    figure=PyPlot.gcf() # return figure to Pluto
end
```

Instead of a `begin/end` block we used a `let` block. In a let block, all new variables are local and don't interfer with other pluto cells.

This plot is not nice. It lacks:

- orientation lines ("grid")
- title
- axis labels
- label of the plot
- size adjustment

```
using LaTeXStrings
```

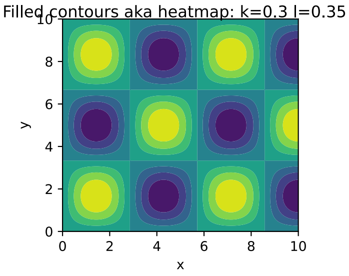


```
let
    X=collect(0:0.01:10)
    PyPlot.clf()  # clear plot
    PyPlot.plot(X,sin.(exp.(X/3)),
        label=L"$\sin(e^{x/3})$", color=:red) # Plot with LaTeX label
    PyPlot.plot(X,exp.(sin.(X/3)),
        label=L"$e^{\sin x/3}$",color=(0.2,0.2,0.7)) # Plot with label
    PyPlot.legend(loc="lower left") # legend placement
    PyPlot.title("A better plot") # The plot title
    PyPlot.grid() # add grid lines to the plot
    PyPlot.xlabel("x") # x axis label
    PyPlot.ylabel("y") # y axis label
    figure=PyPlot.gcf() # obtain figure from the plot
    figure.set_size_inches(8,3) # adjust size 1 inch is about 100 px
    PyPlot.savefig("myplot.png") # save figure to disk
    figure # return figure
end
```

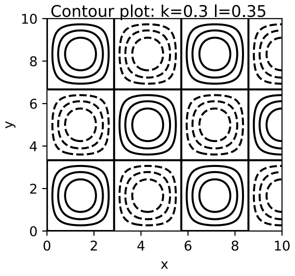
Thanks to the LaTeXStrings package, we can use $\LaTeX$ math strings in plot labels here, we just need to prefix the strings with "L".

Plotting 2D data

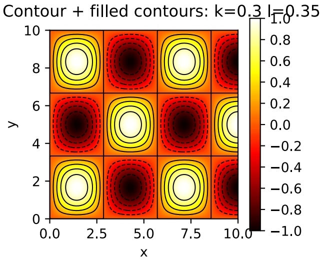
k: l:



```
let
    PyPlot.clf()
    X=collect(0:0.05:10)
    Y=X
    PyPlot.suptitle("Filled contours aka heatmap: k=$(k) l=$(l)")
    F=[sin(k*π*X[i])*sin(l*π*Y[j]) for i=1:length(X), j=1:length(Y)]
    PyPlot.contourf(X,Y,F) # plot filled contours
    PyPlot.xlabel("x")
    PyPlot.ylabel("y")
    figure=PyPlot.gcf()
    figure.set_size_inches(3,3)
    figure
end
```



```
let
    PyPlot.clf()
    X=collect(0:0.05:10)
    Y=X
    PyPlot.suptitle("Contour plot: k=$(k) l=$(l)")
    F=[sin(k*π*X[i])*sin(l*π*Y[j]) for i=1:length(X), j=1:length(Y)]
    PyPlot.contour(X,Y,F,colors=:black)
    PyPlot.xlabel("x")
    PyPlot.ylabel("y")
figure=PyPlot.gcf()
    figure.set_size_inches(3,3)
    figure
end
```

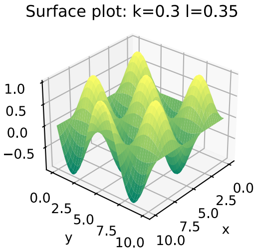


```
let
    PyPlot.clf()
    X=collect(0:0.05:10)
    Y=X
    PyPlot.suptitle("Contour + filled contours: k=$(k) l=$(l)")
    F=[sin(k*π*X[i])*sin(l*π*Y[j]) for i=1:length(X), j=1:length(Y)]
    fmin=minimum(F)
    fmax=maximum(F)
    number_of_isolines=10
    isolines=collect(fmin:(fmax-fmin)/number_of_isolines:fmax)
    cnt=PyPlot.contourf(X,Y,F,cmap="hot",levels=100)
    if fix_moire # It is not clear why this hack is necessary
        for c in cnt.collections
           c.set_edgecolor("face")
        end
    end
    axes=PyPlot.gca()
    axes.set_aspect(1)
    PyPlot.colorbar(ticks=isolines)
    PyPlot.contour(X,Y,F,colors=:black,linewidths=0.75,levels=isolines)
    PyPlot.xlabel("x")
    PyPlot.ylabel("y")
    figure=PyPlot.gcf()
    figure.set_size_inches(3,3)
    figure
end
```

Remove the moire in the plot: ☑

This occurs in `contourf` when we use many colors to make a smooth impression.

α: β:



