Scientific Computing TU Berlin Winter 2021/22 © Jürgen Fuhrmann
Notebook 26

```
begin
    using PlutoUI   ,ExtendableGrids   ,VoronoiFVM   ,GridVisualize   , PlutoVista
    using HypertextLiteral
    GridVisualize.default_plotter!(PlutoVista)
end;
```

Working with VoronoiFVM.jl

In the previous lectures we introduced the Voronoi finite volume method, and showed how to implement it on for a linear diffusion problem on triangular grids, and how to solve nonlinear systems.

The **VoronoiFVM.jl** Julia package provides a synthesis of these two.

We show how to define scalar linear and nonlinear diffusion problems in the VoronoiFVM package and disscuss its inner workings starting with two examples.

For more information, see its **documentation**.

Linear diffusion problem with Dirichlet boundary conditions

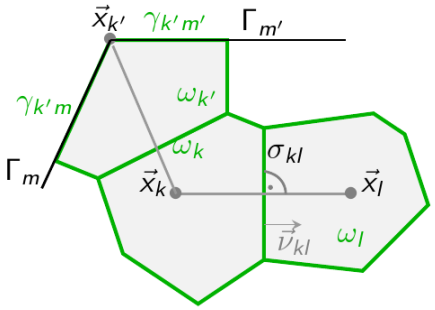
Regard

$$\begin{aligned} -\nabla \cdot (D\vec{\nabla}u) &= f \quad \text{in } \Omega \\ u &= \beta \text{ on } \partial\Omega \end{aligned}$$

The following data characterize the problem:

- Flux $\vec{j} = -D\vec{\nabla}u$
- Dirichlet data β
- Source/sink term f
- Domain Ω

We recall the geometry behind the method:



The package works with multiple interacting species. Therefore we need to define a species index for this particular problem:

```
const spec_idx = 1
```
```
const spec_idx=1
```

- Diffusion coefficient D:

```
const D = 10.0
```
```
const D=10.0
```

Diffusion flux $g(u_k, u_l) = D(u_k - u_l)$.

The following function defines the flux through an interface between two neigboring control volumes which for the Voronoi finite volume method is equivalent to the flux along a triangulation edge. It receives the current unknown data in the two-dimensional array `u`. The first index is the species number, the second index denotes the local index at the given edge. For our problem, we then have $u_k =$ `u[1,1]` and $u_l =$ `u[1,2]`.

The result is written into `f` for species index `1`, so this is a mutating function, which guarantees to cause no allocations.

Additional geometrical data optionally can be obtained from the `edge` parameter.

```
diffusion_flux! (generic function with 1 method)
```
```
function diffusion_flux!(f,u, edge)
    f[spec_idx]=D*(u[spec_idx,1]-u[spec_idx,2])
end
```

- Right hand side function $f(x) = 1$ (just for an example). Once again, the species index is `1`.

```
diffusion_source! (generic function with 1 method)
```
```
function diffusion_source!(f,node)
    f[spec_idx]=1
end
```

- Boundary value β:

```
β = 0.1
```
```
β=0.1
```

Here, we use the `boundary_dirichlet!` function which helps to manage the Dirichlet penalty method for working with Dirichlet boundary conditions.

```
dirichlet_bc! (generic function with 1 method)
```
```
function dirichlet_bc!(f,u,bnode)
    boundary_dirichlet!(f,u,bnode,value=β)
end
```

1D Discretization grid

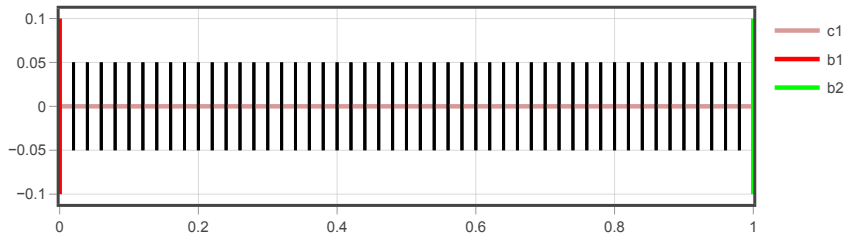
Grid in domain $\Omega = (0, 1)$ consisting of N= 51 points.

```
X =
  [0.0, 0.02, 0.04, 0.06, 0.08, 0.1, 0.12, 0.14, 0.16, 0.18, 0.2, 0.22, 0.24, 0.26, 0.28, 0.3
```

```
X=collect(range(0,1,length=N))
```

```
grid1d = ExtendableGrids.ExtendableGrid{Float64, Int32};
         dim: 1 nodes: 51 cells: 50 bfaces: 2
```

```
grid1d=simplexgrid(X)
```




```
gridplot(grid1d,size=(600,200),legend=:lt)
```

