nb-l06-plotting

November 12, 2019

Scientific Computing, TU Berlin, WS 2019/2020, Lecture 06

Jürgen Fuhrmann, WIAS Berlin

1 Visualization and Visualization in Julia

1.1 Plotting & visualization

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

Purposes of plotting: - Visualization of research result 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 … - …

1.2 Processing steps in visualization

1.2.1 High level tasks:

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

1.2.2 Low level tasks

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

1.3 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

1.4 Hardware for low level tasks

- Huge number of very similar operations
- SIMD parallelism "Single instruction, multiple data" inherent to processing steps in visualization
- Dedicated hardware: *Graphics Processing Unit* (GPU) frees CPU from these taks

- Multiple parallel pipelines, fast memory for intermediate results

1.4.1 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
- google "Julia GPU" ...

1.4.2 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 you write shaders for this, compile them, ...

1.4.3 Library interfaces to GPU useful for Scientific Visualization

GPUs are ubiquitous, why aren't they used for visualization ?

- vtk (backend of Paraview,VisIt)
- GR framework
- Three.js (for WebGL in the browser)
- Makie (as a fresh start in Julia)
- very few ...
  - Money seems to be in gaming, battlefield rendering ...
  - **This is not a Julia only problem** but also for python, C++, ...

1.5 Consequences for Julia

- It is hard to have high quality and performance for lage datasets at once
- Julia in many cases (high performance linear algebra for standard float types, sparse matrix solvers ...) relies on well tested libraries
- Similar approach for graphics
- Makie.jl: fresh start directly based on OpenGL, but but functionality still behind

1.5.1 PyPlot.jl

- Interface to matplotlib from the python world
- Ability to create publication ready graphs
- Limited performance (software rendering, many processing steps in python)
- Best start for users familiar and satisfied with matplotlib performance

1.5.2 Plots.jl

- Meta package with a number of different backends
- Backends are already high level libraries
- Choose based on performance, quality, code stability
- Write code once, just switch backend

GR Framework: `Plots.gr()`

- Design based on Graphical Kernel System (GKS), the first and now nearly forgotten ISO standard for computer graphics as intermediate interface
- Very flexible concerning low level backend (from Tektronix to OpenGL…)
- Corner cases where pyplot has more functionality
- OpenGL ⇒ **fast**!
- Few dependencies

PyPlot.jl once again: `Plots.pyplot()`

- High quality
- Limited performance
- Needs python + matplotlib

More…

- PGFPLots: uses LaTeX typsetting system as backend
  - probably best quality
  - slowest (all data are processed via LaTeX)
  - large dependency
- UnicodePlots: ASCII Art revisited

1.6 Plots.jl workflow

- Use fast backend for exploring large data and developing ideas
- For creating presentable graphics, prepare data in such a way the they can be quickly loaded
- Use high quality backend to tweak graphics for presentation
- If possible, store graphics in vectorized format
- See also blog post by Tamas Papp

1.6.1 Plots.jl ressources

- Learning ressources
- Revise.jl: automatic file reloading + compilation for REPL based workflow