```
let
    PyPlot.clf()
    X=collect(0:0.05:10)
    Y=X
    PyPlot.suptitle("Surface plot: k=$(k) l=$(l)")
    F=[sin(k*π*X[i])*sin(l*π*Y[j]) for i=1:length(X), j=1:length(Y)]

    PyPlot.surf(X,Y,F,cmap=:summer) # 3D surface plot
    ax=PyPlot.gca(projection="3d")  # Obtain 3D plot axes
    ax.view_init(α,β) # Adjust viewing angles


    PyPlot.xlabel("x")
    PyPlot.ylabel("y")
    figure=PyPlot.gcf()
    figure.set_size_inches(3,3)
    figure

end
```

There are analogues for `contour` `contourf` and `surf` on triangular meshes which will be discussed once we get there in the course.

Plots

```
using Plots
```

Plots.jl: General purpose plotting package with different backends

- GPU support via default `gr` backend (based on "old" OpenGL)
- Support of interactivity in the browser via `plotly` backend
- precompilation time significantly improved over the last 2 years
- Problem: up to now no good support for triangulations

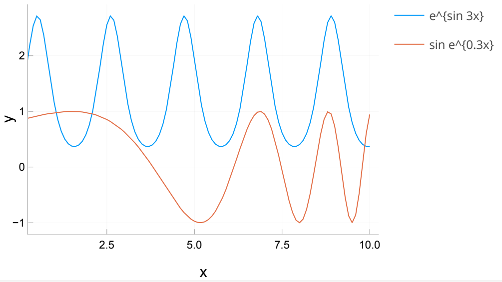
In Pluto it is best to use the plotly interface. Plotly is a Javascript library for plotting which is quite good and all kinds of x-y plots.

```
PlotlyBackend()
```

```
Plots.plotly() # Choose backend:  gr in REPL, plotly in notebook
```

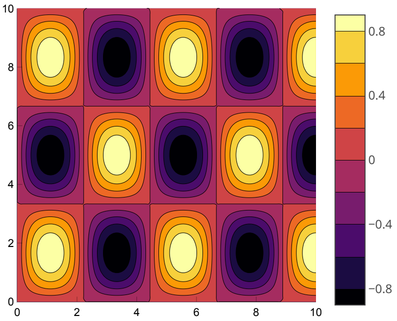
```
X =
 [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1
```

```
X=collect(0:0.1:10)
```

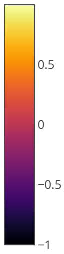


```
let
    p=Plots.plot(size=(500,300),xlabel="x",ylabel="y",)
    Plots.plot!(p, X,exp.(sin.(3X)),label="e^{sin 3x}")
    Plots.plot!(p, X,sin.(exp.(0.3X)),label="sin e^{0.3x}")
end
```

```
F=[sin(k1*π*X[i])*sin(l1*π*X[j]) for i=1:length(X), j=1:length(X)];
```



```
Plots.contour(X,X,F,fill=true,size=(400,350))
```



```
Plots.surface(X,X,F,size=(300,300))
```

Makie

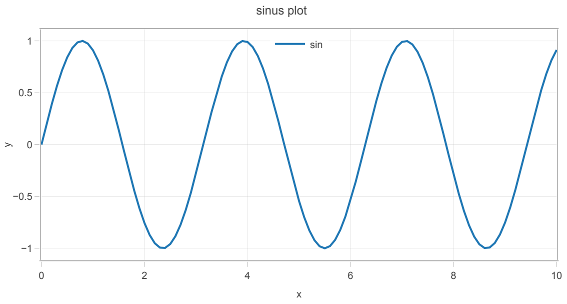
- GLMakie.jl
    - GPU based plotting using modern OpenGL - in fact the only package I know (regardless of Julia) besides of vtk.
    - very good plot performance
    - Problem: still under development, long precompilation time
- WGLMakie.jl maps Makie API to three.js, can be used from the browser
    - Problem: not very stable in the moment
    - Complicated to use in Pluto

Due to long loading time I do not show examples here.

PlutoVista

I created PlutoVista.jl for fast plotting in pluto notebooks. For 1D plots, PlutoVista calls back to Plotly.js, and for 2D/3D plots it uses vtk.js, a visualization library for grid and volume data using WebGL as backend.

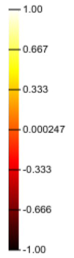
```
using PlutoVista
```



```
PlutoVista.plot(X,sin.(2X),xlabel="x",ylabel="y",label="sin",legend=:ct, resolution=(600,300), title="sinus plot")
```

