Scientific Computing TU Berlin Winter 2021/22 © Jürgen Fuhrmann
Notebook 19

```
using PlutoUI
```


Plotting & visualization

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
- Similar tasks in CAD, Gaming, Virtual Reality . . .

Processing steps in visualization

High level tasks:

- Representation of data using elementary primitives: points,lines, triangles . . .
- Very different depending on purpose

Low level tasks

- Coordinate transformation from "world coordinates" of a particular model to screen coordinates
- Transformation 3D $\to$ 2D, visibility computation
- Coloring, lighting, transparency
- Rasterization: turn smooth data into pixels

Software implementation of low level tasks

- Software: rendering libraries, e.g. Cairo, AGG
- Software for vector based graphics formats, e.g. PDF, postscript, svg
- Typically performed on CPU

Hardware for low level tasks

- Low level tasks are characterized by huge number of very similar operations
- Well adaped to parallelism "Single Instruction, Multiple Data" (SIMD)
- Dedicated hardware: *Graphics Processing Unit* (GPU) can free CPU from these taks
- Multiple parallel pipelines, fast memory for intermediate results

(wikimedia)

GPU Programming

- Typically, GPUs are processing units which are connected via bus interface to CPU
- GPU Programming:
  - Prepare low level data for GPU
  - Send data to GPU
  - Process data in rendering pipeline(s)
- Modern visualization programs have a CPU part and GPU parts a.k.a. *shaders*
  - Shaders allow to program details of data processing on GPU
  - Compiled on CPU, sent along with data to GPU
- Modern libraries: Vulkan, modern OpenGL/WebGL, DirectX
- Possibility to "mis-use" GPU for numerical computations

GPU Programming in the "old days"

- "Fixed function pipeline" in OpenGL 1.1 fixed one particular set of shaders
- Easy to program

```
glClear()
glBegin(GL_TRIANGLES)
glVertex3d(1,2,3)
glVertex3d(1,5,4)
glVertex3d(3,9,15)
glEnd()
glSwapBuffers()
```

- Not anymore: now everything works through shaders leading to highly complex programs

Library interfaces to GPU useful for Scientific Visualization

- **vtk** (backend of **Paraview**)
- **three.js** (for WebGL in the browser)
- vtk.js (for WebGL in the browser)
- plotly.js (for WebGL in the browser)
- Alternatively, work directly with OpenGL...
- very few . . .
  - Money seems to be in gaming, battlefield rendering . . .
  - Problem regadless of julia, python, C++, . . .
- Common approaches:
  - Write data into "vtk" files, use paraview for visualization
  - Visualize using matlab, Mathematica

Graphics in Julia

- **Plots.jl** General purpose plotting package with different backends
  - GPU support via default `gr` backend (based on "old" OpenGL)
  - Support in the browser via `plotly` backend
  - precompilation time significantly improved over the last 2 years
  - Problem: up to now no good support for triangulations
- 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
- **PyPlot.jl**: Interface to **python/matplotlib**
  - realization via PyCall.jl
  - Full functionality of matplotlib
  - also as backend for Plots.jl
  - Problem: slow
- **WriteVTK.jl** vtk file writer for files to be used with paraview - so this is not a plotting library.

PyPlot

Here I show some examples with PyPlot from the previous course

PyPlot resources:

- **Julia package**
- **Julia examples**
- **Matplotlib documentation**

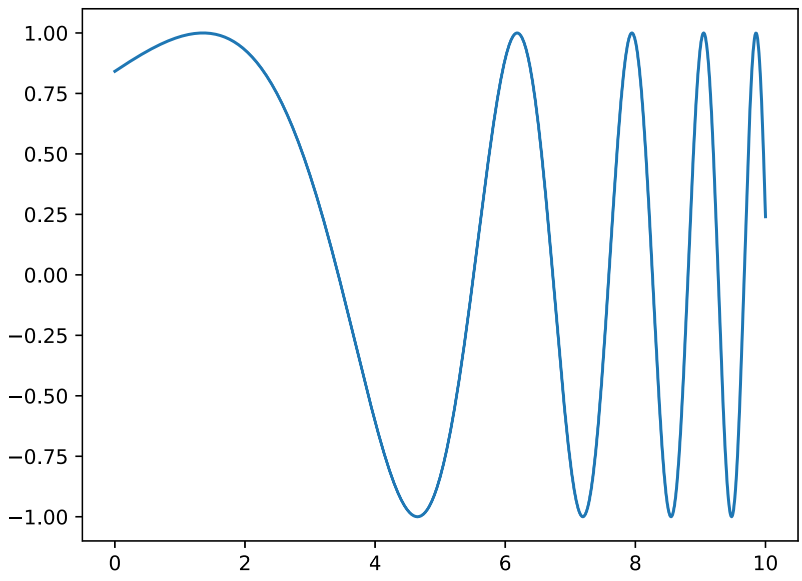
```
import PyPlot
```

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.

```
true
```

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

How to create a plot ?