1.7 Preparing plots

- We import `Plots` so you see which methods come from there

[1]:
```
import Plots
```

- Set a flag variable to check if code runs in jupyter

[2]:
```
injupyter=isdefined(Main, :IJulia) && Main.IJulia.inited
```

[2]:
```
true
```

- A simple plot based on defaults
- Just use the plot method

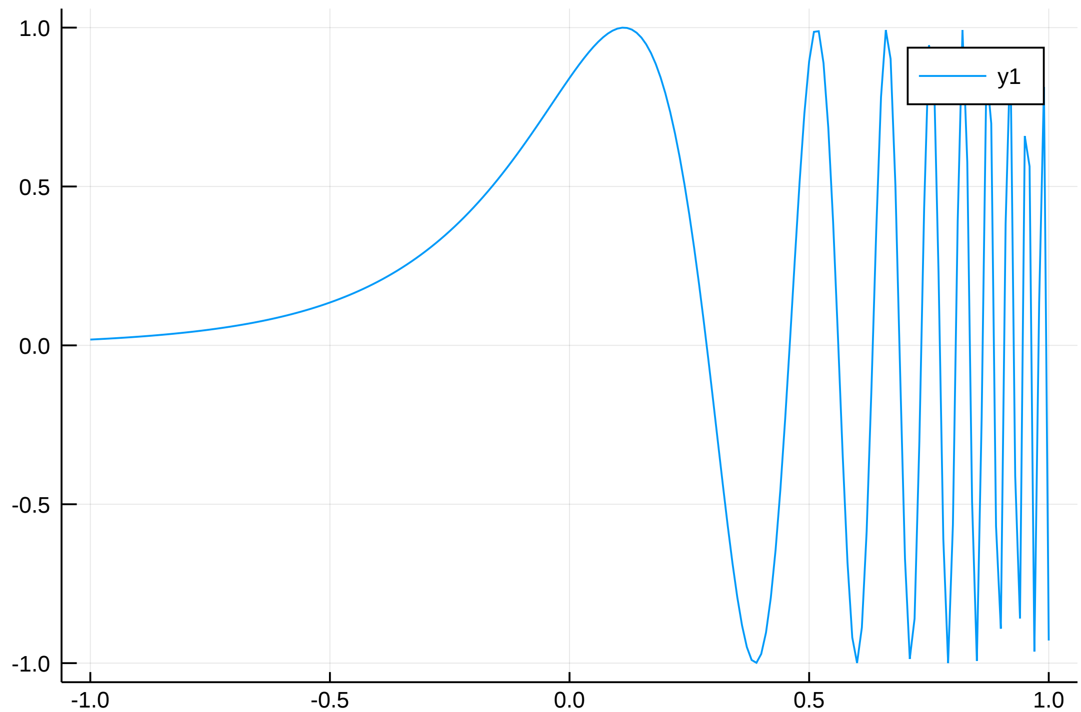
[3]:
```
function example1(;n=100)
    f(x)=sin(exp(4.0*x))
    X=collect(-1.0:1.0/n:1.0)
    p=Plots.plot(X,f.(X))
end
```

[3]:
```
example1 (generic function with 1 method)
```

Run example

[4]:
```
injupyter&&    example1()
```

[4]:



- A good plot has axis labels etc.
- Also we want to have a better label placement
- The plot! method allows a successive build-up of the plot using different *attributes*
- We save the plot to pdf for embedding into presentations

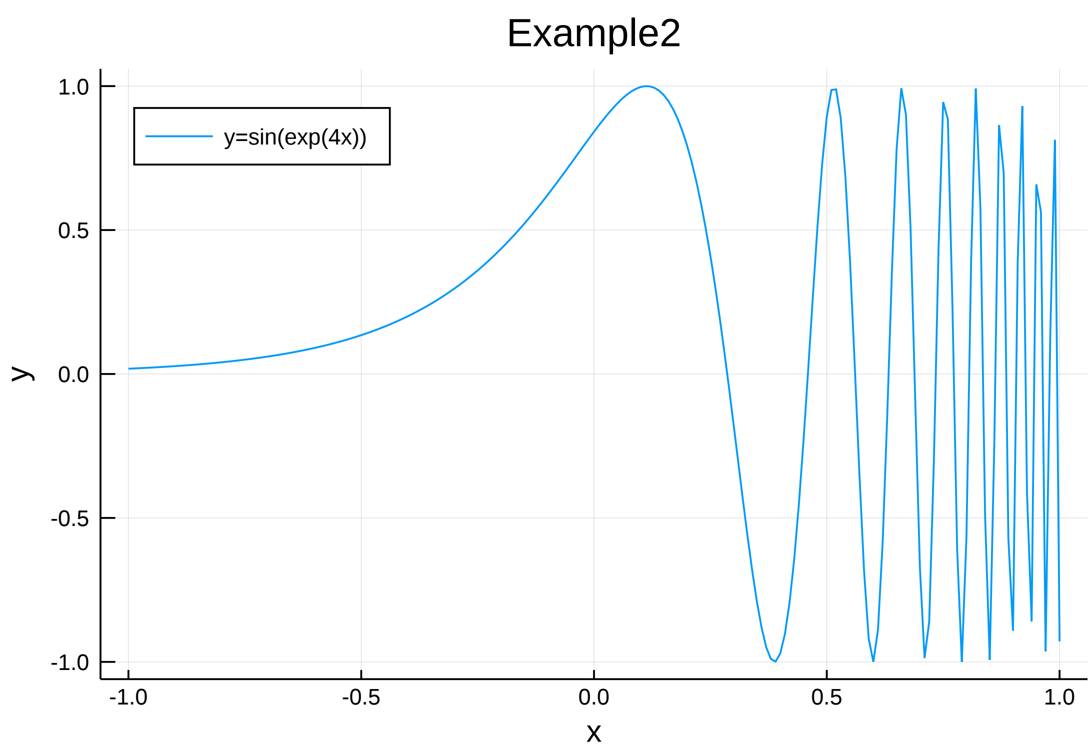
[5]:
```
function example2(;n=100)
    f(x)=sin(exp(4x))
    X=collect(-1.0:1.0/n:1.0)
    p=Plots.plot(framestyle=:full,legend=:topleft, title="Example2")
    Plots.plot!(p, xlabel="x",ylabel="y")
    Plots.plot!(p, X,f.(X), label="y=sin(exp(4x))")
    Plots.savefig(p,"example2.pdf")
    return p
end
```

