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

```julia
begin
    using PlutoUI
    using HypertextLiteral: @htl, @htl_str
    using ExtendableGrids, VoronoiFVM, GridVisualize, PlutoVista
    GridVisualize.default_plotter!(PlutoVista)
end;
```

The Voronoi finite volume method for the discetization of PDEs

Motivation

Regard a stationary second order PDE with Robin boundary conditions as a system of two first order equations in a Lipschitz domain Ω:

$$\begin{aligned}
\nabla\cdot\vec{j} &= f && \text{continuity equation in } \Omega\\
\vec{j} &= -\delta\vec{\nabla}u && \text{flux law in } \Omega\\
-\vec{j}\cdot\vec{n}+\alpha u &= \beta && \text{on } \Gamma
\end{aligned}$$

- The derivation of the continuity equation was based on the consideration of species balances of an representative elementary volume (REV)
- Why not just subdivide the computational domain into a finite number of REV's ?
  - Assign a value of u to each REV
  - Call REVs *control volumes* or *finite volumes*

Constructing control volumes

Assume $\Omega \subset \mathbb{R}^d$ is a polygonal domain such that $\partial\Omega = \bigcup_{m\in\mathcal{G}} \Gamma_m$, where Γ_m are straight lines. We denote the normal vector $\vec{n}|_{\Gamma_m} = \vec{n}_m$.

Subdivide Ω into into a finite number of *control volumes* ω_k such that

- the closure of Ω is the union of the closures of the control volumes: $\bar{\Omega} = \bigcup_{k\in\mathcal{N}} \bar{\omega}_k$
- the control volumes ω_k are open convex domains
- two different control volumes don't intersect: $\omega_k \cap \omega_l = \emptyset$ if $\omega_k \neq \omega_l$
- The intersections of the closures $\sigma_{kl} = \bar{\omega}_k \cap \bar{\omega}_l$ are either empty, points or straight lines
    - If σ_{kl} is a straight line (d=2) or a point (d=1) we say that ω_k, ω_l are neighbours. We set $|\sigma_{kl}|$ to the length of that line or 1, respectively. Otherwise it is 0.
    - Let $\vec{n}_{kl} \perp \sigma_{kl}$ denote the normal of $\partial\omega_k$ at σ_{kl}
    - Let $\mathcal{N}_k = \{l \in \mathcal{N} : |\sigma_{kl}| > 0\}$ denote the set of neighbours of ω_k
- Assume that the boundary parts of $\partial\omega_k$ $\gamma_{km} = \partial\omega_k \cap \Gamma_m$ are straight lines (2D), points (1D) or empty.
    - $\mathcal{G}_k = \{m \in \mathcal{G} : |\gamma_{km}| > 0\}$: set of non-empty boundary parts of $\partial\omega_k$. Obviously, $\mathcal{G}_k = \emptyset$ for control volumes in the interior of Ω.
    - We always have $\partial\omega_k = \left(\cup_{l\in\mathcal{N}_k}\sigma_{kl}\right) \bigcup \left(\cup_{m\in\mathcal{G}_k}\gamma_{km}\right)$

To each control volume ω_k assign a *collocation point*: $\vec{x}_k \in \bar{\omega}_k$ such that

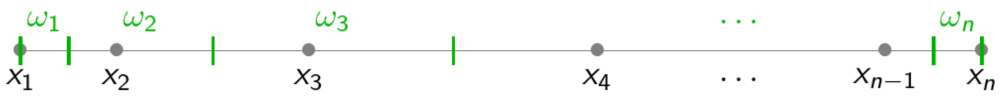
- *Admissibility condition*:if $l \in \mathcal{N}_k$ then the line $\vec{x}_k\vec{x}_l$ is orthogonal to σ_{kl}
    - For a function $u : \Omega \to \mathbb{R}$ this will allow to associate its value $u_k = u(\vec{x}_k)$ as the value of an unknown at $\vec{x}_k$.
    - For two neigboring control volumes ω_k, ω_l , this will allow to approximate $\vec{\nabla} u \cdot \vec{n}_{kl} \approx \frac{u_l - u_k}{h_{kl}}$ where $h_{kl} = |\vec{x}_k\vec{x}_l|$.
- *Placement of boundary unknowns at the boundary*: if ω_k is situated at the boundary, i.e. for $|\partial\omega_k \cap \partial\Omega| > 0$, then $\vec{x}_k \in \partial\Omega$
    - This will allow to apply boundary conditions in a direct manner
    - There are other possibilities to handle boundary conditions which do not require this placement condition