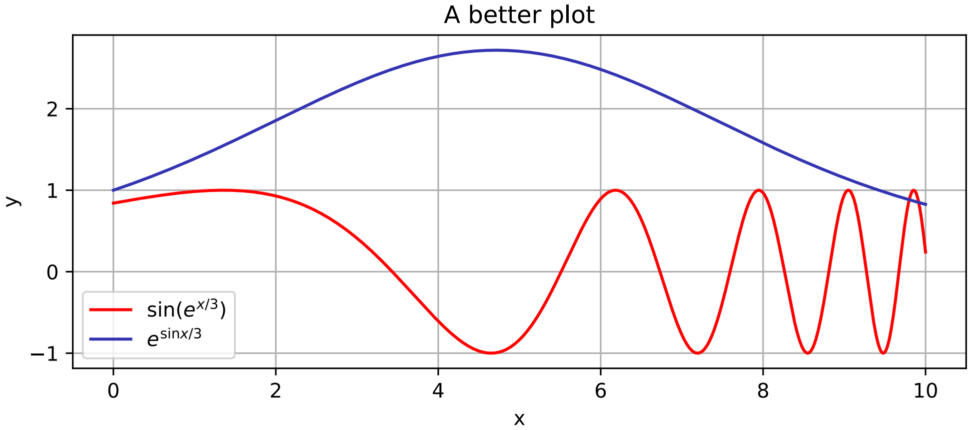



```
let
    X=collect(0:0.01:10)
    PyPlot.clf() # Clear the figure
    PyPlot.plot(X,sin.(exp.(X/3))) # call the plot function
    figure=PyPlot.gcf() # return figure to Pluto
end
```

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

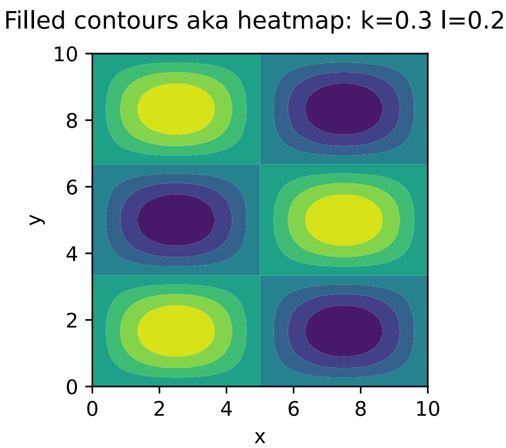



```
let
    X=collect(0:0.01:10)
    PyPlot.clf()
    PyPlot.plot(X,sin.(exp.(X/3)),
        label="\$\\sin(e^{x/3})\$", color=:red) # Plot with label
    PyPlot.plot(X,exp.(sin.(X/3)),
        label="\$e^{\\sin x/3}\$",color=(0.2,0.2,0.7)) # Plot with label
    PyPlot.legend(loc="lower left") # legend 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()
    figure.set_size_inches(8,3) # adjust size
    PyPlot.savefig("myplot.png") # save figure to disk
    figure # return figure
end
```

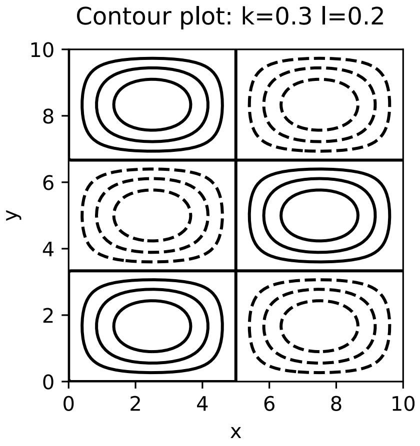
We can use $\LaTeX$ math strings in plot labels here, we just need to escape the `$` symbols with `\` !