[5]: example2 (generic function with 1 method)

Run this example

[6]:
```
injupyter&&    example2()
```

[6]:



- Two functions in one plot

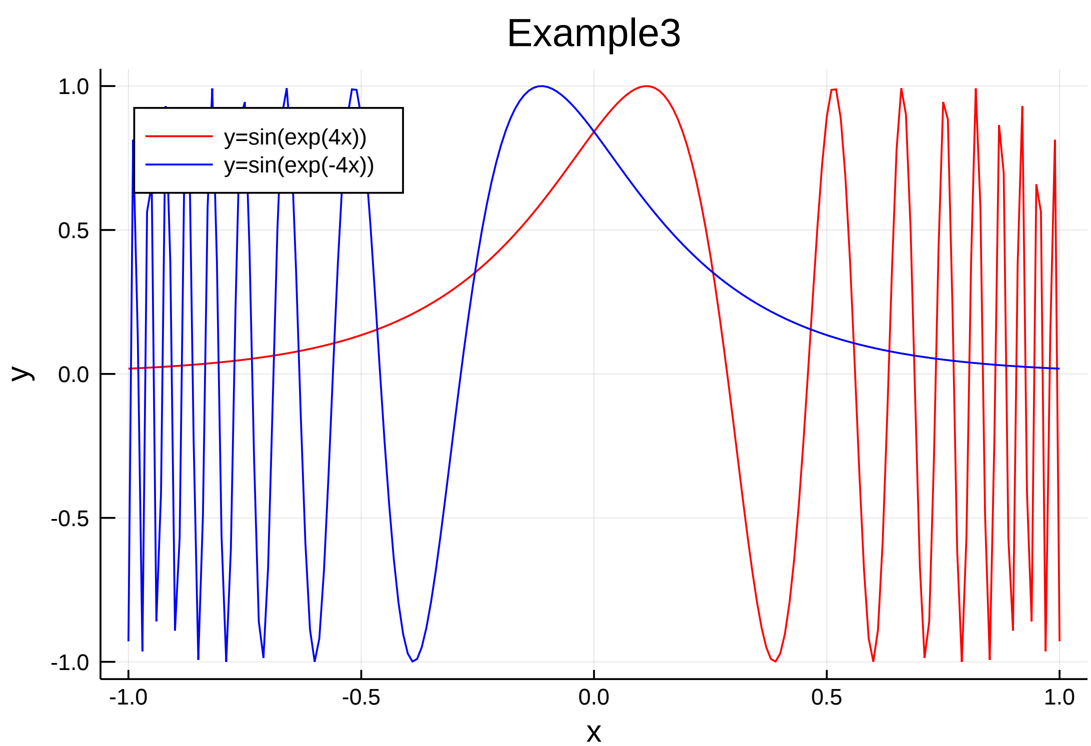
[7]:
```
function example3(;n=100)
    f(x)=sin(exp(4x))
    g(x)=sin(exp(-4x))
    X=collect(-1.0:1.0/n:1.0)
    p=Plots.plot(framestyle=:full,legend=:topleft, title="Example3")
    Plots.plot!(p, xlabel="x",ylabel="y")
    Plots.plot!(p, X,f.(X), label="y=sin(exp(4x))", color=Plots.RGB(1,0,0))
    Plots.plot!(p, X,g.(X), label="y=sin(exp(-4x))", color=Plots.RGB(0,0,1))
end
```

