```
begin
    using PlutoUI: PlutoUI, Resource
    import CairoMakie
end
```

Table of Contents

Introduction to VoronoiFVM.jl

v1.0, 2024-11-17

VoronoiFVM.jl is a two point flux finite volume solver for nonlinear systems of PDEs on one-, two- and three-dimensional grids written in Julia.

As a Julia package, it depends on several packages from the Julia ecosystem. Several of them have been developed together with VoronoiFVM and belong to the WIAS-PDELib github organization.

Discretization apporach

Nonlinear Reaction-Diffusion-Convection

- n nonlinear parabolic/elliptic coupled PDEs in $\Omega \subset \mathbb{R}^d$, d=1,2,3, time interval $[0,T]$:
- **Bold face**: n-vectors, $\vec{\mathbf{arrows}}$: d-vectors.
- Find $\mathbf{u}(\vec{x},t) = (u_1(\vec{x},t) \dots u_n(\vec{x},t)) : \Omega \times [0,T] \to \mathbb{R}^n$ such that

$$\begin{aligned}
\partial_t s_1(\mathbf{u}) - \nabla \cdot \vec{j}_1(\mathbf{u}, \vec{\nabla}\mathbf{u}) + r_1(\mathbf{u}) &= f_1 \\
&\vdots \\
\partial_t s_n(\mathbf{u}) - \nabla \cdot \vec{j}_n(\mathbf{u}, \vec{\nabla}\mathbf{u}) + r_n(\mathbf{u}) &= f_n
\end{aligned}$$

In vector form:

$$\partial_t \mathbf{s}(\mathbf{u}) - \nabla \cdot \vec{\mathbf{j}}(\mathbf{u}, \vec{\nabla}\mathbf{u}) + \mathbf{r}(\mathbf{u}) = \mathbf{f}$$

- "Storage" $\mathbf{s} : \mathbb{R}^n \to \mathbb{R}^n$
- "Reaction" $\mathbf{r} : \mathbb{R}^n \to \mathbb{R}^n$
- "Flux" $\vec{\mathbf{j}} : \mathbb{R}^n \times \mathbb{R}^{nd} \to \mathbb{R}^{nd}$
- "Source" $\mathbf{f} : \Omega \to \mathbb{R}^n$
- $\mathbf{s}, \vec{\mathbf{j}}, \mathbf{r}, \mathbf{f}$ can depend on $\vec{x}, t$ as well.

Boundary conditions

A similar approach describes nonlinear Robin boundary conditions on $\partial\Omega$:

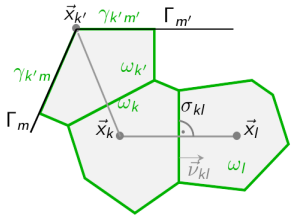
$$\begin{aligned}
j_1(\mathbf{u}, \vec{\nabla}\mathbf{u}) \cdot \vec{n} + a_1(\mathbf{u}) &= b_1 \\
&\vdots \\
j_n(\mathbf{u}, \vec{\nabla}\mathbf{u}) \cdot \vec{n} + a_n(\mathbf{u}) &= b_n
\end{aligned}$$

or

$$\vec{\mathbf{j}}(\mathbf{u}, \vec{\nabla}\mathbf{u}) \cdot \vec{n} + \mathbf{a}(\mathbf{u}) = \mathbf{b}$$

- "Boundary reaction" $\mathbf{a} : \mathbb{R}^n \to \mathbb{R}^n$
- "Boundary source" $\mathbf{b} : \partial\Omega \to \mathbb{R}^n$

Two point flux Voronoi finite volumes



Ω | > triangulation | > **Voronoi cells** $\equiv$ control volumes

$$\int_{\omega_k} \nabla \cdot \vec{\mathbf{j}}\, d\vec{x} = \int_{\partial\omega_k} \vec{\mathbf{j}} \cdot \vec{n}\, ds$$

Approximate flux between neigboring control volumes ω_k, ω_l by finite difference expression $\mathbf{g}(\mathbf{u}_k, \mathbf{u}_l)$ involving $\mathbf{u}_k = \mathbf{u}(\vec{x}_k)$, $\mathbf{u}_l = \mathbf{u}(\vec{x}_l)$.

Subdivision of the time interval: $0 = t_0 < t_1 < \cdots < t_{M-1} < t_M = T$

Let N be the number of control volumes ω_k / collocation points $\vec{x}_k$. For $k = 1 \ldots N$, $i = 2 \ldots M$, $t = t_i$, $t^{\text{old}} = t_{i-1}$, assuming $u_k = u(\vec{x}_k, t_i), u_k^{\text{old}} = u(\vec{x}_k, t_{i-1})$, the implicit Euler two point flux scheme writes

$$|\omega_k| \frac{\mathbf{s}(\mathbf{u}_k) - \mathbf{s}(\mathbf{u}_k^{\text{old}})}{t - t^{\text{old}}} + \sum_{l \in \mathcal{N}_k} \frac{|\sigma_{kl}|}{h_{kl}} \mathbf{g}(\mathbf{u}_k, \mathbf{u}_l) + |\omega_k| \mathbf{r}(\mathbf{u}_k) + |\gamma_k| \mathbf{a}(\mathbf{u}_k) = |\omega_k| \mathbf{f}_k + |\gamma_k| \mathbf{b}_k$$

- ω_k: control volume
- γ_k: boundary interface ($\emptyset$ for interior nodes)
- σ_{kl}: interface between neigboring control volumes
- h_{kl}: distance between neigboring collocation points

With exception of $\vec{\mathbf{j}}$, all constitutive functions introduced above can be used in the discrete version as well. The flux $\vec{\mathbf{j}}$ is replaced by the discrete edge flux $\mathbf{g} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$.

