Advanced Topics from Scientific Computing
TU Berlin Winter 2024/25
Notebook 13a
 Jürgen Fuhrmann

```
begin
    ENV["LC_NUMERIC"] = "C"
    using PlutoUI, PyPlot, Triangulate, SparseArrays, Printf
    using HypertextLiteral: @htl, @htl_str
    PyPlot.svg(true)
end;
```

Contents

Implementation of the finite volume method

This notebook gives an insight in to the assembly techniques in the VoronoiFVM.jl package on a very elementary level.

We discuss the implementation of the method for the problem

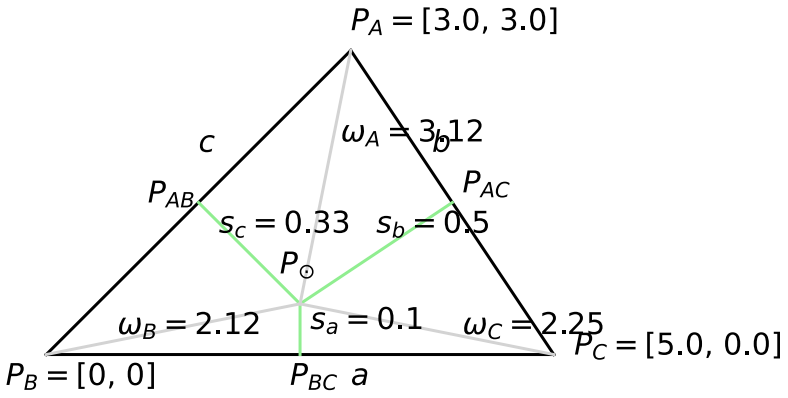
$$-\nabla \cdot (\delta \vec{\nabla} u) = f$$
$$\delta \partial_n u + \alpha u = g$$

on triangular meshes.

Geometrical data for finite volumes

The basic idea of this approach consists in a triangle-wise assembly loop which is very similar to the way the finite element method would be implemented. As a consequence, we need to be able to calculate the contributions to the Voronoi cell data for each triangle.

PA=[3.0, 3.0]




```
let
    PA = [PA1, PA2]
    PC = [5.0, 0.0]
    PB = [0, 0]
    ω = zeros(3)
    e = zeros(3)
    trifactors!(ω,e,1,hcat(PA, PB, PC),hcat([1,2,3]) )
    ω=round.(ω, digits=2)
    e=round.(e, digits=2)

    line(p1, p2; color = :black) = PyPlot.plot([p1[1], p2[1]], [p1[2], p2[2]], "-", color = color)
    text(p, txt; fontsize = 15) = PyPlot.text(p[1], p[2], txt, fontsize = fontsize)
    circumcenter(PA, PB, PC) = Triangulate.tricircumcenter!([0.0, 0.0], PA, PB, PC)
    edgecenter(PA, PB) = [(PA[1] + PB[1]) / 2, (PA[2] + PB[2]) / 2]
    clf()
    ax = PyPlot.axes(aspect = 1.0)
    CC = circumcenter(PA, PB, PC)
    line(PA, PB)
    line(PB, PC)
    line(PA, PC)
    line(CC, edgecenter(PA, PB), color = :lightgreen)
    line(CC, edgecenter(PA, PC), color = :lightgreen)
    line(CC, edgecenter(PB, PC), color = :lightgreen)
    line(CC, PA, color = :lightgray)
    line(CC, PB, color = :lightgray)
    line(CC, PC, color = :lightgray)

    text(PB - [0.4, 0.3], L"P_B=" * "$(PB)")
    text(PB + [0.7, 0.2], L"$\omega_B=$"*"$(ω[2])")
    text(PA + [0, 0.2], L"P_A=" * "$(PA)")
    text(PA - [0.1, 0.9], L"$\omega_A=$"*"$(ω[1])")
    text(PC + [-0.9, 0.2], L"$\omega_C=$"*"$(ω[3])")
    text(PC + [0.2, 0], L"P_C=" * "$(PC)")
    text(CC + [-0.2, 0.3], L"$P_{\odot}$")
    text(edgecenter(edgecenter(PB, PC), CC) + [0.1, 0], L"${s}_a=$"*"$(e[1])")
    text(edgecenter(edgecenter(PA, PC), CC) + [0, 0.2], L"${s}_b=$"*"$(e[2])")
    text(edgecenter(edgecenter(PA, PB), CC) + [-0.3, 0.2], L"${s}_c=$"*"$(e[3])")
    text(edgecenter(PB, PC) + [0.5, -0.3], L"a")
    text(edgecenter(PB, PC) + [-0.1, -0.3], L"$P_{BC}$")
    text(edgecenter(PA, PC) + [-0.2, 0.5], L"b")
    text(edgecenter(PA, PC) + [0.1, 0.1], L"$P_{AC}$")
    text(edgecenter(PA, PB) + [0.0, 0.5], L"c")
    text(edgecenter(PA, PB) + [-0.5, 0.0], L"$P_{AB}$")
    axis("off")

    ax.get_xaxis().set_visible(false)
    ax.get_yaxis().set_visible(false)

    gcf().set_size_inches(5, 5)
    gcf()
end
```

```
Qt: Session management error: None of the authentication protocols specified are supported
```