[7]: example3 (generic function with 1 method)

Run this example

[8]:
```
injupyter&&    example3()
```

[8]:



- Two plots arranged

[9]:
```
function example4(;n=100)
    f(x)=sin(exp(4x))
    g(x)=sin(exp(-4x))
    X=collect(-1.0:1.0/n:1.0)
```

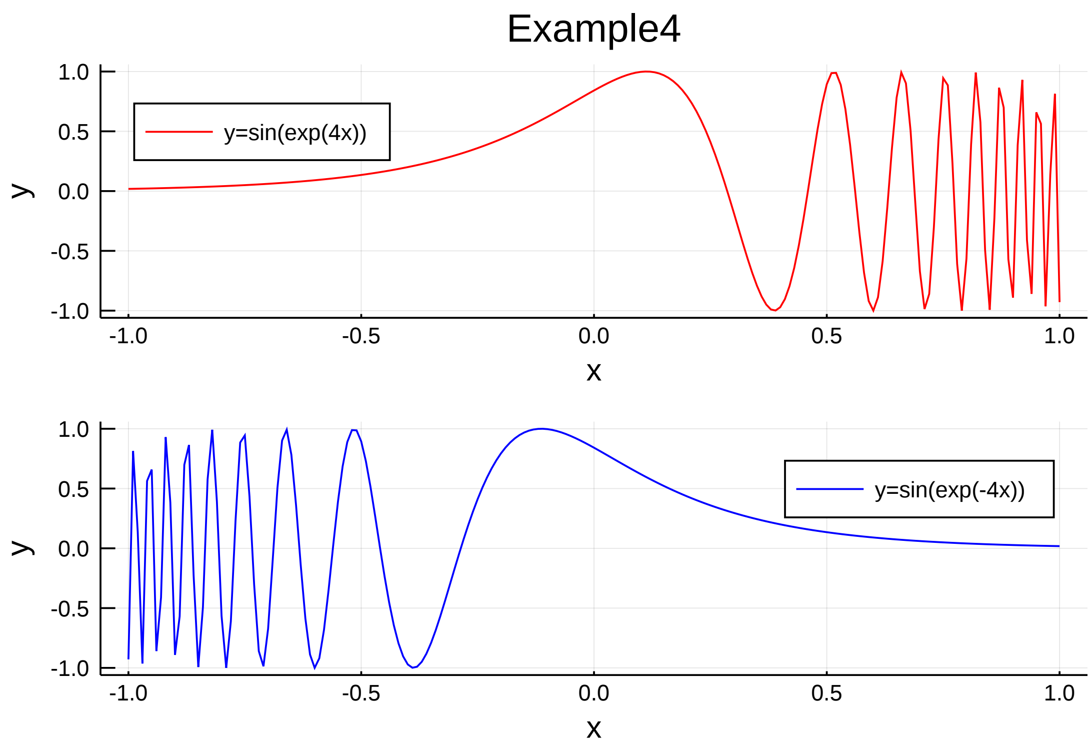
```
    p1=Plots.plot(framestyle=:full,legend=:topleft, title="Example4")
    Plots.plot!(p1, xlabel="x",ylabel="y")
    Plots.plot!(p1, X,f.(X), label="y=sin(exp(4x))", color=Plots.RGB(1,0,0))
    p2=Plots.plot(framestyle=:full,legend=:topright)
    Plots.plot!(p2, xlabel="x",ylabel="y")
    Plots.plot!(p2, X,g.(X), label="y=sin(exp(-4x))", color=Plots.RGB(0,0,1))
    p=Plots.plot(p1,p2,layout=(2,1))
end
```

[9]: example4 (generic function with 1 method)

Run this example

[10]:
```
injupyter&&    example4()
```

[10]:



- Two plots arranged, one scattered

[11]:
```
function example5(;n=100)
    f(x)=sin(exp(4x))
    g(x)=sin(exp(-4x))
    X=collect(-1.0:1.0/n:1.0)
    p1=Plots.plot(framestyle=:full,legend=:topleft, title="Example5")
    Plots.plot!(p1, xlabel="x",ylabel="y")
    Plots.plot!(p1, X,f.(X), label="y=sin(exp(4x))", color=Plots.RGB(1,0,0))
```