Dirichlet boundary conditions can be described within this formulation via the penalty method.

As a consequence, the finite volume scheme can be described by the discretization **grid** giving rise to the geometry derived data ω_k,γ_k, σ_{kl}, h_{kl} and the **physics** function

- `flux` $\mathbf{g} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$
- `storage` $\mathbf{s} : \mathbb{R}^n \to \mathbb{R}^n$
- `reaction` $\mathbf{r} : \mathbb{R}^n \to \mathbb{R}^n$
- `source` $\mathbf{f} : \Omega \to \mathbb{R}^n$
- `breaction` $\mathbf{a} : \mathbb{R}^n \to \mathbb{R}^n$
- `bsource` $\mathbf{b} : \partial\Omega \to \mathbb{R}^n$

How to create the Voronoi cells ?

The Voronoi cells are created from Delaunay triangulations. As a demonstration, we show how to use `Triangulate.jl`, a Julia wrapper of J. R. Shewchuk's `Triangle` code to create a Voronoi tesselation. It is created from a boundary conforming Delaunay grid (warranted by the "D"-flag in the control string).

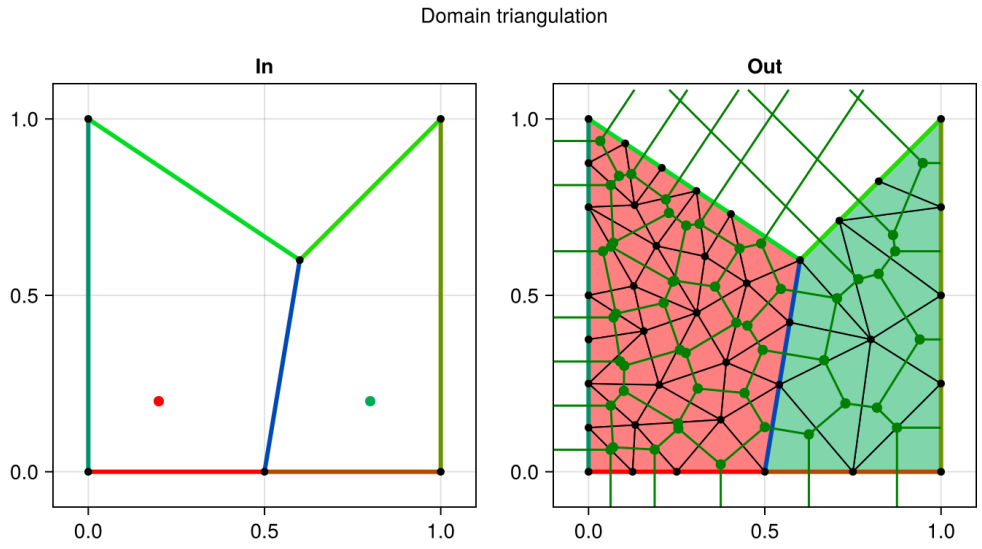
```
1 using Triangulate: Triangulate, TriangulateIO, triangulate, plot_in_out
```

```
TriangulateIO(
pointlist=[0.0 0.5 … 0.6 0.0; 0.0 0.0 … 0.6 1.0],
segmentlist=Int32[1 2 … 6 2; 2 3 … 1 5],
segmentmarkerlist=Int32[1, 2, 3, 4, 5, 6, 7],
regionlist=[0.2 0.8; 0.2 0.2; 1.0 3.0; 0.02 0.05],
)
```

```
begin
    triin = TriangulateIO()
    # Point describing the boundary
    triin.pointlist =
        Matrix{Cdouble}([0.0 0.0; 0.5 0.0; 1.0 0.0; 1.0 1.0; 0.6 0.6; 0.0 1.0]')
    # Boundary segments
    triin.segmentlist = Matrix{Cint}([1 2; 2 3; 3 4; 4 5; 5 6; 6 1; 2 5]')
    # Boundary segment markers
    triin.segmentmarkerlist = Vector{Int32}([1, 2, 3, 4, 5, 6, 7])
    # Region points, region markers, region volumes
    triin.regionlist = Matrix{Cdouble}([0.2 0.8; 0.2 0.2; 1 3; 0.02 0.05])
    triin
end
```

```
▸ (TriangulateIO(
  pointlist=[0.0 0.5 … 0.306009 0.800751; 0.0 0.0 … 0.795994 0.375],
```

```
triout, vorout = triangulate("paADq20Qv", triin)
```




```
plot_in_out(
    CairoMakie,
    triin,
    triout;
    figure = CairoMakie.Figure(; size = (700, 400)),
    voronoi = vorout,
    title = "Domain triangulation",
)

```

The output of `triangulate` needs to be converted to a computational grid for VoronoiFVM.jl. These grids are managed by the ExtendableGrids.jl package.

```
begin
    using ExtendableGrids: simplexgrid, bfacemask!
    using ExtendableGrids: Coordinates, CellNodes, CellRegions, BFaceNodes,
BFaceRegions
end
```