System creation and solution

Here, we bring together the "physics" part of the problem desribed in the flux function etc. and the geometry part described by the discretization grid.

```
system1d =
VoronoiFVM.System{Float64, Int32, Int64, Matrix{Int32}, Matrix{Float64}}(num_species=1)
```

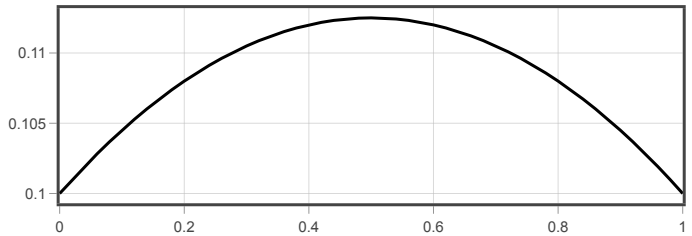
```
system1d=VoronoiFVM.System(grid1d;
         flux=diffusion_flux!,
         source=diffusion_source!,
         bcondition=dirichlet_bc!,
         species=[spec_idx])
```

Using default settings, the system is solved. Optionally, we can obtain information on the solution history.

```
(seconds = 2.89, iters = 2, absnorm = 1.08e-15, relnorm = 9.61e-15, roundoff = 7.41e-15,
```

```
begin
    solution=solve(system1d,inival=0.0, log=true)
    history_summary(system1d)
end
```

We can plot the solution using the `scalarplot` method from the GridVisualize.jl package.




```
scalarplot(grid1d,solution[spec_idx,:],size=(500,200))
```

2D Linear diffusion

For solving a 2D problem, we just need to replace the 1D grid with a 2D grid.

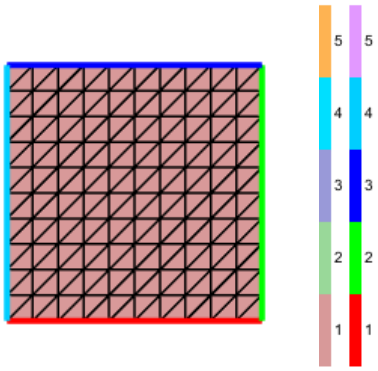
Grid in domain $\Omega = (0,1) \times (0,1)$ consisting of N2= **11** points in each coordinate direction

```
X2 =  [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
```

```
X2=collect(range(0,1,length=N2))
```

```
grid2d = ExtendableGrids.ExtendableGrid{Float64, Int32};
         dim: 2 nodes: 121 cells: 200 bfaces: 40
```

```
grid2d=simplexgrid(X2,X2)
```




```
gridplot(grid2d,size=(300,300))
```

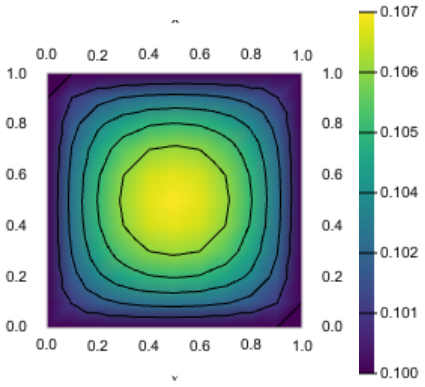
We can define and solve the 2D problem with the same physics functions as the 1D problem:

```
system2d =
VoronoiFVM.System{Float64, Int32, Int64, Matrix{Int32}, Matrix{Float64}}(num_species=1)
```

```
system2d=VoronoiFVM.System(grid2d;
        flux=diffusion_flux!,
        source=diffusion_source!,
        bcondition=dirichlet_bc!,
        species=[spec_idx])
```

```
(seconds = 2.89, iters = 2, absnorm = 1.08e-15, relnorm = 9.61e-15, roundoff = 7.41e-15,
```

```
begin
    solution2d=solve(system2d, log=true)
    history_summary(system1d)
end
```