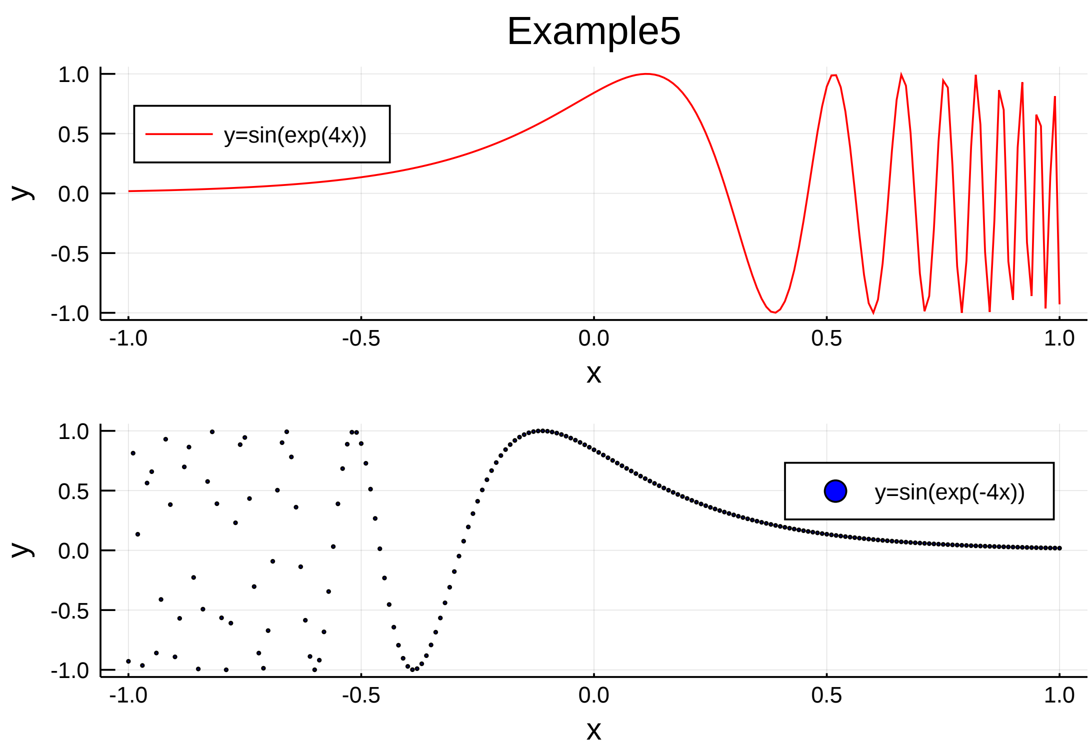
```
    p2=Plots.plot(framestyle=:full,legend=:topright)
    Plots.plot!(p2, xlabel="x",ylabel="y")
    Plots.plot!(p2, X,g.(X), label="y=sin(exp(-4x))", seriestype=:scatter,␣
↪color=Plots.RGB(0,0,1), markersize=0.5)
    p=Plots.plot(p1,p2,layout=(2,1))
end
```

[11]: example5 (generic function with 1 method)

Run this example

[12]:
```
injupyter&&    example5()
```

[12]:



1.8 Plots terminology

- Plot: The whole figure/window
- Subplot: One subplot, containing a title, axes, colorbar, legend, and plot area.
- Axis: One axis of a subplot, containing axis guide (label), tick labels, and tick marks.
- Plot Area: The part of a subplot where the data is shown… contains the series, grid lines, etc.
- Series: One distinct visualization of data. (For example: a line or a set of markers) ### Appearance

- Appearance of Plot, Subplot, Axis, Series is influenced by *attributes*
- Attributes given as Keyword argument to Plots.plot(),Plots.plot!()

1.9 Which attributes are supported in Plots ?

- ``Google'' vs. ``google the right thing''

[13]:
```
Plots.plotattr()
```

```
Specify an attribute type to get a list of supported attributes. Options are
Series, Subplot, Plot, Axis
```