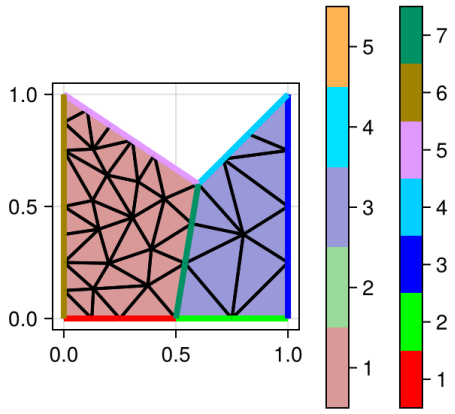
```
grid = ExtendableGrids.ExtendableGrid{Float64, Int32}
           dim =       2
         nnodes =      38
         ncells =      50
        nbfaces =      27
```

```
grid = simplexgrid(triout)
```

These grids and functions on them can be plotted using the `GridVisualize.jl` package.

```
CairoMakie
```

```
begin
    using GridVisualize: GridVisualize, GridVisualizer
    using GridVisualize: scalarplot, scalarplot!, gridplot, reveal
    GridVisualize.default_plotter!(CairoMakie)
end
```




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

The visualization reflects the basic information held by an `ExtendableGrid`. It consists of:

- Point coordinates:

```
2×38 Matrix{Float64}:
 0.0  0.5  1.0  1.0  0.6  0.0  1.0  1.0   …  0.13245  0.130716  0.306009  0.800751
 0.0  0.0  0.0  1.0  0.6  1.0  0.5  0.75     0.13245  0.755755  0.795994  0.375
```

```
grid[Coordinates]
```

- Simplex connectivity:

```
3×50 Matrix{Int32}:
 16  33  16  8  10  38   5  15   8   8  14  …  13  19  26  26  14  28  18  38  31  11
 23  30   2  4   5  12  21   7  10   9   2      32  33  34   1  35  36  30   7  37  38
 35  18  11  9  38  15  38  38   7  10  16      33  32  35  34  34  17  36  10  30  21
```

```
grid[CellNodes]
```

- Cell region markers (color coded in pastel colors):

```
▸[1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 1, 3, 1, 1, 3, 1, 1, 1, 1, 1, ⋯ more ,1, 1, 1, 1, 1, 1, 1, 3, 1,
```

```
grid[CellRegions]
```

- Exterior and interior boundary faces:

```
2×27 Matrix{Int32}:
  2   3  4  10  28   1   2   7  8  9  10  …  18  21  22  24  25  26  28  29  34  37
 14  12  8   5   6  26  11  15  7  4   9      29  11  25   5  13  22  17   6   1  24
```

```
grid[BFaceNodes]
```

- Boundary region markers (color coded in bright colors):

```
▸[1, 2, 3, 4, 5, 6, 7, 3, 3, 4, 4, 7, 2, 6, 1, 3, 5, 6, 7, 6, 5, 6, 6, 5, 6, 1, 5]
```

```
grid[BFaceRegions]
```

More information can be created on demand, but is mostly not necessary for VoronoiFVM.jl

Grid generation examples

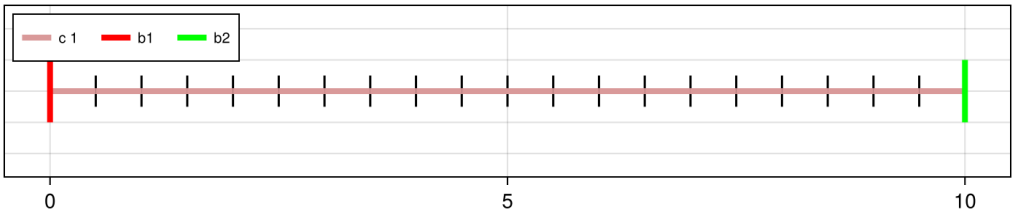
ExtendableGrids provides a number of simpler grid generation methods.

```
X = 0.0:0.5:10.0
```

```
1 X = 0:0.5:10
```

```
grid1d = ExtendableGrids.ExtendableGrid{Float64, Int32}
              dim =       1
            nnodes =      21
            ncells =      20
          nbfaces =       2
```

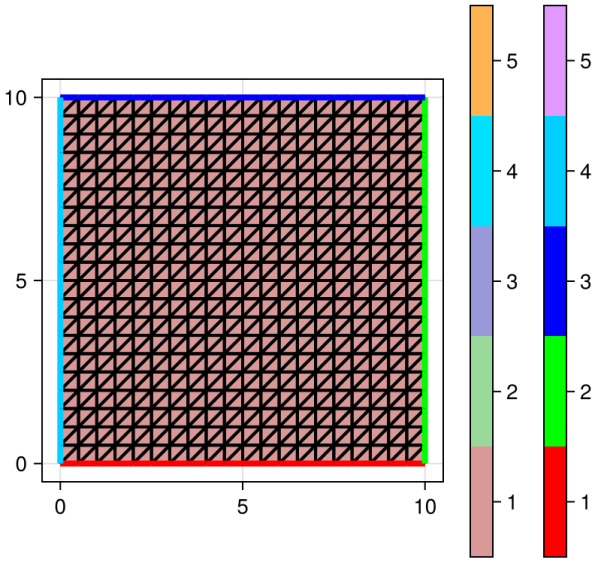
```
1 grid1d = simplexgrid(X)
```




```
1 gridplot(grid1d, size = (700, 150), legend = :lt)
```

```
grid2d = ExtendableGrids.ExtendableGrid{Float64, Int32}
              dim =       2
            nnodes =     441
            ncells =     800
          nbfaces =      80
```

```
1 grid2d = simplexgrid(X, X)
```