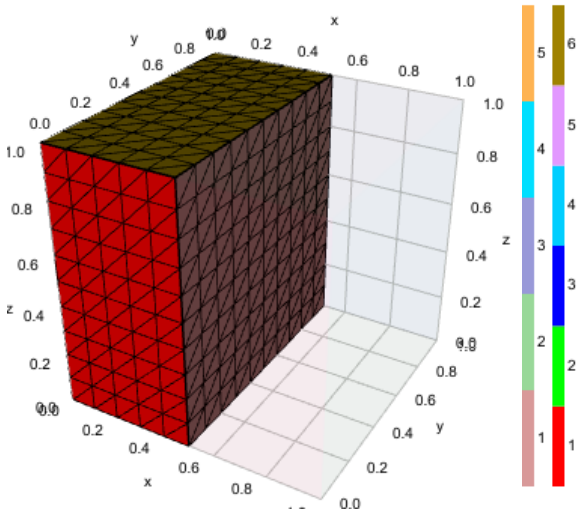



```
scalarplot(grid2d,solution2d[1,:],size=(300,300))
```

3D Linear diffusion

```
grid3d = ExtendableGrids.ExtendableGrid{Float64, Int32};
         dim: 3 nodes: 1331 cells: 6000 bfaces: 1200
```

```
grid3d=simplexgrid(X2,X2, X2)
```



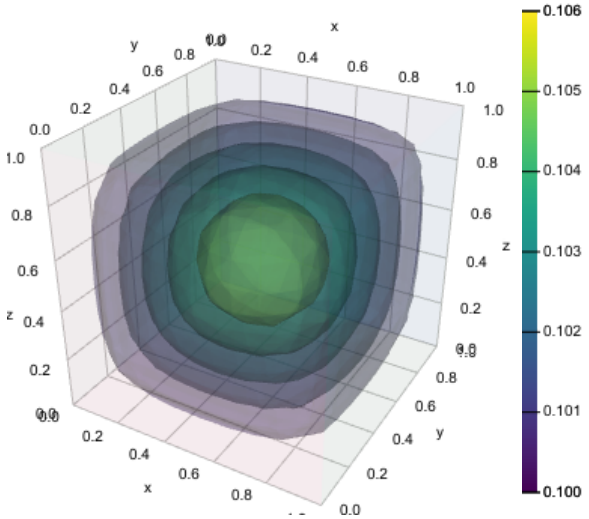

```
gridplot(grid3d,xplanes=[0.4],size=(400,400))
```

```
system3d =
VoronoiFVM.System{Float64, Int32, Int64, Matrix{Int32}, Matrix{Float64}}(num_species=1)
```

```
system3d=VoronoiFVM.System(grid3d;
         flux=diffusion_flux!,
         source=diffusion_source!,
         bcondition=dirichlet_bc!,
         species=[spec_idx])
```

```
sol3 =
1×1331 Matrix{Float64}:
 0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  …  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1
```

```
sol3=solve(system3d;inival=0)
```




```
scalarplot(grid3d,sol3,size=(400,400))
```

Nonlinear diffusion

Here, we define a nonlinear diffusion problem with diffusion coefficient depending on the solution:

Let $\vec{j} = -D(u)\vec{\nabla}u$ with $D(u) = u^2$. In order to obtain the diffusion coefficient along the discretization edge, we evaluate it a the average of the solutions at both ends of the discretization edge. Just note that there are more sophisticated ways to define this.

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

```
nlD(u)=u^2
```

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

```
function nldiffusion_flux!(f,u, edge)
    avgu=(u[spec_idx,1]+u[spec_idx,2])/2
    f[spec_idx]=nlD(avgu)*(u[spec_idx,1]-u[spec_idx,2])
end
```

1D Nonlinear diffusion

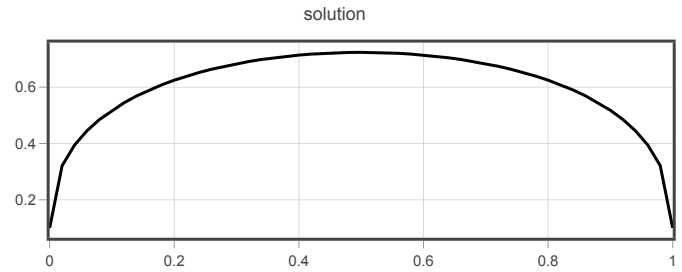
```
nlsystem1d =
VoronoiFVM.System{Float64, Int32, Int64, Matrix{Int32}, Matrix{Float64}}(num_species=1)
```

```
nlsystem1d=VoronoiFVM.System(grid1d;
        flux=nldiffusion_flux!,
        source=diffusion_source!,
        bcondition=dirichlet_bc!,
        species=[spec_idx])
```

```
(seconds = 1.18, iters = 13, absnorm = 8.32e-13, relnorm = 6.66e-14, roundoff = 2.1e-13,
```