Plotting 2D data

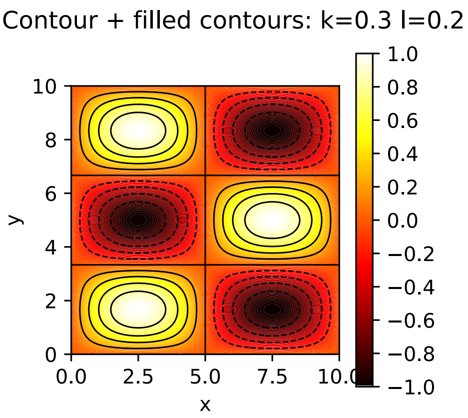
k: l:




```
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
```



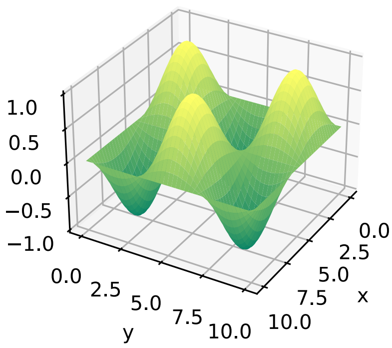

```
let
    PyPlot.clf()
    X=collect(0:0.05:10)
    Y=X
    PyPlot.suptitle("Contour + filled contours: k=$(k) l=$(l)")
    F=[sin(k*π*X[i])*sin(l*π*Y[j]) for i=1:length(X), j=1:length(Y)]
    fmin=minimum(F)
    fmax=maximum(F)
    number_of_isolines=10
    isolines=collect(fmin:(fmax-fmin)/number_of_isolines:fmax)
    cnt=PyPlot.contourf(X,Y,F,cmap="hot",levels=100)
    if fix_moire
        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:

Surface plot: k=0.3 l=0.2




```
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.

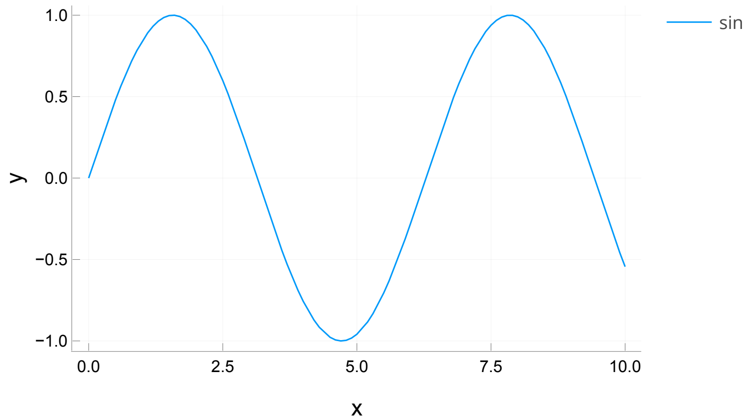
Plots

```
using Plots
```

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()
```

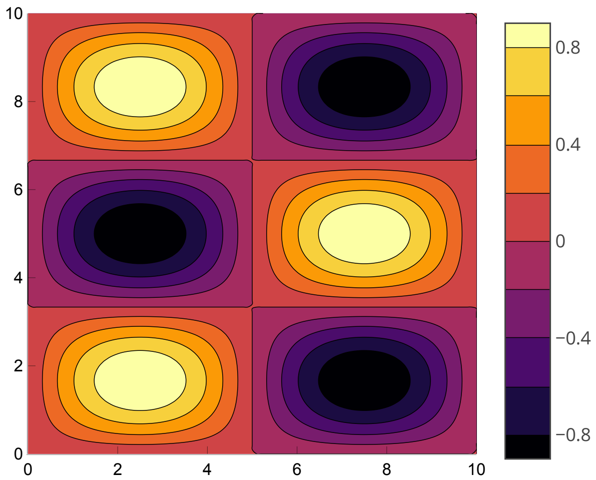
```
X =
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1
X=collect(0:0.1:10)
```



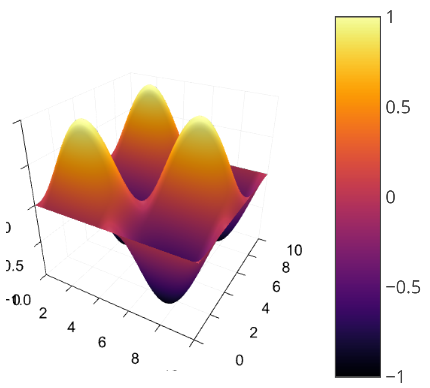

```
Plots.plot(X,sin.(X),size=(500,300),xlabel="x",ylabel="y",label="sin")
```

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

k: l:



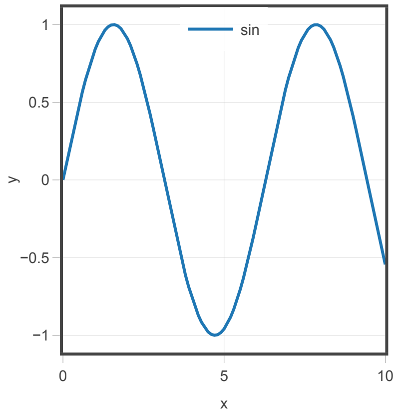