[14]:
```
Plots.plotattr(:Series)
```

```
Defined Series attributes are:
arrow, bar_edges, bar_position, bar_width, bins, colorbar_entry, contour_labels,
contours, fill_z, fillalpha, fillcolor, fillrange, group, hover, label, levels,
line_z, linealpha, linecolor, linestyle, linewidth, marker_z, markeralpha,
markercolor, markershape, markersize, markerstrokealpha, markerstrokecolor,
markerstrokestyle, markerstrokewidth, match_dimensions, normalize, orientation,
primary, quiver, ribbon, series_annotations, seriesalpha, seriescolor,
seriestype, smooth, stride, subplot, weights, x, xerror, y, yerror, z
```

[15]:
```
Plots.plotattr("seriestype")
```

```
seriestype {Symbol}
linetype, lt, seriestypes, st, t, typ

This is the identifier of the type of visualization for this series. Choose from
Symbol[:none, :line, :path, :steppre, :steppost, :sticks, :scatter, :heatmap,
:hexbin, :barbins, :barhist, :histogram, :scatterbins, :scatterhist, :stepbins,
:stephist, :bins2d, :histogram2d, :histogram3d, :density, :bar, :hline, :vline,
:contour, :pie, :shape, :image, :path3d, :scatter3d, :surface, :wireframe,
:contour3d, :volume] or any series recipes which are defined.
Series attribute,  default: path
```

- Heatmap with contourlines

[16]:
```
function example6(;n=100)
    f(x,y)=sin(10x)*cos(10y)*exp(x*y)
    X=collect(-1.0:1.0/n:1.0)
    Y=view(X,:)
    Z=[f(X[i],Y[j]) for i=1:length(X),j=1:length(Y)]
    p=Plots.plot(X,Y,Z, seriestype=:heatmap,seriescolor=Plots.cgrad([:red,:
↪yellow,:blue]))
    p=Plots.plot!(p,X,Y,Z, seriestype=:contour, seriescolor=:black)
```

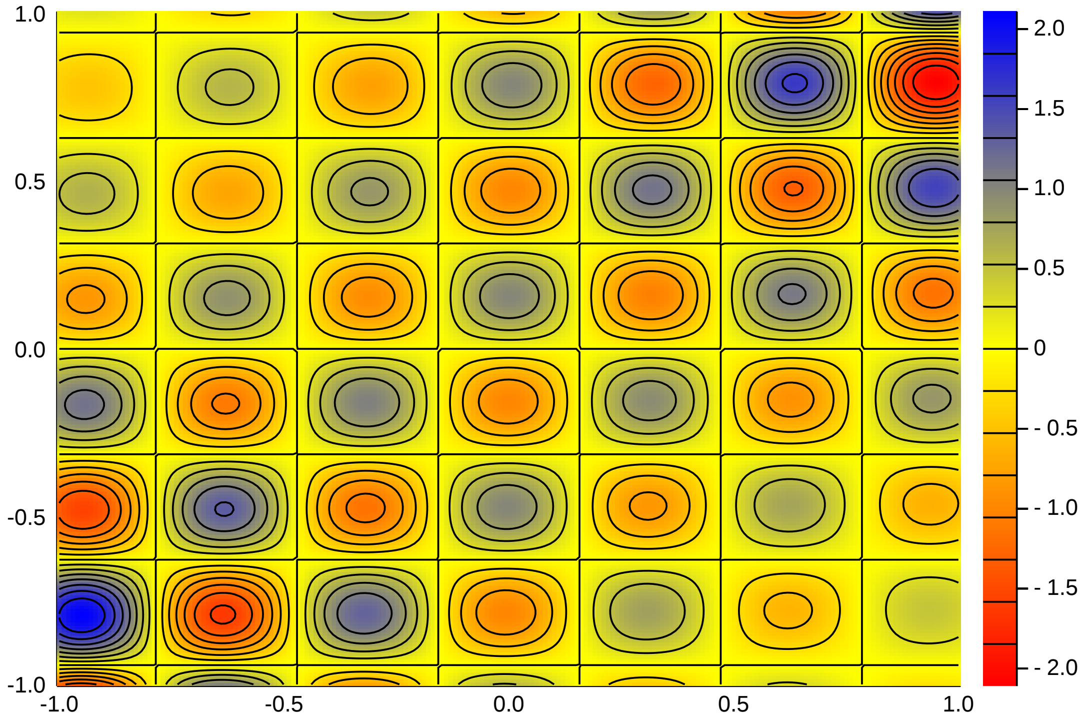
```
end
```

[16]: example6 (generic function with 1 method)