```
F2=[sin(k2*π*X[i])*sin(l2*π*X[j]) for i=1:length(X), j=1:length(X)];
```

k: l:



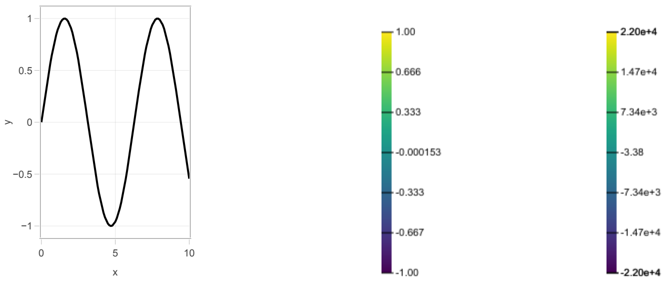
```
PlutoVista.contour(X,X,F2,size=(400,350),levels=5,colormap=:hot)
```

GridVisualize.jl

```
begin
    using GridVisualize
    using ExtendableGrids
end
```

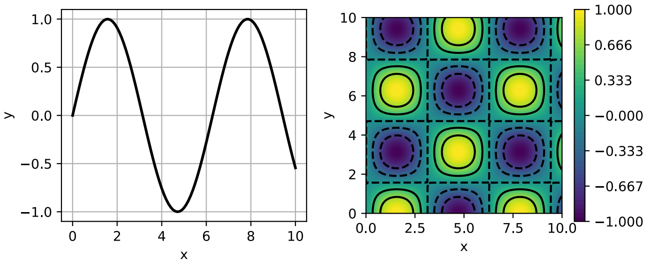
The GridVisualize.jl package focuses on PlutoVista (for notebooks) and GLMakie (from REPL) as backends. PyPlot and Plots are supported as well, but with less functionality.

It is tailored to the visualization of solutions of partial differential equations on 1D/2D/3D grids (of simplices). The idea is to allow the same syntax for different space dimensions.

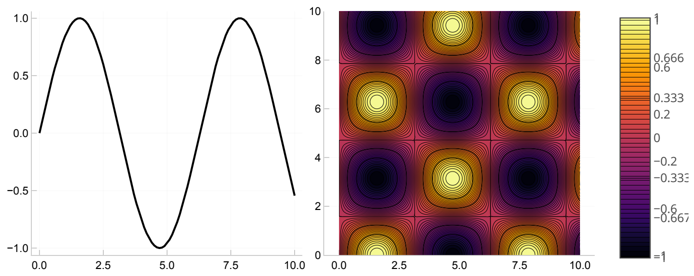


```
let
    vis=GridVisualizer(;Plotter=PlutoVista,layout=(1,3),size=(700,300))

    g1=simplexgrid(X)
    f1=map(sin,g1)
    scalarplot!(vis[1,1],g1,f1)

    g2=simplexgrid(X,X)
    f2=map((x,y)->(sin(x)*cos(y)),g2)
    scalarplot!(vis[1,2],g2,f2)

    g3=simplexgrid(X,X,X)
    f3=map((x,y,z)->(sin(x)*cos(y)*exp(z)),g3)
    scalarplot!(vis[1,3],g3,f3)

    reveal(vis)
end
```

By passing the backend as a parameter to the visualization calls, we can have several backends used in parallel.



```julia
let
    vis=GridVisualizer(;Plotter=PyPlot,layout=(1,2),size=(700,300))

    g1=simplexgrid(X)
    f1=map(sin,g1)
    scalarplot!(vis[1,1],X,x->sin(x))

    g2=simplexgrid(X,X)
    f2=map((x,y)->(sin(x)*cos(y)),g2)
    scalarplot!(vis[1,2],g2,f2)


    reveal(vis)
end
```



```julia
let
    vis=GridVisualizer(;Plotter=Plots,layout=(1,2),size=(700,300))

    g1=simplexgrid(X)
    f1=map(sin,g1)
    scalarplot!(vis[1,1],X,x->sin(x))

    g2=simplexgrid(X,X)
    f2=map((x,y)->(sin(x)*cos(y)),g2)
    scalarplot!(vis[1,2],g2,f2)


    reveal(vis)
end
```

```
setting arbitrary contour levels with Plotly backend is not supported; use a rang
e to set equally-spaced contours or an integer to set the approximate number of c
ontours with the keyword `levels`. Setting levels to -0.9999232575641008:0.039990
7916144022:0.9996163231560091

setting arbitrary contour levels with Plotly backend is not supported; use a rang
e to set equally-spaced contours or an integer to set the approximate number of c
ontours with the keyword `levels`. Setting levels to -0.9999232575641008:0.333256
596786685:0.9996163231560092
```

```
Gtk-Message: 22:54:45.012: Failed to load module "colorreload-gtk-module"
```

```
setting arbitrary contour levels with Plotly backend is not supported; use a rang
e to set equally-spaced contours or an integer to set the approximate number of c
ontours with the keyword `levels`. Setting levels to -0.9999232575641008:0.039990
7916144022:0.9996163231560091

setting arbitrary contour levels with Plotly backend is not supported; use a rang
e to set equally-spaced contours or an integer to set the approximate number of c
ontours with the keyword `levels`. Setting levels to -0.9999232575641008:0.333256
596786685:0.9996163231560092
```