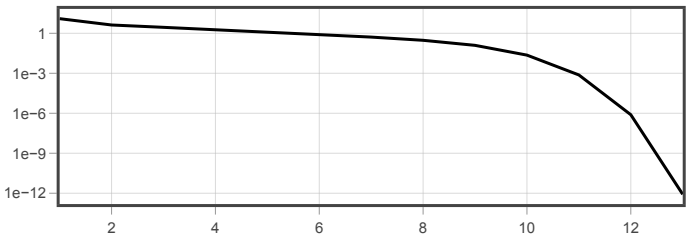
```
begin
    nlsolution1d=solve(nlsystem1d,inival=0.1, log=true)
    nlhistory1d=history(nlsystem1d)
    summary(nlhistory1d)
end
```

Here, Newton's method is used in order to solve the nonlinear system of equations. The Jacobi matrix is assembled from the partial derivatives of the flux function $g(u_k, u_l)$.




```
scalarplot(grid1d,nlsolution1d[1,:],size=(500,200),title="solution")
```

We can plot the solver history




```
scalarplot(nlhistory1d, yscale=:log, size=(500,200))
```

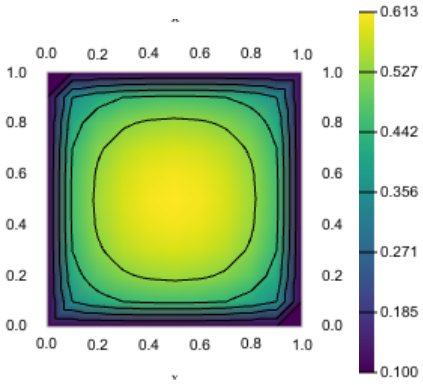
2D Nonlinear diffusion

```
nlsystem2d =
VoronoiFVM.System{Float64, Int32, Int64, Matrix{Int32}, Matrix{Float64}}(num_species=1)
```

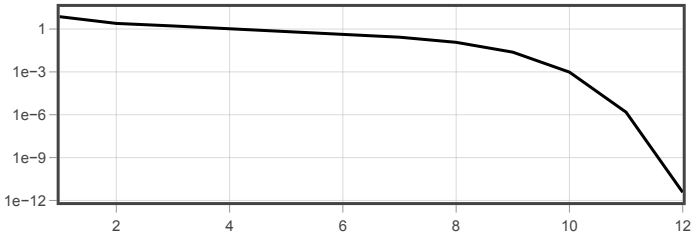
```
nlsystem2d=VoronoiFVM.System(grid2d;
        flux=nldiffusion_flux!,
        source=diffusion_source!,
        bcondition=dirichlet_bc!,
        species=[spec_idx])
```

```
(seconds = 1.18, iters = 13, absnorm = 8.32e-13, relnorm = 6.66e-14, roundoff = 2.1e-13,
```

```
begin
    nlsolution2d=solve(nlsystem2d,inival=0.1, log=true)
    nlhistory2d=history(nlsystem2d)
    summary(nlhistory1d)
end
```




```
scalarplot(grid2d,nlsolution2d[1,:],size=(300,300),title="solution")
```




```
scalarplot(nlhistory2d, yscale=:log,size=(500,200))
```

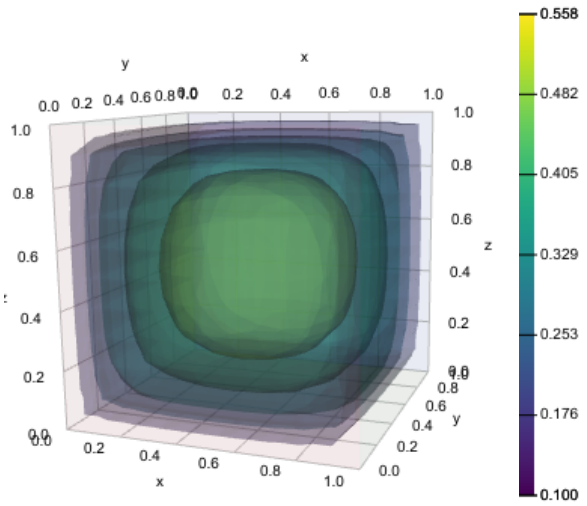
3D Nonlinear diffusion

```
nlsystem3d =
VoronoiFVM.System{Float64, Int32, Int64, Matrix{Int32}, Matrix{Float64}}(num_species=1)
```

```
nlsystem3d=VoronoiFVM.System(grid3d;
        flux=nldiffusion_flux!,
        source=diffusion_source!,
        bcondition=dirichlet_bc!,
        species=[spec_idx])
```

```
nlsol3d =
1×1331 Matrix{Float64}:
 0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  …  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1
```

```
nlsol3d=solve(nlsystem3d, inival=0.1)
```




```
scalarplot(grid3d,nlsol3d[1,:],size=(400,400))
```

Behind the scenes

In the previous lectures, we learned:

- how to generate discretization grids
- how to assemble linear systems of equations for the finite volume method into sparse matrices
- how to solve a nonlinear problem utilizing automatic differentiation

In VoronoiFVM.jl, these things are put together.