Needed data

- Edge lengths h_{kl}:

$$a = |P_B P_C|,$$
$$b = |P_A P_C|,$$
$$c = |P_A P_B|$$

- Contributions to lengths of the interfaces between Voronoi cells $|\sigma_{kl} \cap T|$: lines joining the edge centers with the triangle circumcenter $P_\odot$:

$$s_a = |P_{BC} P_\odot|,$$
$$s_b = |P_{AC} P_\odot|,$$
$$s_c = |P_{AB} P_\odot|$$

- Practically, we need the values of the ratios $\frac{\sigma_{kl}}{h_{kl}}$:

$$e_a = \frac{s_a}{a}, e_b = \frac{s_b}{b}, e_c = \frac{s_c}{c}$$

- Triangle contributions to the Voronoi cell areas around the respective triangle nodes $\omega_A = |P_A P_{AB} P_\odot P_{AC}|, \omega_B = |P_B P_{BC} P_\odot P_{AB}|, \omega_C = |P_C P_{AC} P_\odot P_{BC}|$

Calculation steps for the interface contributions

We show the calculation steps for e_a, ω_a, the others can be obtained via corresponding permutations.

1. Semiperimeter:

$$s = \frac{a}{2} + \frac{b}{2} + \frac{c}{2}$$

2. Square area (from Heron's formula):

$$16A^2 = 16s(s-a)(s-b)(s-c) = (-a+b+c)\,(a-b+c)\,(a+b-c)\,(a+b+c)$$

3. Square circumradius:

$$R^2 = \frac{a^2 b^2 c^2}{(-a+b+c)\,(a-b+c)\,(a+b-c)\,(a+b+c)} = \frac{a^2 b^2 c^2}{16A^2}$$

4. Square of the Voronoi interface contribution via Pythagoras:

$$s_a^2 = R^2 - \left(\frac{1}{2}a\right)^2 = -\frac{a^2\left(a^2-b^2-c^2\right)^2}{4\,(a-b-c)\,(a-b+c)\,(a+b-c)\,(a+b+c)}$$

5. Square of interface contribution over edge length:

$$e_a^2 = \frac{s_a^2}{a^2} = -\frac{\left(a^2-b^2-c^2\right)^2}{4\,(a-b-c)\,(a-b+c)\,(a+b-c)\,(a+b+c)} = \frac{(b^2+c^2-a^2)^2}{64A^2}$$

6. Interface contribution over edge length:

$$e_a = \frac{s_a}{a} = \frac{b^2+c^2-a^2}{8A}$$

7. Calculation of the area contributions

$$\omega_a = \frac{1}{4}cs_c + \frac{1}{4}bs_b = \frac{1}{4}(c^2 e_c + b^2 e_b)$$

