Scientific Computing TU Berlin Winter 2021/22 © Jürgen Fuhrmann
Notebook 24

```
begin
    ENV["LC_NUMERIC"]="C"
    using PlutoUI    , PyPlot    , Triangulate    ,SimplexGridFactory    ,ExtendableGrids
    ,ExtendableSparse    ,GridVisualize    ,SparseArrays    , Printf    , HypertextLiteral
    ,PlutoVista
end;
```

Finite volume method: further aspects

Julia packages supporting PDE solution

Up to now we used the Triangulate.jl in order to access mesh generation, for all other functionality, standard Julia packages were used.

There are a number of PDE solution packages in Julia, in particular for the finite element method. During this course, we will use a number of recently developed packages supporting basic functionality for the solution of PDEs. They emerged from the WIAS pdelib project and from scientific computing courses from previous years. These are:

- **ExtendableGrids.jl**: unstructured grid management library
- **GridVisualize.jl**: grid and function visualization related to ExtendableGrids.jl
- **PlutoVista.jl** Efficient plotting in Pluto notebooks bases on Javascript and WebGL. Alternative to PyPlot and able to work with GridVisualize.jl.
- **SimplexGridFactory.jl**: unified high level mesh generator interface
- **ExtendableSparse.jl**: convenient and efficient sparse matrix assembly

We will use all of them in this lecture.

Contents

Finite volume method: further aspects
- Julia packages supporting PDE solution
- Dirichlet boundary conditions
  - Three main possibilities to implement Dirichlet boundary conditions:
  - Algebraic manipulation
  - Modification of boundary equations
  - Penalty method: the "lazy" way
  - Matrix assembly
- Calculation example
  - Grid generation
    - Desired number of triangles
  - Solving the problem
    - Problem data
- Convergence test
  - Conclusions

Dirichlet boundary conditions

So far, we discussed the implementation of Robin boundary conditions for the finite volume method. Neumann boundary conditions are a special case.

Dirichlet boundary conditions already have been qualified as a limiting case. We will discuss this issue here.

Assume the Dirichlet boundary value problem

$$\begin{aligned} -\nabla\delta\cdot\nabla u &= f & \text{in } \Omega \\ u &= \beta & \text{on } \partial\Omega \end{aligned}$$

Three main possibilities to implement Dirichlet boundary conditions:

- Eliminate Dirichlet BC algebraically after building of the matrix, i.e. fix ``known unknowns'' at the Dirichlet boundary $\Rightarrow$ highly technical when only a part of the boundary is affected
- Modifiy matrix such that equations at boundary exactly result in Dirichlet values $\Rightarrow$ loss of symmetry of the matrix
- Penalty method: replace the Dirichlet boundary condition by a Robin boundary condition with high transfer coefficient

We discuss these possibilities for a 1D problem in $\Omega = (0,1)$ with tridiagonal matrix:

$$\begin{aligned} -u_{xx} &= f & \text{in } \Omega \\ u(0) &= \beta_0 \\ u(1) &= \beta_1 \end{aligned}$$

Algebraic manipulation

- Matrix A of homogeneous Neumann problem - no regard to boundary values.

$$AU = \begin{pmatrix} \frac{1}{h} & -\frac{1}{h} & & & \\ -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & \\ & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & \\ & & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \end{pmatrix}$$

- A is diagonally dominant, but neither idd, nor sdd.
- Introduce the Dirichlet boundary conditions by fixing the value of u_1 and eliminating the corresponding equation:

$$A'U = \begin{pmatrix} \frac{2}{h} & -\frac{1}{h} & & \\ -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & \\ & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} u_2 \\ u_3 \\ \vdots \end{pmatrix} = \begin{pmatrix} f_2 + \frac{1}{h}\beta \\ f_3 \\ \vdots \end{pmatrix}$$

- A' is idd and stays symmetric

This operation is quite technical to implement, even more so for triangular meshes or for systems with multiple PDEs.