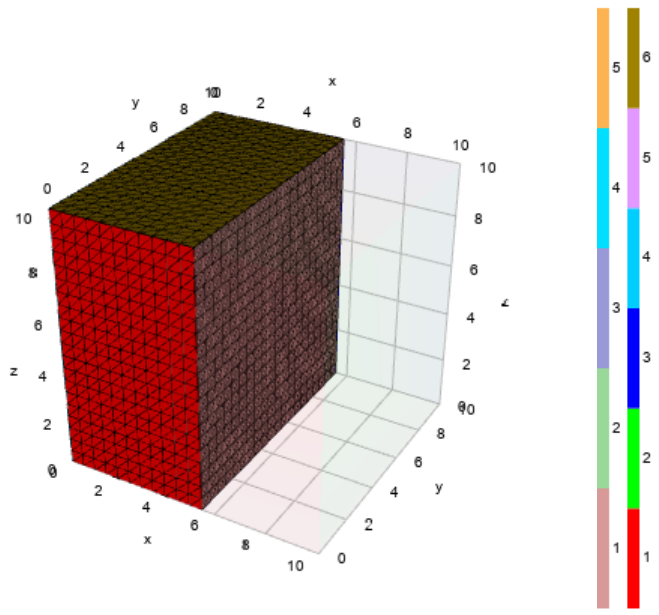



```
1 gridplot(grid2d, size = (400, 400))
```

```
grid3d = ExtendableGrids.ExtendableGrid{Float64, Int32}
              dim =       3
            nnodes =    9261
            ncells =   48000
          nbfaces =    4800
```

```
1 grid3d = simplexgrid(X, X, X)
```

```
1 import PlutoVista
```



```
gridplot(grid3d, Plotter = PlutoVista, xplanes = [5])
```

```
using SimplexGridFactory: SimplexGridBuilder, point!, facet!, facetregion!,
options!
```

```
ExtendableGrids.ExtendableGrid{Float64, Int32}
      dim =       2
   nnodes =     187
   ncells =     306
  nbfaces =      66
```

```
begin
    builder = SimplexGridBuilder(; Generator = Triangulate)

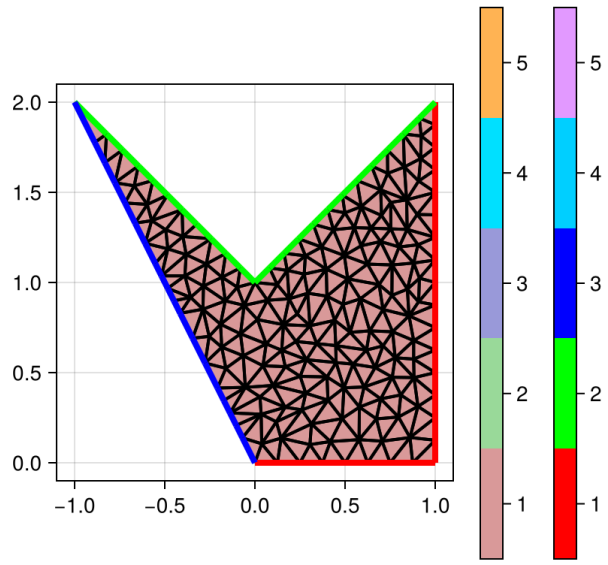
    p1 = point!(builder, 0, 0)
    p2 = point!(builder, 1, 0)
    p3 = point!(builder, 1, 2)
    p4 = point!(builder, 0, 1)
    p5 = point!(builder, -1, 2)

    facetregion!(builder, 1)
    facet!(builder, p1, p2)
    facet!(builder, p2, p3)
    facetregion!(builder, 2)
    facet!(builder, p3, p4)
    facet!(builder, p4, p5)
    facetregion!(builder, 3)
    facet!(builder, p5, p1)

    options!(builder; maxvolume = 0.01)

    grid2dsf = simplexgrid(builder)

end
```




```
gridplot(grid2dsf, size = (400, 400))
```

Examples I

```
begin
    using VoronoiFVM: VoronoiFVM, solve, ramp, unknowns
    using VoronoiFVM: boundary_dirichlet!, boundary_neumann!
end
```

1D scalar stationary reaction-diffusion

Let $\Omega = (0,1)$, $\Gamma_1 = \{0\}$, $\Gamma_2 = \{1\}$ solve

$$\begin{aligned} -\nabla \cdot (D\nabla u) + Ru^2 &= 0 \\ u|_{\Gamma_1} &= 1 \\ u|_{\Gamma_2} &= 0 \end{aligned}$$

- Specify grid:

```
rd1dgrid = ExtendableGrids.ExtendableGrid{Float64, Int32}
               dim =       1
             nnodes =     101
             ncells =     100
           nbfaces =       2
```

```
rd1dgrid = simplexgrid(0:0.01:1)
```

- Specify flux, reaction, boundary condition

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

```
function rdflux(y, u, edge, data)
    y[1] = data.D * (u[1, 1] - u[1, 2])
    end
```

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

```
function rdreaction(y, u, node, data)
        y[1] = data.R * u[1]^2
end
```

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

```
function rdbc(y, u, bnode, data)
    boundary_dirichlet!(y, u, bnode, region = 1, value = 1, species = 1)
    boundary_dirichlet!(y, u, bnode, region = 2, value = 0, species = 1)
    boundary_neumann!(y,u,bnode, region=3, value=0)
end
```

- Specify parameters, create and solve system:

```
rd1ddata = (D = 2, R = 100)
```

```
rd1ddata = (D = 2, R = 100)
```

```
rdphysics =
Physics(flux=rdflux, storage=default_storage, reaction=rdreaction, breaction=rdbc, )
```