1D case

Let $\Omega = (a, b)$ be subdivided into intervals by $x_1 = a < x_2 < x_3 < \cdots < x_{n-1} < x_n = b$. Then we set

$$\omega_k = \begin{cases} \left(x_1, \frac{x_1+x_2}{2}\right), & k = 1 \\ \left(\frac{x_{k-1}+x_k}{2}, \frac{x_k+x_{k+1}}{2}\right), & 1 < k < n \\ \left(\frac{x_{n-1}+x_n}{2}, x_n\right), & k = n \end{cases}$$

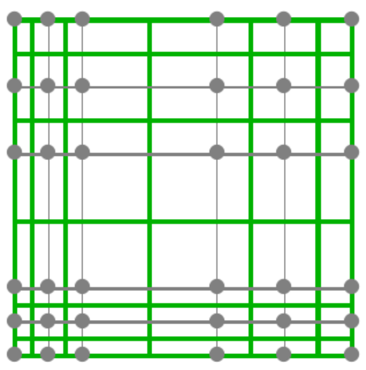
[



2D Rectangular domain

- Let $\Omega = (a, b) \times (c, d) \subset \mathbb{R}^2$.
- Assume subdivisions $x_1 = a < x_2 < x_3 < \cdots < x_{n-1} < x_n = b$ and $y_1 = c < y_2 < y_3 < \cdots < y_{n-1} < y_n = d$

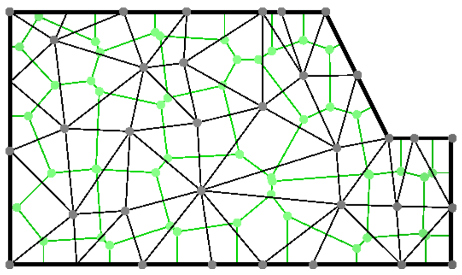
- $\Rightarrow$ 1D control volumes ω_k^x and ω_k^y
- Set $\vec{x}_{kl} = (x_k, y_l)$ and $\omega_{kl} = \omega_k^x \times \omega_l^y$.

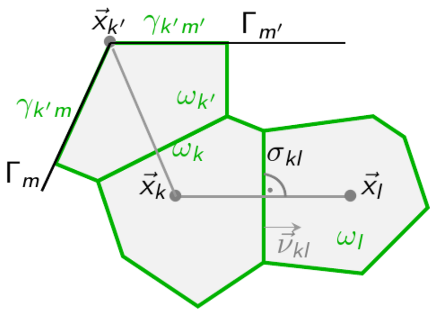


- Green: Control volume boundaries
- Gray: original grid lines and points

2D, polygonal domain

- Obtain a boundary conforming Delaunay triangulation with vertices $\vec{x}_k$
- Construct restricted Voronoi cells ω_k with $\vec{x}_k \in \omega_k$
- Corners of Voronoi cells are either cell circumcenters or midpoints of boundary edges
- Admissibility condition $\vec{x}_k\vec{x}_l \perp \sigma_{kl} = \bar{\omega}_k \cap \bar{\omega}_l$ fulfilled in a natural way
- Triangulation edges $\equiv$ connected neigborhood graph of Voronoi cells
- Triangulation nodes $\equiv$ collocation points
- Boundary placement of collocation points of boundary control volumes





Discretization of second order PDE

Discretization of continuity equation

- Stationary continuity equation: $\nabla \cdot \vec{j} = f$
- Integrate over control volume ω_k:

$$
\begin{aligned}
0 &= \int_{\omega_k} \nabla \cdot \vec{j}\, d\omega - \int_{\omega_k} f\, d\omega \\
&= \int_{\partial\omega_k} \vec{j} \cdot \vec{n}_\omega\, ds - \int_{\omega_k} f\, d\omega \\
&= \sum_{l \in \mathcal{N}_k} \int_{\sigma_{kl}} \vec{j} \cdot \vec{n}_{kl}\, ds + \sum_{m \in \mathcal{G}_k} \int_{\gamma_{km}} \vec{j} \cdot \vec{n}_m\, ds - \int_{\omega_k} f d\omega \\
&= \text{flux between CV} + \text{flux in/out of } \Omega - \text{sources}
\end{aligned}
$$

We have seen this during the derivation of conservation laws.