Modification of boundary equations

- Modify equation at boundary to exactly represent Dirichlet values

$$A'U = \begin{pmatrix} \frac{1}{h} & 0 & & & \\ -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & \\ & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & \\ & & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \end{pmatrix} = \begin{pmatrix} \frac{1}{h}\beta \\ f_2 \\ f_3 \\ \vdots \end{pmatrix}$$

- A is not anymore irreducible
- Loss of symmetry $\Rightarrow$ problem e.g. with CG method

Penalty method: the "lazy" way

This corresponds to replacing the Dirichlet boundary condition $u = \beta$ with a Robin boundary condition

$$\delta\partial_n u + \frac{1}{\varepsilon}u = \frac{1}{\varepsilon}\beta$$

In practice we perform this operation on a discrete level:

$$A'U = \begin{pmatrix} \frac{1}{\varepsilon} + \frac{1}{h} & -\frac{1}{h} & & & \\ -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & \\ & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & \\ & & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \end{pmatrix} = \begin{pmatrix} f_1 + \frac{1}{\varepsilon}\beta \\ f_2 \\ f_3 \\ \vdots \end{pmatrix}$$

- A' is idd, symmetric, and the realization is technically easy.
- If ε is small enough, $u_1 = \beta$ will be satisfied exactly within floating point accuracy.
- Drawback: formally, this creates a large condition number
- Iterative methods should be initialized with Dirichlet values, so we start in a subspace where this is not relevant
- Works also nonlinear problems, finite volume methods

Matrix assembly

```
function trifactors!(ω, e, itri, pointlist, trianglelist)
    # Obtain the node numbers for triangle itri
    i1=trianglelist[1,itri]
    i2=trianglelist[2,itri]
    i3=trianglelist[3,itri]

    # Calculate triangle area:
    #   Matrix of edge vectors
    V11= pointlist[1,i2]- pointlist[1,i1]
    V21= pointlist[2,i2]- pointlist[2,i1]

    V12= pointlist[1,i3]- pointlist[1,i1]
    V22= pointlist[2,i3]- pointlist[2,i1]

    V13= pointlist[1,i3]- pointlist[1,i2]
    V23= pointlist[2,i3]- pointlist[2,i2]

    #   Compute determinant
    det=V11*V22 - V12*V21

    #   Area
    area=0.5*det

    # Squares of edge lengths
    dd1=V13*V13+V23*V23 # l32
    dd2=V12*V12+V22*V22 # l31
    dd3=V11*V11+V21*V21 # l21

    # Contributions to e_kl=σ_kl/h_kl
    e[1]= (dd2+dd3-dd1)*0.125/area
    e[2]= (dd3+dd1-dd2)*0.125/area
    e[3]= (dd1+dd2-dd3)*0.125/area

    # Contributions to ω_k
    ω[1]= (e[3]*dd3+e[2]*dd2)*0.25
    ω[2]= (e[1]*dd1+e[3]*dd3)*0.25
    ω[3]= (e[2]*dd2+e[1]*dd1)*0.25
end;
```

```
function bfacefactors!(γ,ibface, pointlist, segmentlist)
    i1=segmentlist[1,ibface]
    i2=segmentlist[2,ibface]
    dx=pointlist[1,i1]-pointlist[1,i2]
    dy=pointlist[2,i1]-pointlist[2,i2]
    d=0.5*sqrt(dx*dx+dy*dy)
    γ[1]=d
    γ[2]=d
end;
```

```
assemble! (generic function with 1 method)
```

```
function assemble!(matrix, # System matrix
                   rhs,     # Right hand side vector
                   δ,       # heat conduction coefficient
                   f::Tf, # Source/sink function
                   β::Tβ,  # boundary condition function
                   pointlist,
                   trianglelist,
                   segmentlist) where{Tf,Tβ}
    penalty=1.0e30
    num_nodes_per_cell=3;
    num_edges_per_cell=3;
    num_nodes_per_bface=2
    ntri=size(trianglelist,2)
    nbface=size(segmentlist,2)

    # Local edge-node connectivity
    local_edgenodes=[ 2 3; 3 1; 1 2]'

    # Storage for form factors
    e=zeros(num_nodes_per_cell)
    ω=zeros(num_edges_per_cell)
    γ=zeros(num_nodes_per_bface)

    # Initialize right hand side to zero
    rhs.=0.0

    # Loop over all triangles
    for itri=1:ntri
        trifactors!(ω,e,itri,pointlist,trianglelist)
    # Assemble nodal contributions to right hand side
        for k_local=1:num_nodes_per_cell
            k_global=trianglelist[k_local,itri]
            x=pointlist[1,k_global]
            y=pointlist[2,k_global]
            rhs[k_global]+=f(x,y)*ω[k_local]
        end

        # Assemble edge contributions to matrix
        for iedge=1:num_edges_per_cell
            k_global=trianglelist[local_edgenodes[1,iedge],itri]
            l_global=trianglelist[local_edgenodes[2,iedge],itri]
            matrix[k_global,k_global]+=δ*e[iedge]
            matrix[l_global,k_global]-=δ*e[iedge]
            matrix[k_global,l_global]-=δ*e[iedge]
            matrix[l_global,l_global]+=δ*e[iedge]
        end
    end

    # Assemble boundary conditions

    for ibface=1:nbface
        for k_local=1:num_nodes_per_bface
            k_global=segmentlist[k_local,ibface]
            matrix[k_global,k_global]+=penalty
            x=pointlist[1,k_global]
            y=pointlist[2,k_global]
            rhs[k_global]+=penalty*β(x,y)
        end
    end
end
```