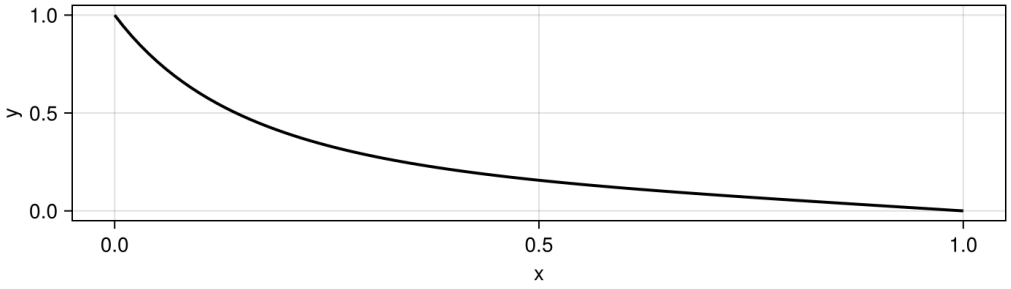
```
rdphysics=VoronoiFVM.Physics(flux=rdflux, reaction=rdreaction, breaction=rdbc)
```

```
rd1dsystem =
VoronoiFVM.System{Float64, Float64, Int32, Int64, Matrix{Int32}}(
  grid = ExtendableGrids.ExtendableGrid{Float64, Int32}(dim=1, nnodes=101, ncells=100,
  nbfaces=2),
  physics = Physics(flux=rdflux, storage=default_storage, reaction=rdreaction,
  breaction=rdbc, ),
  num_species = 1)
```

```
rd1dsystem = VoronoiFVM.System(
    rd1dgrid,
    rdphysics;
    species = [1]
)
```

```
rd1dsol =
1×101 VoronoiFVM.DenseSolutionArray{Float64, 2}:
 1.0  0.944635  0.893732  0.846823  0.803499  …  0.00542387  0.00271192  5.42383e-31
```

```
rd1dsol = solve(rd1dsystem; data=rd1ddata)
```




```
scalarplot(rd1dgrid, rd1dsol[1, :], size = (700, 200))
```

2D scalar stationary reaction-diffusion

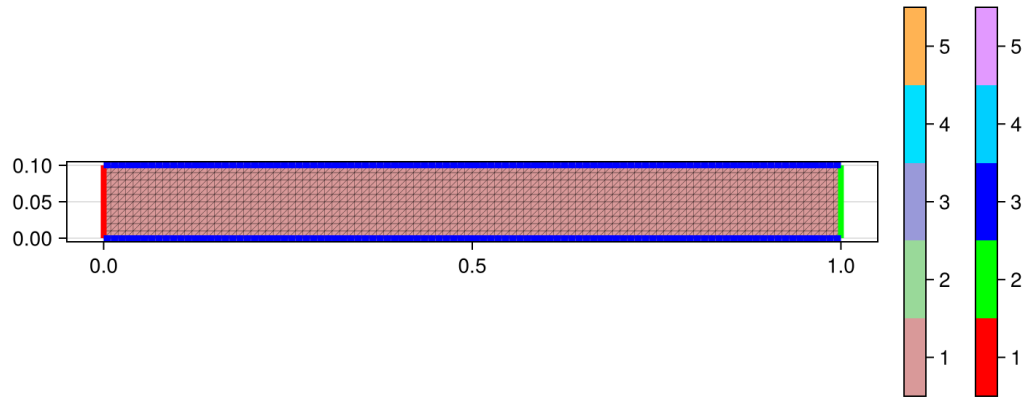
With the same data, one can solve a 2D reaction diffusion system

Let $\Omega = (0,1)$, $\Gamma_1 = \{0\}\times(0,0.1)$, $\Gamma_2 = \{1\}\times(0,0.1)$, $\Gamma_3 = \partial\Omega \setminus (\Gamma_1 \cup \Gamma_2)$ solve

$$\begin{aligned} -\nabla\cdot(D\nabla u) + Ru^2 &= 0 \\ u|_{\Gamma_1} &= 1 \\ u|_{\Gamma_2} &= 0 \\ D\partial_n u|_{\Gamma_3} &= 0 \end{aligned}$$

```
ExtendableGrids.ExtendableGrid{Float64, Int32}
      dim =       2
   nnodes =    1111
   ncells =    2000
  nbfaces =     220
```

```
begin
    rd2dgrid=simplexgrid(0:0.01:1, 0:0.01:0.1)
    bfacemask!(rd2dgrid,[0,0], [1,1], 3, allow_new=false)
    bfacemask!(rd2dgrid,[0,0], [0,1], 1)
    bfacemask!(rd2dgrid,[1,0], [1,1], 2)
end
```



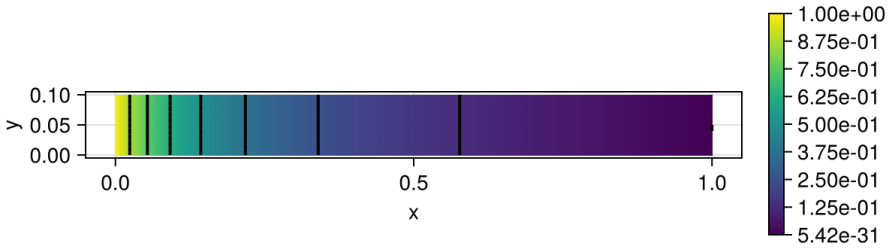

```
gridplot(rd2dgrid, linewidth=0.1,size=(700,300))
```

```
rd2dsystem =
VoronoiFVM.System{Float64, Float64, Int32, Int64, Matrix{Int32}}(
  grid = ExtendableGrids.ExtendableGrid{Float64, Int32}(dim=2, nnodes=1111, ncells=200
  nbfaces=220),
  physics = Physics(flux=rdflux, storage=default_storage, reaction=rdreaction,
  breaction=rdbc, ),
  num_species = 1)
```

```
rd2dsystem = VoronoiFVM.System(
    rd2dgrid, rdphysics;
    species = [1]
)
```

```
rd2dsol =
1×1111 VoronoiFVM.DenseSolutionArray{Float64, 2}:
 1.0  0.944635  0.893732  0.846823  0.803499  …  0.00542387  0.00271192  5.42383e-31
```

```
rd2dsol=solve(rd2dsystem; data=rd1ddata)
```




```
scalarplot(rd2dgrid, rd2dsol[1, :], size = (600, 200))
```

What is going on here ?