Approximation of flux between control volumes

Utilize flux law: $\vec{j} = -\delta\vec{\nabla}u$ and admissibility condition which results in $\vec{x}_k\vec{x}_l \parallel \vec{n}_{kl}$

- Let $u_k = u(\vec{x}_k)$, $u_l = u(\vec{x}_l)$
- $h_{kl} = |\vec{x}_k - \vec{x}_l|$: distance between neigboring collocation points
- Finite difference approximation of normal derivative:

$$\vec{\nabla}u \cdot \vec{n}_{kl} \approx \frac{u_l - u_k}{h_{kl}}$$

- $\Rightarrow$ flux between neigboring control volumes:

$$\begin{aligned}\int_{\sigma_{kl}} \vec{j} \cdot \vec{n}_{kl}\, ds &\approx \frac{|\sigma_{kl}|}{h_{kl}}\delta(u_k - u_l)\\ &=: \frac{|\sigma_{kl}|}{h_{kl}} g(u_k, u_l)\end{aligned}$$

where $g(\cdot,\cdot)$ is called *flux function*

Approximation of boundary fluxes

- Utilize boundary condition $\vec{j} \cdot \vec{n} = \alpha u - \beta$
- Assume $\alpha|_{\Gamma_m} = \alpha_m$, $\beta|_{\Gamma_m} = \beta_m$
- Approximation of $\vec{j} \cdot \vec{n}_m$ at the boundary of ω_k:

$$\vec{j} \cdot \vec{n}_m \approx \alpha_m u_k - \beta_m$$

- Approximation of flux from ω_k through Γ_m:

$$\int_{\gamma_{km}} \vec{j} \cdot \vec{n}_m\, ds \approx |\gamma_{km}|(\alpha_m u_k - \beta_m)$$

Approximation of right hand side

- Let $f_k = \frac{1}{|\omega_k|}\int_{\omega_k} f(\vec{x})\, d\omega$ or $f_k = f(\vec{x}_k)$
- Approximate $\int_{\omega_k} f\, d\omega \approx |\omega_k| f_k$

Discretized system of equations

- The discrete system of equations then writes for $k \in \mathcal{N}$:

$$\sum_{l \in \mathcal{N}_k} \frac{|\sigma_{kl}|}{h_{kl}}\delta(u_k - u_l) + \sum_{m \in \mathcal{G}_k} |\gamma_{km}|\alpha_m u_k = |\omega_k| f_k + \sum_{m \in \mathcal{G}_k} |\gamma_{km}|\beta_m$$

$$\left(\delta \sum_{l \in \mathcal{N}_k} \frac{|\sigma_{kl}|}{h_{kl}} + \alpha_m \sum_{m \in \mathcal{G}_k} |\gamma_{km}|\right) u_k - \sum_{l \in \mathcal{N}_k} \delta \frac{|\sigma_{kl}|}{h_{kl}} u_l = |\omega_k| f_k + \sum_{m \in \mathcal{G}_k} |\gamma_{km}|\beta_m$$

- This can be rewritten as matrix equation $Au = b$ such that

$$a_{kk}u_k + \sum_{l=1\ldots|\mathcal{N}|, l \neq k} a_{kl}u_l = b_k \qquad \text{for } k = 1\ldots|\mathcal{N}|$$

with coefficients

$$a_{kl} = \begin{cases} \sum_{l'\in\mathcal{N}_k} \delta\frac{|\sigma_{kl'}|}{h_{kl'}} + \sum_{m\in\mathcal{G}_k} |\gamma_{km}|\alpha_m, & l = k \\ -\delta\frac{\sigma_{kl}}{h_{kl}}, & l \in \mathcal{N}_k \\ 0, & \text{else} \end{cases}$$

$$b_k = |\omega_k| f_k + \sum_{m\in\mathcal{G}_k} |\gamma_{km}|\beta_m$$

Matrix properties

- $N = |\mathcal{N}|$ equations (one for each control volume ω_k)
- $N = |\mathcal{N}|$ unknowns (one for each collocation point $x_k \in \omega_k$)
- Matrix is sparse: nonzero entries only for neighboring control volumes
- Matrix graph is connected: nonzero entries correspond to edges in Delaunay triangulation $\Rightarrow$ irreducible
- A is irreducibly diagonally dominant if at least for one i, $|\gamma_{i,k}|\alpha_i > 0$
- Main diagonal entries are positive, off diagonal entries are non-positive
- $\Rightarrow A$ has the M-property.
- A is symmetric $\Rightarrow A$ is positive definite