- The sign chosen implies a positive value of s_a if the angle $\alpha_A < \frac{\pi}{2}$, and a negative value if it is obtuse. In the latter case, this corresponds to the negative length of the line between

edge midpoint and circumcenter, which is exactly the value which needs to be added to the corresponding amount from the opposite triangle in order to obtain the measure of the Voronoi face.
- If an edge between two triangles is locally Delaunay, the summary contribution from the two triangles with respect to this edge will become positive. In 2D, this is equivalent to the fact that the sum of the angles opposite to that edge is $< \pi$

Steps to the implementation

We describe a triangular discretization mesh by three arrays:

- `pointlist`: $2 \times n_{points}$ floating point array of node coordinates of the triangulations. `pointlist[:,i]` then contains the coordinates of point `i`.
- `trianglelist`: $3 \times n_{ntri}$ integer array describing which three nodes belong to a given triangle. `trianglelist[:,i]` then contains the numbers of nodes belonging to triangle `i`.
- `segmentlist`: $2 \times n_{segs}$ integer array describing which two nodes belong to a given boundary segment. `segmentlist[:,i]` contains the numbers of nodes for boundary segment i.

Triangle form factors

For triangle `itri`, we want to calculate the corresponding form factors e and ω:

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

```
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
    return ω[3] = (e[2] * dd2 + e[1] * dd1) * 0.25
end
```

Boundary form factors

Here we need for an interface segment of two points P_A, P_B the contributions to the intersection of the Voronoi cell boundary with the outer boundary which is just the half length: $\gamma_A = \frac{1}{2}|P_A P_B|, \gamma_B = \frac{1}{2}|P_A P_B|$

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

```
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
    return γ[2] = d
end
```

Matrix assembly

The matrix assembly consists of two loops, one over all triangles, and another one over the boundary segments.

The implementation hints at the possibility to work in different space dimensions

```
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
        α,      # boundary transfer coefficient
        β::TB,  # boundary condition function
        pointlist,
        trianglelist,
        segmentlist
    ) where {TF, TB}

    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 in 1:ntri
        trifactors!(ω, e, itri, pointlist, trianglelist)

        # Assemble nodal contributions to right hand side
        for k_local in 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 in 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 in 1:nbface
        bfacefactors!(γ, ibface, pointlist, segmentlist)
        for k_local in 1:num_nodes_per_bface
            k_global = segmentlist[k_local, ibface]
            matrix[k_global, k_global] += α * γ[k_local]
            x = pointlist[1, k_global]
            y = pointlist[2, k_global]
            rhs[k_global] += β(x, y) * γ[k_local]
        end
    end
    return
end

```

Graphical representation

It would be nice to have a graphical representation of the solution data. We can interpret the solution as a piecewise linear function on the triangulation: each triangle has three nodes each carrying one solution value.

On the other hand, a linear function of two variables is defined by values in three points. This allows to define a piecewise linear, continuous solution function. This approach is well known for the P1 finite element method.

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

```
function plot(u, pointlist, trianglelist)
    cmap = "coolwarm" # color map for color coding function values
    num_isolines = 10 # number of isolines for plot
    ax = gca(); ax.set_aspect(1) # don't distort the plot

    # bring data into format understood by PyPlot
    x = view(pointlist, 1, :)
    y = view(pointlist, 2, :)
    t = transpose(triout.trianglelist .- 1)

    # Many (50) filled contour lines give the impression of a smooth color scale
    tricontourf(x, y, t, u, levels = 50, cmap = cmap)
    colorbar(shrink = 0.5) # Put a color bar next to the plot

    # Overlay the plot with isolines
    return tricontour(x, y, t, u, levels = num_isolines, colors = "k")
end

```

An alternative way of showing the result is the 3D plot of the function graph:

```
plot3d (generic function with 1 method)
```

```
function plot3d(u, pointlist, trianglelist)
    cmap = "coolwarm"
    fig = figure(2)
    x = view(pointlist, 1, :)
    y = view(pointlist, 2, :)
    t = transpose(triout.trianglelist .- 1)
    return plot_trisurf(x, y, t, u, cmap = cmap)
end
```

Calculation example

Now we are able to solve our intended problem.