We solved a nonlinear system of equations, and just specified some possibly nonlinear constituting functions. The solution process involves:

- Calculation of the data derived from the geometry: $|\omega_k|$, $|\frac{\sigma_{kl}}{h_{kl}}|$, $|\gamma_k|$. This then allows to piece togeteher a nonlniear operator $A : \mathbf{R}^N \to \mathbf{R}^N$ essentially defined on the graph made of the edges and nodes of the triangulation, where $A(u) = 0$ gives the solution of the discrete problem.
- Iterative solution by (damped) Newton`s method: given u_k , iterate until ||d_k||<` tol:

$$\begin{aligned} \text{residual: } r_k &= A(u_k) \\ \text{solve } A'(u_k)d_k &= r_k \\ \text{update: } u_{k+1} &= u_k - d_k \end{aligned}$$

- Using (multi)dual numbers implemented in the Julia package ForwardDiff.jl, values and partial derivatives (local Jacobi matrices) of the constitutive function are jointly calculated.

- The values are assembled into the residual vector r_k, and the local Jacobi matrices are assembled into the global Jacobi matrix $A'(u_k)$. Efficient sparse matrix assembly is supported by the Julia package ExtendableSparse.jl.
- The linear system yielding the Newton update is solved using sparse direct or iterative solvers.

Examples II

Reaction-diffusion system

Regard the following system of reacting species with the reaction

$$u_1 \leftrightharpoons 2u_2$$

in the time interval $[0, t_{end}]$

$$\begin{aligned}
\partial_t u_1 - \nabla \cdot D_1 \nabla u_1 + R(u_1, u_2) &= 0 \\
\partial_t u_2 - \nabla \cdot D_2 \nabla u_2 - R(u_1, u_2) &= 0 \\
R(u1, u2) &= k^+ u_1 - k^- u_2^2 \\
u_1|_{t=0} = u_2|_{t=0} &= 0 \\
u_1|_{\Gamma_1} &= 1 \\
D_2 \partial_n u_2|_{\Gamma_1} &= 0 \\
D_1 \partial_n u_1|_{\Gamma_2} &= 0 \\
u_2|_{\Gamma_2} &= 0
\end{aligned}$$

```
md"""
### Reaction-diffusion system

Regard the following system of reacting species with the reaction
```math
 u_1 \leftrightharpoons 2u_2
```
in the time interval ``[0, t_{end}]``
```math
\begin{aligned}
    \partial_t  u_1  - \nabla \cdot D_1 \nabla u_1 + R(u_1, u_2) & =0\\
    \partial_t  u_2  - \nabla \cdot D_2 \nabla u_2 - R(u_1, u_2) & =0\\
                          R(u1, u2) &= k^+u_1 - k^- u_2^2\\
                        u_1|_{t=0}=u_2|_{t=0} &=0\\
                           u_1|_{Γ_1} & =1\\
                           D_2\partial_n u_2|_{Γ_1} &=0\\
                           D_1\partial_n u_1|_{Γ_2} &=0\\
                           u_2|_{Γ_2} & =0
\end{aligned}
```

"""
```

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

```
function rdsflux(y,u, edge, data)
    (;D₁, D₂)= data
    y[1]= D₁*(u[1,1]- u[1,2])
    y[2]= D₂*(u[2,1]- u[2,2])
end
```

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

```
function rdsreaction(y, u, node, data)
    (;k⁺, k⁻) = data
    R=k⁺*u[1] - k⁻*u[2]^2
    y[1]=R
    y[2]=-R
end
```

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

```
function rdsstorage(y,u, node, data)
    y[1]=u[1]
    y[2]=u[2]
end
```