- Due to the connection between Voronoi diagram and Delaunay triangulation, one can assemble the discrete system based on the triangulation
- Assembly in two loops:
  - Loop over all triangles, calculate triangle contribution to matrix entries
  - Loop over all boundary segments, calculate contribution to matrix entries

- One solution value per control volume ω_k allocated to the collocation point $x_k \Rightarrow$ piecewise constant function on collection of control volumes
- But: x_k are at the same time nodes of the corresponding Delaunay mesh $\Rightarrow$ representation as piecewise linear function on triangles

Penalty method for Dirichlet boundary conditions

A Dirichlet boundary condition $u|_\Gamma = \beta_D$ can be approximated using the so-called penalty method. For a small value ε, regard the a Robin condition

$$-\vec{j}\cdot\vec{n} + \alpha u = \beta$$

with $\alpha = \frac{1}{\varepsilon}$ and $\beta = \frac{1}{\varepsilon}\beta_D$. If ε is small, the solution of the Robin problem will closely approximate the solution of the Dirichlet problem. In fact, an implementation can be chosen in such a way, that in the floating point aritmetic, the approximation is exact.

This approach significantly eases the implementation.

VoronoiFVM.jl

The VoronoiFVM.jl Julia package implements the Voronoi finite volume method for systems of nonlinear PDEs.

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

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

```
const spec_idx = 1
```

```julia
const spec_idx = 1
```

- Diffusion coefficient D:

```
const D = 10.0
```

```julia
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)
```

```julia
function diffusion_flux!(f, u, edge, data)
    f[spec_idx] = D * (u[spec_idx, 1] - u[spec_idx, 2])
    return nothing
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)
```

```julia
function diffusion_source!(f, node, data)
    f[spec_idx] = 1
    return nothing
end
```

- Boundary value β:

```
const β = 0.1
```

```julia
const 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)
```

```julia
function dirichlet_bc!(f, u, bnode, data)
    boundary_dirichlet!(f, u, bnode; value = β)
    return nothing
end
```

1D Discretization grid

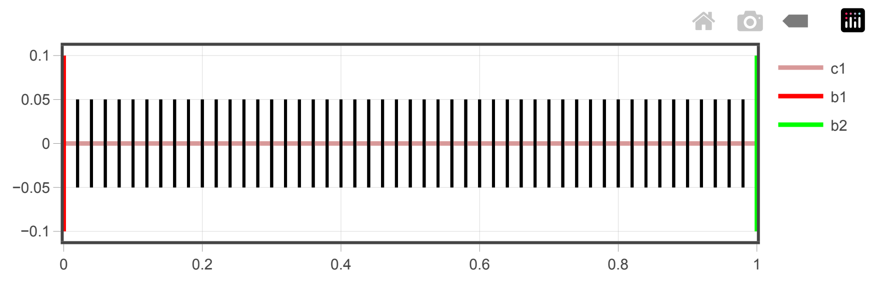
Grid in domain $\Omega = (0, 1)$ consisting of N= **51** points.

```
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
```

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

```
grid1d = ExtendableGrids.ExtendableGrid{Float64, Int32}
            dim =       1
         nnodes =      51
         ncells =      50
        nbfaces =       2
```