We already have shown how to assemble linear systems of equations from the finite volume method.

Assembling Jacobi matrices

We show how to assemble the Jacobi matrix for a nonlinear system of equations coming from the finite volume method.

Linear system of equations in 1D case:

$$Au = \begin{pmatrix} a_{11} & a_{12} & & & & \\ a_{21} & a_{22} & a_{23} & & & \\ & a_{32} & a_{33} & \ddots & & \\ & & \ddots & \ddots & a_{N-1,N} & \\ & & & a_{N,N-1} & a_{NN} & \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-1} \\ f_N \end{pmatrix}$$

$$\begin{aligned} a_{11}u_1 + a_{12}u_3 &= f_1 \\ a_{12}u_1 + a_{22}u_2 + a_{23}u_3 &= f_2 \\ a_{32}u_2 + a_{33}u_3 + a_{34}u_4 &= f_3 \\ &\vdots \\ a_{N-1,N}u_{N-1} + a_{NN}u_N N &= f_N \end{aligned}$$

Nonlinear system of equations `A(u)=f` in 1D case: as in the linear case, the equations only couple neigboring unknowns.

$$\begin{aligned} A_1(u_1, u_2) &= f_1 \\ A_2(u_1, u_2, u_3) &= f_2 \\ A_3(u_2, u_3, u_4) &= f_4 \\ &\vdots \\ A_N(u_{N-1}, u_N) &= f_N \end{aligned}$$

We have

$$\begin{aligned} A_i(u_1 \dots u_N) &= \frac{g(u_i, u_{i-1})}{h} + \frac{g(u_i - u_{i+1})}{h} \\ &= \sum_{j \in \mathcal{N}_i} \frac{|\sigma_{ij}|}{h_{ij}} g(u_i, u_j) \end{aligned}$$

with $g(u,v)\frac{u+v}{2}(u-v)$, in the case of nonlinear diffusion, so each contribution can be assembled by a calculation on the the corresponding discretization edge. This works in 1D, 2D, and even 3D case.

For a given equation i, the only dependencies come from unknowns in the neigbourhood of a given discretization point.

i-th step of Newton's method:

- Calculate residual: $r^i = A(u^i) - f$
- Solve linear system for update: $A'(u^i)h^i = r^i$
- Update solution: $u^{i+1} = u^i - h^i$

requires the calculation of the Jacobi matrix. Given the structure described above, we see, that the Jacobi matrix is sparse and can be assembled from contributions from the discretization edges:

$$A'(u)h = \begin{pmatrix} \frac{\partial A_1}{\partial u_1} & \frac{\partial A_1}{\partial u_2} & & & \\ \frac{\partial A_2}{\partial u_1} & \frac{\partial A_2}{\partial u_2} & \frac{\partial A_2}{\partial u_2} & & \\ & \frac{\partial A_3}{\partial u_2} & \frac{\partial A_3}{\partial u_3} & \ddots & \\ & & \ddots & \ddots & \frac{\partial A_{N-1}}{\partial A_N} \\ & & & \frac{\partial A_N}{\partial u_{N-1}} & \frac{\partial A_N}{\partial u_N} \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \\ \vdots \\ h_{N-1} \\ h_N \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_{N-1} \\ r_N \end{pmatrix}$$

$$\frac{\partial A_i(u_1 \dots u_N)}{\partial u_j} = \begin{cases} \sum_{k \in \mathcal{N}_i} \frac{|\sigma_{ik}|}{h_{ik}} \frac{\partial g(u_i, u_k)}{\partial u_i}, & j = i \\ \frac{|\sigma_{ij}|}{h_{ij}} \frac{\partial g(u_i, u_j)}{\partial u_j}, & j \in \mathcal{N}_i \\ 0, & \text{else} \end{cases}$$

As in the linear case, in the 2D case, assembly of $A(u)$ and the Jacobi matrix $A'(u)$ can be realized by a loop over all simplices of a triangulation.

Derivatives can be calculated locally, it is sufficient to calculate them from the constitutive functions on each edge. This is a convenient case to use automatic differentiation locally which can be very well performed by Julia's `ForwardDiff.jl`.