Calculation example

Now we are able to solve our intended problem. This time, we create the discretization grid using the package `SimplexGridFactory.jl` which provides and easier interface to mesh generation via `Triangulate.jl`.

Grid generation

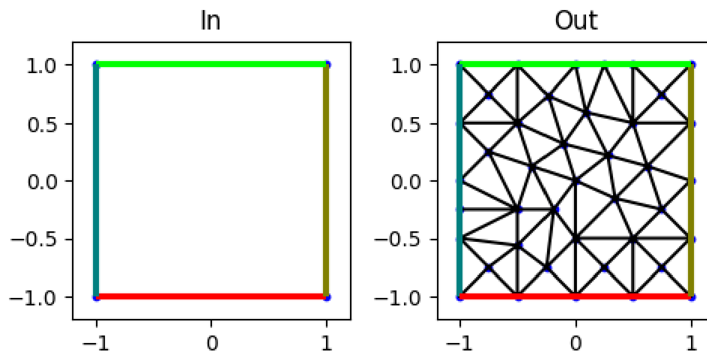
```
describe_grid (generic function with 1 method)
```

```
# We use the SimplexGridBuilder from SimplexGridFactory.jl
function describe_grid()
    # Create a SimplexGridBuilder structure which can collect
    # geometry information
    builder=SimplexGridBuilder(Generator=Triangulate)

    # Add points, record their numbers
    p1=point!(builder,-1,-1)
    p2=point!(builder,1,-1)
    p3=point!(builder,1,1)
    p4=point!(builder,-1,1)

    # Connect points by respective facets (segments)
    facetregion!(builder,1)
    facet!(builder,p1,p2)
    facetregion!(builder,2)
    facet!(builder,p2,p3)
    facetregion!(builder,3)
    facet!(builder,p3,p4)
    facetregion!(builder,4)
    facet!(builder,p4,p1)
    options!(builder,maxvolume=0.1)
    builder
end
```

```
builder =
  SimplexGridBuilder(Triangulate, 4, 1, 1.0, 1.0e-12, [1, 2, 3, 4], [[1, 2], [2, 3], [3, 4],
```

```
builder=describe_grid()
```