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



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

System creation and solution

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

```
system1d =
VoronoiFVM.System{Float64, Float64, Int32, Int64, Matrix{Int32}}(
  grid = ExtendableGrids.ExtendableGrid{Float64, Int32}(dim=1, nnodes=51, ncells=50,
  nbfaces=2),
  physics = Physics(flux=diffusion_flux!, storage=default_storage,
  source=diffusion_source!, breaction=dirichlet_bc!, ),
  num_species = 1)
```

```julia
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.

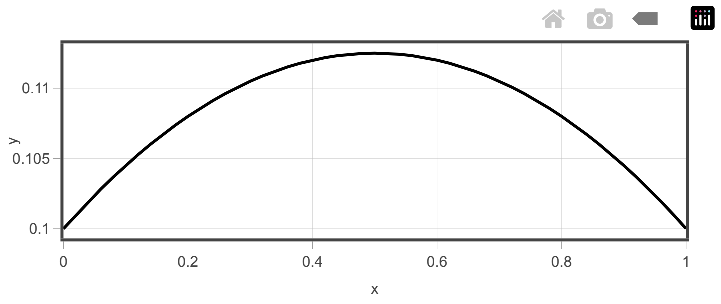
```
▸(seconds = 4.99, tasm = 4.0, tlinsolve = 0.263, iters = 2, absnorm = 9.6e-16, relnorm = 8.53e-
```

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

```
1×51 VoronoiFVM.DenseSolutionArray{Float64, 2}:
 0.1  0.10098  0.10192  0.10282  0.10368  …  0.10368  0.10282  0.10192  0.10098  0.1
```

```julia
solution
```

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



```julia
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.

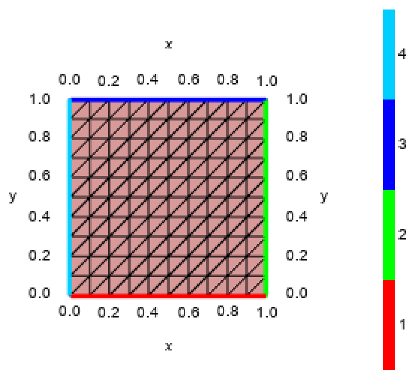
Grid in domain $\Omega = (0,1) \times (0,1)$ consisting of N2=**11** points in each coordinate direction

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

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

```
grid2d = ExtendableGrids.ExtendableGrid{Float64, Int32}
              dim =        2
           nnodes =      121
           ncells =      200
          nbfaces =       40
```

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



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

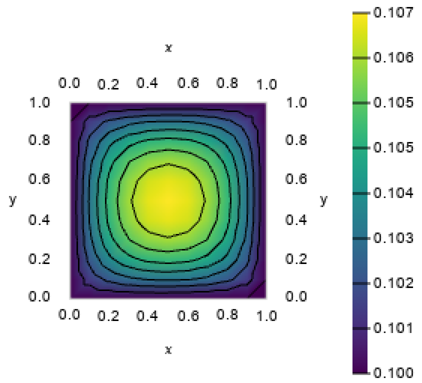
We can define and solve the 2D problem with the same physics functions as the 1D problem:

```
system2d =
VoronoiFVM.System{Float64, Float64, Int32, Int64, Matrix{Int32}}(
  grid = ExtendableGrids.ExtendableGrid{Float64, Int32}(dim=2, nnodes=121, ncells=200,
  nbfaces=40),
  physics = Physics(flux=diffusion_flux!, storage=default_storage,
  source=diffusion_source!, breaction=dirichlet_bc!, ),
  num_species = 1)
```

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

```
▸ (seconds = 0.00352, tasm = 0.0028, tlinsolve = 0.000652, iters = 2, absnorm = 7.05e-17, relno
```

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

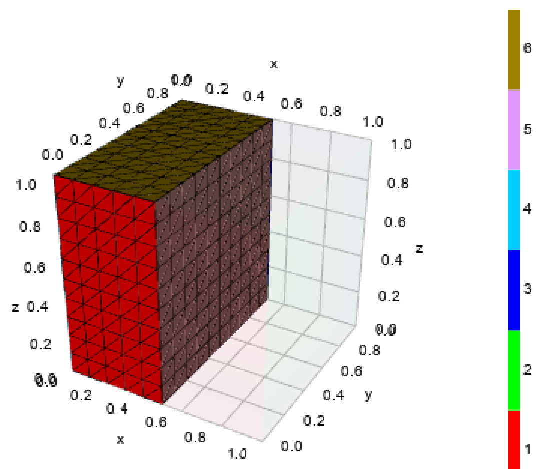


```
scalarplot(grid2d, solution2d[1, :]; size = (300, 300))
```

3D Linear diffusion

```
grid3d = ExtendableGrids.ExtendableGrid{Float64, Int32}
          dim =        3
       nnodes =     1331
       ncells =     6000
      nbfaces =     1200
```