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

```
function rdsbc(y,u, bnode, data)
    boundary_dirichlet!(y,u,bnode, species=1, value=1, region=1)
    boundary_dirichlet!(y,u,bnode, species=2, value=0, region=2)
end
```

```
rds1ddata = (k⁺ = 1.0, k⁻ = 0.1, D₁ = 1, D₂ = 0.1)
```

```
rds1ddata= (k⁺=1.0, k⁻ =0.1, D₁=1, D₂=0.1)
```

```
rdsphysics =
Physics(data=@NamedTuple{k⁺::Float64, k⁻::Float64, D₁::Int64, D₂::Float64}, flux=rdsflu
```

```
rdsphysics=VoronoiFVM.Physics(storage=rdsstorage, flux=rdsflux,
reaction=rdsreaction, breaction=rdsbc, data=rds1ddata)
```

```
rds1dsys =
VoronoiFVM.System{Float64, Float64, Int32, Int64, Matrix{Int32}}(
  grid = ExtendableGrids.ExtendableGrid{Float64, Int32}(dim=1, nnodes=21, ncells=20,
  nbfaces=2),
  physics = Physics(data=@NamedTuple{k⁺::Float64, k⁻::Float64, D₁::Int64, D₂::Float64}
  flux=rdsflux, storage=rdsstorage, reaction=rdsreaction, breaction=rdsbc, ),
  num_species = 2)
```

```
rds1dsys=VoronoiFVM.System(grid1d, rdsphysics; species=[1,2])
```

```
rds1dinival =
2×21 VoronoiFVM.DenseSolutionArray{Float64, 2}:
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```
rds1dinival=unknowns(rds1dsys, inival=0)
```

```
tend = 100.0
```

```
tend=100.0
```

```
rds1dtsol =
t: 160-element Vector{Float64}:
   0.0
   0.001
   0.001099999999999999
   0.0012199999999999978
   0.0013639999999999963
   0.0015367999999999944
   0.0017441599999999922
   ⋮
  95.62932919203968
  96.62932919203968
  97.47199689402976
  98.31466459601984
  99.15733229800992
 100.0
u: 160-element Vector{Matrix{Float64}}:
 [0.0 0.0 … 0.0 0.0; 0.0 0.0 … 0.0 0.0]
 [1.0 0.00396438341461381 … 2.3192319539438434e-46 1.8388360388058222e-48; 0.000999204
 [1.0 0.004360465973178008 … 2.576924393270934e-46 2.043151154228689e-48; 0.0010991164
 [1.0 0.004835253376884072 … 2.928323174171512e-46 2.3217626752598706e-48; 0.001218999
 [1.0 0.005404262601370727 … 3.4209382875835354e-46 2.7123395739017134e-48; 0.00136284
 [1.0 0.006060162861009065 … 4.135563693887164e-46 3.2789404906935426e-48; 0.00153543
 [1.0 0.006902601462189601 … 5.217455205241221e-46 4.136733562137586e-48; 0.0017425047
 ⋮
 [1.0 0.932038491597038 … 0.06249638663737795 0.055483648207606484; 3.0871823783675634
 [1.0 0.9323303606763975 … 0.06317516128721831 0.0560884938505353; 3.087536428453885 3
 [1.0 0.9325711881893448 … 0.06373662681664896 0.056588809573455155; 3.087828468697075
 [1.0 0.9328069915414926 … 0.0642876529658222 0.05707983551223265; 3.0881143295935556
 [1.0 0.9330378662038462 … 0.0648283304899315 0.05756165207133387; 3.088394135255242 3
 [1.0 0.9332639062377491 … 0.06535876459488141 0.058034351239639424; 3.088668006960300
```

```
rds1dtsol=solve(rds1dsys; inival=rds1dinival, times=(0,tend), Δt=1.0e-4,
force_first_step=true)
```

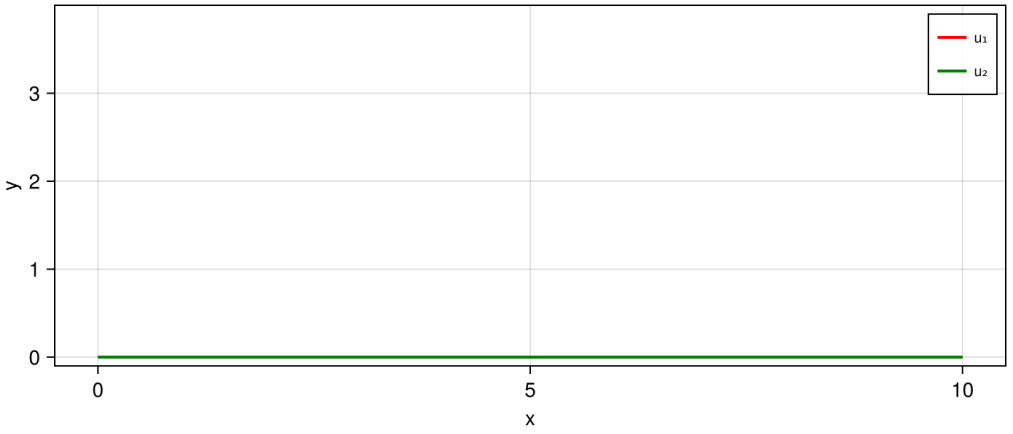
This is a time dependent solution which can be accessed in different ways:

- `rd1dtsol[k,j,k]` is the value of the solution for species `i` in point `j` and timestep `k`.

- `rd1dtsol(t)` interpolates the solution for time `t` and returns an `nspecies × npoints` matrix

0.0

```
@bind rds1dt PlutoUI.Slider(range(0,rds1dtsol.t[end], length=101),
show_value=true)
```




```
let
    u=rds1dtsol(rds1dt)
    vis=GridVisualizer(size=(700,300), limits=(-0.1,4),legend=:rt )
    scalarplot!(vis, grid1d, u[1,:], color=:red, label="u₁")
    scalarplot!(vis, grid1d, u[2,:], color=:green, label="u₂", clear=false)
    reveal(vis)
end
```