We can plot the input and the possible output of the builder.




```
builderplot(builder,Plotter=PyPlot)
```

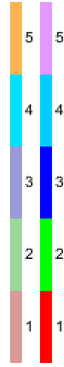
The simplexgrid method creates an object of type ExtendableGrid which is defined in ExtendableGrids.jl. We can overwrite the maxvolume default which we used in `describe_grid`.

```
grid = ExtendableGrids.ExtendableGrid{Float64, Int32};
       dim: 2 nodes: 24 cells: 30 bfaces: 16
```

```
grid=simplexgrid(builder,maxvolume=4/desired_number_of_triangles)
```

Desired number of triangles

From the desired number of triangles, we can calculate a value fo the maximum area constraint passed to the mesh generator: Desired number of triangles: 20




```
gridplot(grid, Plotter=PlutoVista,resolution=(300,300))
```

Solving the problem

Problem data

f (generic function with 1 method)

```
f(x,y)=sinpi(x)*sinpi(y)
```

β (generic function with 1 method)

```
β(x,y)=0
```

δ = 1

Data of the grid are accessed in a **Dictionary** like fashion. `Coordinates` , `CellNodes` and `BFaceNodes` are abstract types defined in `ExtendableGrids.jl` . Behind this is a dictionary with types as keys allowing type-stable access of the contents like in a struct and easy extension by defining additional key types. See **here** for more information.

solve_example (generic function with 1 method)

```
function solve_example(grid)
    # Initialize sparse matrix and right hand side
    n=num_nodes(grid)
    matrix=spzeros(n,n)
    rhs=zeros(n)
    # Call the assemble function.
    assemble!(matrix,rhs,δ,f,β,
          grid[Coordinates],
          grid[CellNodes],
          grid[BFaceNodes])
    # Solve
    sol=matrix\rhs
end
```

solution =
[7.58983e-63, -1.58037e-62, 1.56301e-62, -1.8106e-62, 0.00546284, 1.23156e-32, -2.0255e-33

```
solution=solve_example(grid)
```

`scalarplot` from GridVisualize.jl allows easy handling of plotting on unstructured grids with reasonable defaults.

```
scalarplot(grid,solution,Plotter=PlutoVista,resolution=
(300,300),isolines=11,colormap=:bwr)
```

Convergence test

How good is our implementation and the choice of the penalty method for Dirichlet boundary conditions ? - Perform a convergence test on ever finer grids!

For this purpose we need to calculate error norms. Based on the L2-Norm

$$||u||_0^2 = \int_\Omega u^2 d\omega$$

we implement a discrete analogon for a discrete solution $u_h = (u_k)_{k\in\mathcal{N}}$

$$||u_h||_{0,h}^2 = \int_\Omega u_h^2 d\omega = \sum_{k\in\mathcal{N}} |\omega_k| u_k^2$$

Further, we implement the "h1"-norm

$$|u|_1^2 = \int_\Omega |\vec{\nabla} u|^2 d\omega$$

wich measures the error in the gradient, and its discrete analogon We may discuss the details later.

```
fvnorms (generic function with 1 method)
```

```
function fvnorms(u,pointlist,trianglelist)
    local_edgenodes=[ 2 3; 3 1; 1 2]'
    num_nodes_per_cell=3;
    num_edges_per_cell=3;
    e=zeros(num_nodes_per_cell)
    ω=zeros(num_edges_per_cell)
    l2norm=0.0
    h1norm=0.0
    ntri=size(trianglelist,2)
    for itri=1:ntri
        trifactors!(ω,e,itri,pointlist,trianglelist)
        for k_local=1:num_nodes_per_cell
            k=trianglelist[k_local,itri]
            x=pointlist[1,k]
            y=pointlist[2,k]
            l2norm+=u[k]^2*ω[k_local]
        end
        for iedge=1:num_edges_per_cell
            k=trianglelist[local_edgenodes[1,iedge],itri]
            l=trianglelist[local_edgenodes[2,iedge],itri]
            h1norm+=(u[k]-u[l])^2*e[iedge]
        end
    end
    return (sqrt(l2norm),sqrt(h1norm));
end
```