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



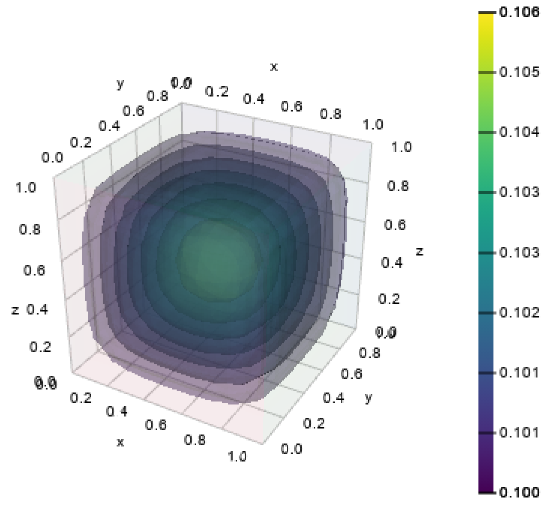
```
gridplot(grid3d; xplanes = [0.4], size = (400, 400))
```

```
system3d =
VoronoiFVM.System{Float64, Float64, Int32, Int64, Matrix{Int32}}(
  grid = ExtendableGrids.ExtendableGrid{Float64, Int32}(dim=3, nnodes=1331, ncells=600
  nbfaces=1200),
  physics = Physics(flux=diffusion_flux!, storage=default_storage,
  source=diffusion_source!, breaction=dirichlet_bc!, ),
  num_species = 1)
```

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

```
sol3 =
1×1331 VoronoiFVM.DenseSolutionArray{Float64, 2}:
 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[1, :]; 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)
```

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

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

```julia
function nldiffusion_flux!(f, u, edge, data)
    avgu = (u[spec_idx, 1] + u[spec_idx, 2]) / 2
    f[spec_idx] = nlD(avgu) * (u[spec_idx, 1] - u[spec_idx, 2])
    return nothing
end
```

1D Nonlinear diffusion

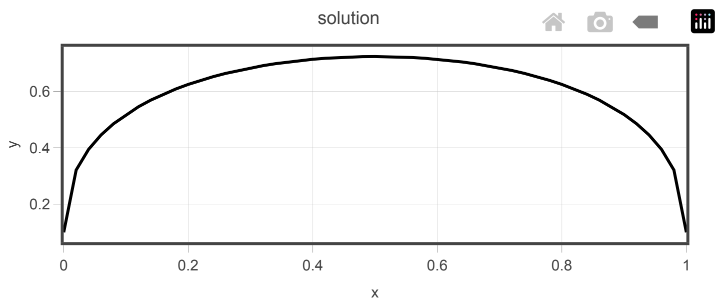
```
nlsystem1d =
VoronoiFVM.System{Float64, Float64, Int32, Int64, Matrix{Int32}}(
  grid = ExtendableGrids.ExtendableGrid{Float64, Int32}(dim=1, nnodes=51, ncells=50,
  nbfaces=2),
  physics = Physics(flux=nldiffusion_flux!, storage=default_storage,
  source=diffusion_source!, breaction=dirichlet_bc!, ),
  num_species = 1)
```

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

```
▸(seconds = 0.528, tasm = 0.528, tlinsolve = 0.000319, iters = 13, absnorm = 8.32e-13, relnorm
```

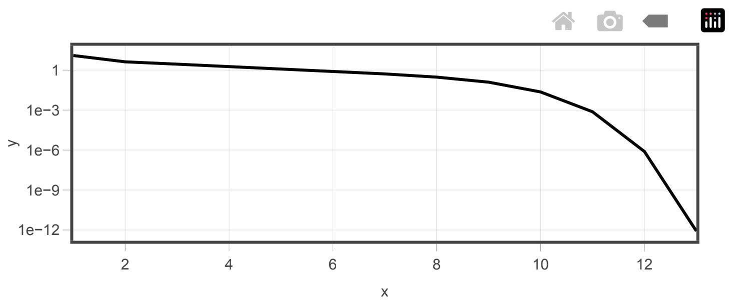
```julia
begin
    nlsolution1d = solve(nlsystem1d; inival = 0.1, log = true)
    nlhistory1d = history(nlsolution1d)
    summary(nlhistory1d)
end
```

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 $g(u_k, u_l)$.