Grid generation

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

```
function make_grid(; maxarea = 0.01)
    triin = TriangulateIO()
    triin.pointlist = Matrix{Cdouble}([-1.0 -1.0; 1.0 -1.0 ; 1.0 1.0 ; -1.0 1.0
]')
    triin.segmentlist = Matrix{Cint}([1 2 ; 2 3 ; 3 4 ; 4 1 ]')
    triin.segmentmarkerlist = Vector{Int32}([1, 2, 3, 4])
    a = @sprintf("%f", maxarea)
    (triout, vorout) = triangulate("pqAa$(a)qQD", triin)
    return triin, triout
end
```

```
(TriangulateIO(                                  , TriangulateIO(
 pointlist=[-1.0 1.0 1.0 -1.0; -1.0 -1.0 1.0 1.0],  pointlist=[-1.0 1.0 … 0.5 -0.5; -1
```

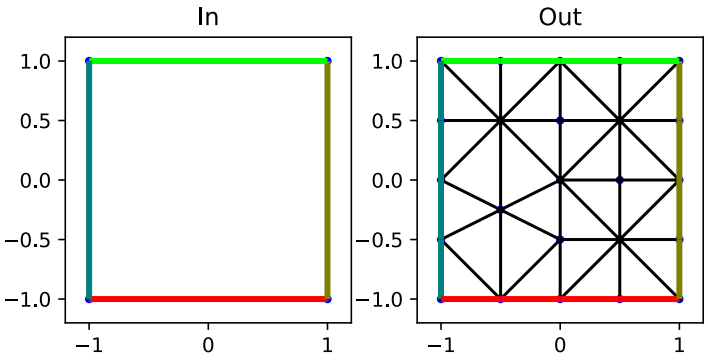
```
triin, triout = make_grid(maxarea = 4.0 / desired_number_of_triangles)
```

Plotting the grid

In the triout data structure, we indeed see a `pointlist`, a `trianglelist` and a `segmentlist`.

We use the `plot_in_out` function from Triangulate.jl to plot the grid.

Plot grid: ☑



Number of points: 24, number of triangles: 30.

Desired number of triangles

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

Solving the problem

Problem data

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

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

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

```
β(x, y) = 0
```

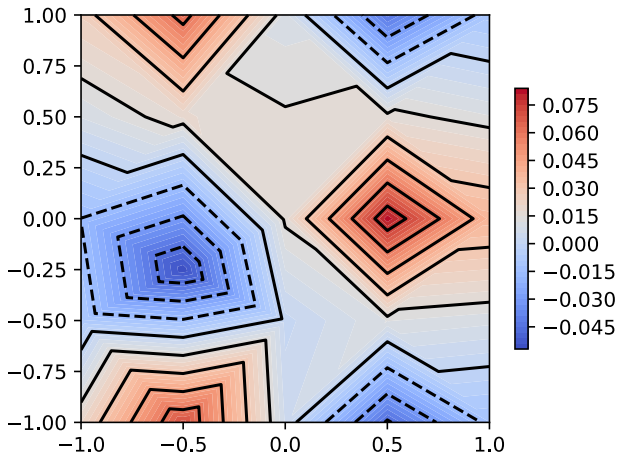
δ: 1.0 α: 1.0

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

```
function solve_example(triout)
    n = size(triout.pointlist, 2)
    matrix = spzeros(n, n)
    rhs = zeros(n)
    assemble!(matrix, rhs, δ, f, α, β, triout.pointlist, triout.trianglelist, triout.segmentlist)
    return sol = matrix \ rhs
end
```

```
solution =
▸ [0.0207156, -0.0121475, -0.010301, 0.0245238, 0.0162066, -0.0152359, 0.00557976, 0.0377
```

```
solution = solve_example(triout)
```




```
clf(); plot(solution, triout.pointlist,
triout.trianglelist);gcf().set_size_inches(4, 4);gcf()
```

3D Plot ? ☐)

```
if do_3d_plot
    clf(); plot3d(solution, triout.pointlist,
triout.trianglelist);gcf().set_size_inches(4, 4);gcf()
end
```