Define an exact solution of the homogeneous Dirichlet boundary value problem on $\Omega=(-1,1)\times(-1,1)$

$$\begin{aligned} -\nabla\delta\cdot\nabla u &= f && \text{in } \Omega \\ u &= 0 && \text{on } \partial\Omega \end{aligned}$$

```
k=1; l=1;
```

```
fexact(x,y)=sinpi(k*x)*sinpi(l*y);
```

The right corresponding hand side is

```
frhs(x,y)=(k^2+l^2)*pi^2*fexact(x,y);
```

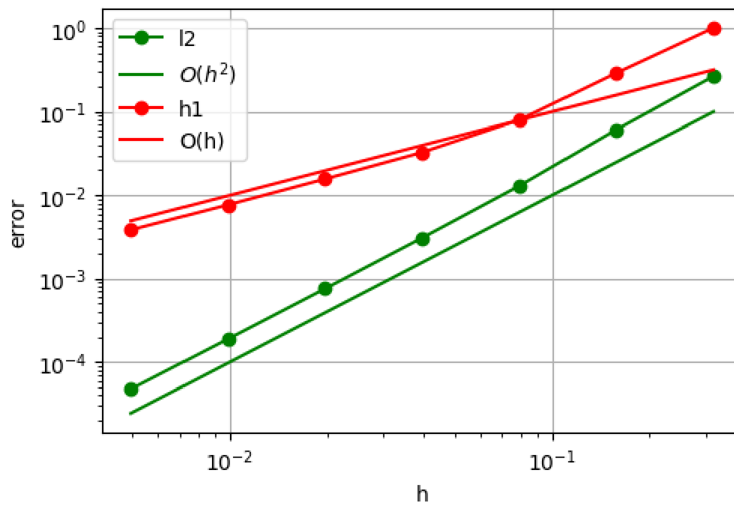
Run convergence test for a number of grid refinement levels

```
convergence_test (generic function with 1 method)
```

```
function convergence_test(;nref0=0, nref1=1,k=1,l=1,extsparse=false)
    allh=[]
    alll2=[]
    allh1=[]

    β(x,y)=0

    for iref=nref0:nref1
     # define the refinement level via the maximum area constraint
        area=0.1*2.0^(-2*iref)
        h=sqrt(area)
        grid=simplexgrid(builder,maxvolume=area)

        n=num_nodes(grid)
        rhs=zeros(n)

        # Optionally, use the sparse matrix from ExtendableGrids
        if extsparse
            matrix=ExtendableSparseMatrix(n,n)
        else
            matrix=spzeros(n,n)
        end
        rhs=zeros(n)

        assemble!(matrix,rhs,δ,frhs,β,
          grid[Coordinates],grid[CellNodes],grid[BFaceNodes])
        sol=matrix\rhs
        uexact=map(fexact,grid)

        (l2norm,h1norm)=fvnorms(uexact-sol,grid[Coordinates],grid[CellNodes])

        push!(allh,h)
        push!(allh1,h1norm)
        push!(alll2,l2norm)
    end
    allh,alll2,allh1
end

```

```
([0.316228, 0.158114, 0.0790569, 0.0395285, 0.0197642, 0.00988212, 0.00494106], [0.265187,
```

```
allh,alll2,allh1=convergence_test(nref0=0,nref1=6,extsparse=true)
```



Conclusions

We see the second order convergence of the solution and first order convergence of the gradient. This is the typical behavior which we also would expect from the finite element method.

Concerning the complexity, the ExtendableSparseMatrix uses an intermediate data structure for collecting the matrix entries. If we directly insert data into a compressed column data structure, there is a considerable overhead for reorganization of the long arrays describing the matrix.