Instead of the inbuilt implicit Euler method, it is also possible to uses transient solvers from the Julia SciML DifferentialEquations suite

```
begin
   using OrdinaryDiffEqRosenbrock: ODEProblem, Rosenbrock23
end
```

```
problem = ODEProblem with uType Vector{Float64} and tType Float64. In-place: true
        timespan: (0.0, 100.0)
        u0: 42-element Vector{Float64}:
         0.0
         0.0
         0.0
         0.0
         0.0
         0.0
         0.0
         ⋮
         0.0
         0.0
         0.0
         0.0
         0.0
         0.0
```

```
problem = ODEProblem(rds1dsys,rds1dinival,(0,tend))
```

odesol =

	timestamp	value1	value2	value3	value4	value5	valu
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	3.10874e-32	0.116998	1.83778e-33	7.35111e-33	1.04575e-64	3.80273e-64	4.9807
3	3.73048e-32	0.138688	2.63278e-33	1.05311e-32	1.65568e-64	6.02064e-64	8.3467
4	7.05973e-32	0.246153	9.05913e-33	3.62365e-32	9.99697e-64	3.63526e-63	8.0958
5	8.47244e-32	0.287575	1.28306e-32	5.13223e-32	1.67847e-63	6.10352e-63	1.5673
6	1.22875e-31	0.388494	2.57517e-32	1.03007e-31	4.88937e-63	1.77795e-62	6.4773
7	1.47035e-31	0.444843	3.58238e-32	1.43295e-31	8.15601e-63	2.96582e-62	1.2792
8	1.91764e-31	0.535901	5.77884e-32	2.31154e-31	1.73339e-62	6.30325e-62	3.5462
9	2.27289e-31	0.597426	7.79326e-32	3.1173e-31	2.79269e-62	1.01552e-61	6.7718
10	2.80318e-31	0.674498	1.11694e-31	4.46774e-31	5.00093e-62	1.81852e-61	1.5039

⋮ more

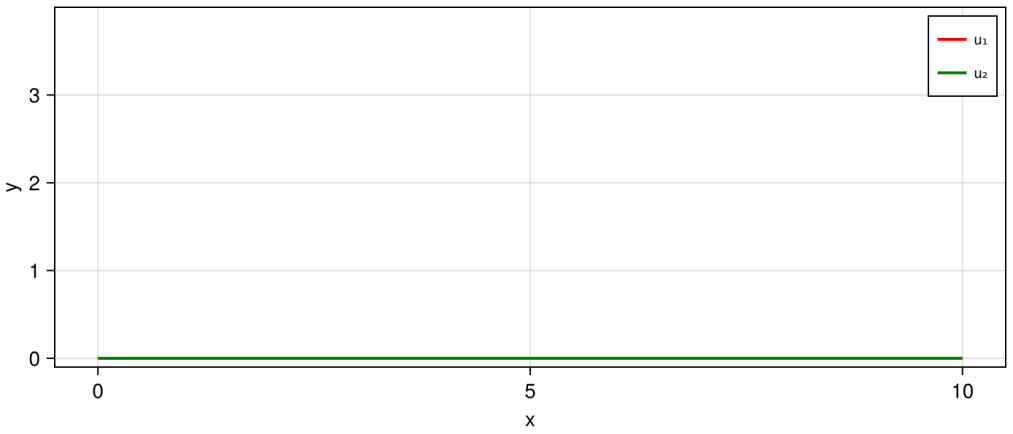
```
odesol = solve(problem, Rosenbrock23(), dt=1.0e-4,reltol=1.0e-4)
```

```
rds1dodesol =
t: 196-element Vector{Float64}:
   0.0
   3.108735017666712e-32
   3.730482021200054e-32
   7.059726066864108e-32
   8.472439797237249e-32
   1.2287505881434168e-31
   1.4703456755457298e-31
   ⋮
  85.43860055432833
  88.3851526445507
  91.56548406058275
  95.01786684501052
  98.79080995800926
 100.0
u: 196-element Vector{VoronoiFVM.DenseSolutionArray{Float64, 2}}:
 [0.0 0.0 … 0.0 0.0; 0.0 0.0 … 0.0 0.0]
 [0.11699828739938116 7.351113307287329e-33 … 0.0 0.0; 1.8377783268218322e-33 1.045751
 [0.138688147836371 1.0531133011631163e-32 … 0.0 0.0; 2.632783252907791e-33 1.65567566
 [0.24615251751743403 3.623652515713031e-32 … 0.0 0.0; 9.059131289282577e-33 9.9969665
 [0.2875753274351265 5.132226445436349e-32 … 0.0 0.0; 1.2830566113590873e-32 1.6784692
 [0.3884935216444162 1.0300671361295054e-31 … 0.0 0.0; 2.5751678403237636e-32 4.889369
 [0.44484316019238135 1.432951100259106e-31 … 0.0 0.0; 3.582377750647765e-32 8.1560056
 ⋮
 [0.9999999999999998 0.9289452543787162 … 0.055380750794150564 0.04914348866618975; 3.
 [0.9999999999999998 0.9300135207529505 … 0.05780338791102084 0.0513017790829304; 3.08
 [0.9999999999999998 0.9310824771973141 … 0.0602577260971862 0.05348854343298262; 3.08
 [0.9999999999999998 0.9321509933501144 … 0.06273796153598622 0.05569859757513246; 3.0
 [0.9999999999999998 0.933217609978539 … 0.06523763287278744 0.057926170345922824; 3.0
 [0.9999999999999998 0.9335384709868622 … 0.06599554716271573 0.05860151771096899; 3.0
```

```
rds1dodesol=reshape(odesol,rds1dsys)
```

0.0

```
@bind rds1dodet PlutoUI.Slider(range(0,rds1dodesol.t[end], length=1001),
show_value=true)
```




```
let
    u=rds1dodesol(rds1dodet)
    vis=GridVisualizer(size=(700,300), limits=(-0.1,4),legend=:rt )
    scalarplot!(vis, grid1d, u[1,:], color=:red, label="u₁")
    scalarplot!(vis, grid1d, u[2,:], color=:green, label="u₂", clear=false)
    reveal(vis)
end

```