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

We can plot the solver history



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

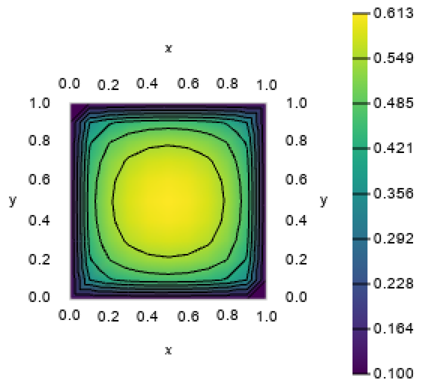
2D Nonlinear diffusion

```
nlsystem2d =
VoronoiFVM.System{Float64, Float64, Int32, Int64, Matrix{Int32}}(
  grid = ExtendableGrids.ExtendableGrid{Float64, Int32}(dim=2, nnodes=121, ncells=200,
  nbfaces=40),
  physics = Physics(flux=nldiffusion_flux!, storage=default_storage,
  source=diffusion_source!, breaction=dirichlet_bc!, ),
  num_species = 1)
```

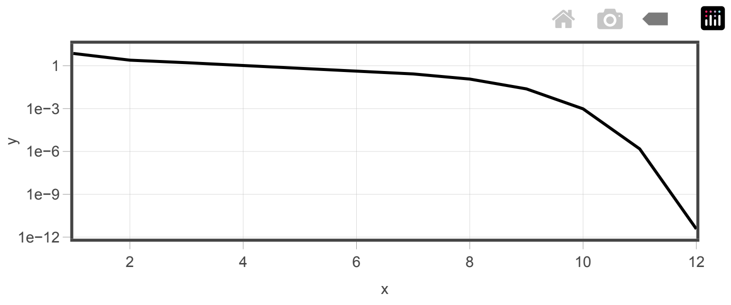
```
nlsystem2d = VoronoiFVM.System(
    grid2d;
    flux = nldiffusion_flux!,
    source = diffusion_source!,
    bcondition = dirichlet_bc!,
    species = [spec_idx]
)
```

```
▸ (seconds = 0.0171, tasm = 0.0154, tlinsolve = 0.00168, iters = 12, absnorm = 3.71e-12, relnorr
```

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



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



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

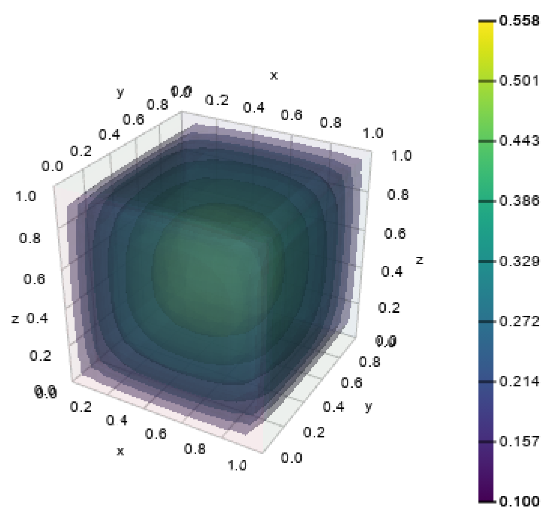
3D Nonlinear diffusion

```
nlsystem3d =
VoronoiFVM.System{Float64, Float64, Int32, Int64, Matrix{Int32}}(
  grid = ExtendableGrids.ExtendableGrid{Float64, Int32}(dim=3, nnodes=1331, ncells=600(
  nbfaces=1200),
  physics = Physics(flux=nldiffusion_flux!, storage=default_storage,
  source=diffusion_source!, breaction=dirichlet_bc!, ),
  num_species = 1)
```

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

```
nlsol3d =
1×1331 VoronoiFVM.DenseSolutionArray{Float64, 2}:
 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

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 and can be generalized to the 2D and 3D cases.

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

An iteration step (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:

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

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

Assembly of $A(u)$ and the Jacobi matrix $A'(u)$ can be realized by a loop over all simplices of a triangulation.

Derivatives are be calculated locally from the constitutive functions on each edge resp. node using forward moda automatic differentiaton performed by Julia's `ForwardDiff.jl` package.