Run this example

[17]:
```
injupyter&&    example6()
```

[17]:



- Heatmap with contourlines plotted during loop

[18]:
```
# import PyPlot
```

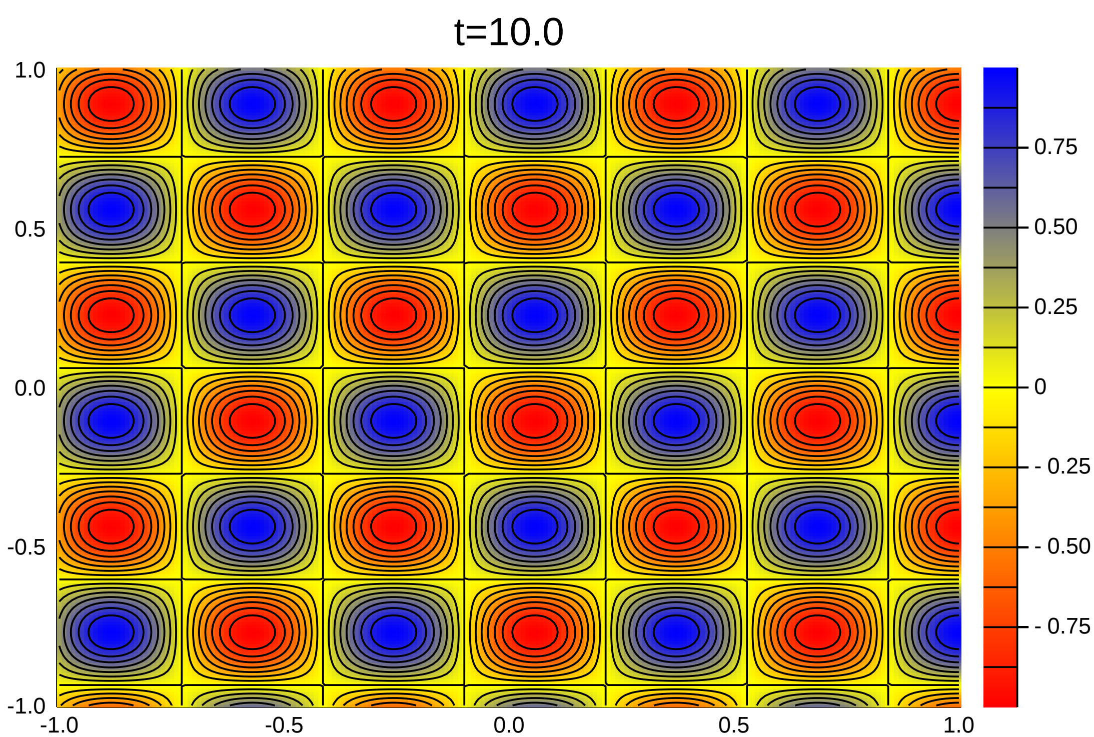
[19]:
```
function example7(;n=100,tend=10)
    f(x,y,t)=sin((10+sin(t))*x-t)*cos(10y-t)
    X=collect(-1.0:1.0/n:1.0)
    Y=view(X,:)
    Z=[f(X[i],Y[j],0) for i=1:length(X),j=1:length(Y)]
    for t=1:0.1:tend
        injupyter && IJulia.clear_output(true)
        for i=1:length(X),j=1:length(Y)
            Z[i,j]=f(X[i],Y[j],t)
        end
```

```
        p=Plots.plot(title="t=$(t)")
        p=Plots.plot!(p,X,Y,Z, seriestype=:heatmap,seriescolor=Plots.cgrad([:
↪red,:yellow,:blue]))
        p=Plots.plot!(p,X,Y,Z, seriestype=:contour, seriescolor=:black)
        Plots.display(p)
        if Plots.backend_name()==:pyplot
            PyPlot.pause(1.0e-10)
        end
    end
end
```

[19]: example7 (generic function with 1 method)

Run this example

[20]:
```
injupyter&&    example7()
```



This notebook was generated using Literate.jl.