```
Plots.contour(X,X,F,fill=true,size=(400,350))
```




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

PlutoVista

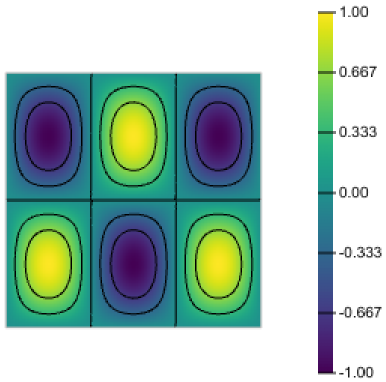
```
using PlutoVista
```




```
PlutoVista.plot(X,sin.(X),xlabel="x",ylabel="y",label="sin",legend=:ct)
```

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

k: l:



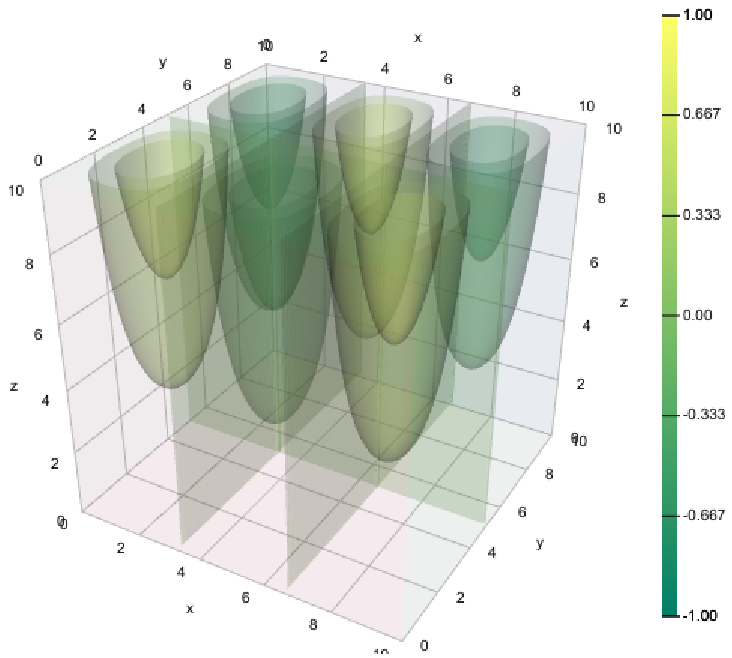

```
PlutoVista.contour(X,X,F2,size=(400,350),levels=5,color=:hot)
```

```
using GridVisualize
```

```
X3=collect(0:0.1:10);
```

```
F3=[sin(k3*π*X3[i])*sin(l3*π*X3[j])*X3[k]/10 for i=1:length(X3), j=1:length(X3),
k=1:length(X3)];
```

k: l:



```
GridVisualize.scalarplot(X3,X3,X3,F3,Plotter=PlutoVista,colormap=:summer)
```

This renders 1030301 values.

"Bonus track"

Feel free watch my **vizcon2 talk** about using vtk from Julia - just to show what could be possible. Unfortunately, these things currently work only on Linux...

Since then, I created code which can do many of these things with GLMakie or PlutoVista through **GridVisualize.jl**

LoadError: UndefVarError: @htl_str not defined

in expression starting at /home/fuhrmann/Wias/teach/scicomp/pluto/nb19-plotting.jl#==#95f489e6-6ea8-4281-adf2-67c7dacc905c:3

1. **top-level scope** @ *:0*
2. **#macroexpand#50** @ *expr.jl:112* [inlined]
3. **macroexpand** @ *expr.jl:111* [inlined]
4. **try_macroexpand**(::Module, ::Base.UUID, ::Expr) @ *PlutoRunner.jl:248*
5. **var"#run_expression#25"**(::Bool, ::typeof(Main.PlutoRunner.run_expression), ::Module, ::Expr, ::Base.UUID, ::Nothing, ::Nothing) @ *PlutoRunner.jl:477*
6. **top-level scope** @ *none:1*

```
begin
    using HypertextLiteral
    highlight(mdstring,color)= htl"""<blockquote style="padding: 10px; background-color: $(color);">$(mdstring)</blockquote>"""

    macro important_str(s)  :(highlight(Markdown.parse($s),"#ffcccc")) end
    macro definition_str(s) :(highlight(Markdown.parse($s),"#ccccff")) end
    macro statement_str(s)  :(highlight(Markdown.parse($s),"#ccffcc")) end


    html"""
    <style>
     h1{background-color:#dddddd;  padding: 10px;}
     h2{background-color:#e7e7e7;  padding: 10px;}
     h3{background-color:#eeeeee;  padding: 10px;}
     h4{background-color:#f7f7f7;  padding: 10px;}
    </style>
"""
end
```