VoronoiFVM.jl: Tipps and Examples

Grid generation

VoronoiFVM works on simplicial grids provided by the package **ExtendableGrids.jl**

There are several ways to create a grid.

1D grids

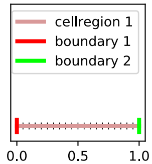
1D grids are created from a vector of monotonicaly increasing x-axis positions.

```
X =
  Float64[0.0,  0.05,  0.1,  0.15,  0.2,  0.25,  0.3,  0.35,  0.4,  0.45,  0.5,  0.55,  0.6,  0
```

```
# Create a 1D vector:
X=collect(range(0,1,length=21))
```

```
grid1d_a = ExtendableGrids.ExtendableGrid{Float64, Int32};
           dim: 1 nodes: 21 cells: 20 bfaces: 2
```

```
# Create grid from vector:
grid1d_a=ExtendableGrids.simplexgrid(X)
```



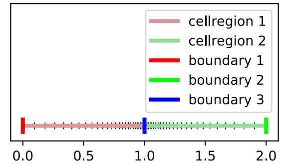

```
# Visualize grid
GridVisualize.gridplot(grid1d_a,resolution=(600,200),Plotter=PyPlot)
```

As we see, the grid is chacracterized by interior points, and boundary points. Each grid cell is endowed with a region number (for allowing different physics, parameters etc. for different regions). Each boundary node has a boundary region number, which is meant to be used to distinguish different boundary conditions.

More sophisticated grids can be created, as we see in the following example:

```
grid1d_b = ExtendableGrids.ExtendableGrid{Float64, Int32};
           dim: 1 nodes: 53 cells: 52 bfaces: 3
```

```
grid1d_b=let
    hmax=0.1
    hmin=0.01
    # Create vectors with geometric distributions of interval sizes
    X1=ExtendableGrids.geomspace(0.0,1.0,hmax,hmin)
    X2=geomspace(1.0,2.0,hmin,hmax)
    # Glue them together at common point x=1 (this is different from vcat!)
    X3=glue(X1,X2)
    grid1d_b=simplexgrid(X3)
    # Mark an additional interior boundary point at x=1
    ExtendableGrids.bfacemask!(grid1d_b,[1.0],[1.0],3)
    # Change cell region number at the right part
    ExtendableGrids.cellmask!(grid1d_b,[1.0],[2.0],2)
    grid1d_b
end
```



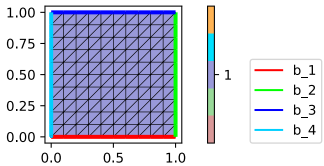

```
gridplot(grid1d_b,resolution=(600,200),Plotter=PyPlot,aspect=0.5)
```

2D Tensor product grids

These are created from two vectors of x and y coordinates, respectively. This results in the creation of a grid of quadrilaterals. Then, each of them is subdivided into two triangles, resulting in a boundary conforming Delaunay grid.

```
grid2d_a = ExtendableGrids.ExtendableGrid{Float64, Int32};
           dim: 2 nodes: 121 cells: 200 bfaces: 40
```

```
grid2d_a=let
    X=collect(range(0,1,length=11))
    Y=collect(range(0,1,length=11))
    simplexgrid(X,Y)
end
```

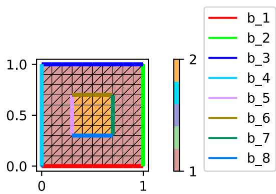



```
gridplot(grid2d_a,resolution=(600,200),Plotter=PyPlot,legend_location=(1.5,0))
```

Once again, we see a default distrbution of cell regions and boundary regions. This can be modified in a similar manner as in the 1D case.

```
grid2d_b = ExtendableGrids.ExtendableGrid{Float64, Int32};
           dim: 2 nodes: 121 cells: 200 bfaces: 56, edges: 320
```

```
grid2d_b=let
    X=collect(range(0,1,length=11))
    Y=collect(range(0,1,length=11))
    grid=simplexgrid(X,Y)
    cellmask!(grid,[0.3,0.3],[0.7,0.7],2)
    bfacemask!(grid,[0.3,0.3],[0.3,0.7],5)
    bfacemask!(grid,[0.3,0.7],[0.7,0.7],6)
    bfacemask!(grid,[0.7,0.3],[0.7,0.7],7)
    bfacemask!(grid,[0.3,0.3],[0.7,0.3],8)
    grid
end
```




```
gridplot(grid2d_b,resolution=(600,200),Plotter=PyPlot,legend_location=(1.5,0))
```

2D Unstructured grids

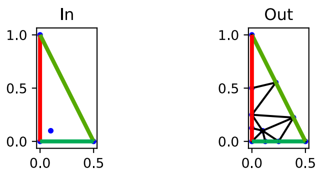
These can be created using the mesh generator Triangle (by J. Shewchuk) via the packages **Triangulate.jl** and **SimplexGridFactory.jl**.

```
builder2d (generic function with 1 method)
```

```
function builder2d()
    b=SimplexGridFactory.SimplexGridBuilder(Generator=Triangulate)
    p1=point!(b,0,0)
    p2=point!(b,0,1)
    p3=point!(b,0.5,0)
    facetregion!(b,1)
    facet!(b,p1,p2)
    facetregion!(b,2)
    facet!(b,p2,p3)
    facetregion!(b,3)
    facet!(b,p1,p3)
    point!(b,0.1,0.1)
    b
end
```

```
builder =
  SimplexGridBuilder(Triangulate,  3,  1,  1.0,  1.0e-12,  Int32[1,  2,  3],  Vector{Int32}[I
```

For debugging purposes, the current state of the builder and its possible output can be visualized:

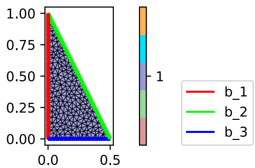



```
builderplot(builder,Plotter=PyPlot)
```

Finally, we can create a grid from the builder:

```
grid2d_c = ExtendableGrids.ExtendableGrid{Float64, Int32};
           dim: 2 nodes: 230 cells: 381 bfaces: 77
```

```
grid2d_c=simplexgrid(builder,maxvolume=0.001)
```




```
gridplot(grid2d_c,resolution=(600,200),Plotter=PyPlot,legend_location=(2,0))
```

Stationary scalar problems

Diffusion with Dirichlet boundary conditions

This is mathematically similar to heat conduction and other problems.

$$
\begin{aligned}
-\nabla \cdot D\nabla u &= 10 \\
u_{\Gamma_{east}} &= 0 \\
u_{\Gamma_{west}} &= 1 \\
D\nabla u \cdot \vec{n}|_{\partial\Omega \setminus (\Gamma_{east} \cup \Gamma_{west})} &= 0
\end{aligned}
$$

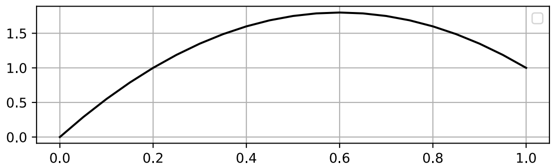
Besides of the domain and its boundary it is characterize by a flux term and a source term.

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

```
function solve_diffproblem_dirichlet(grid;D=1.0)
    species1=1

    # Use finite difference flux between disretization points.
    # Division by distance and multiplication by interface size
    # is done by the VoronoiFVM Module.
    function flux(f,u0,edge)
        u=unknowns(edge,u0)
        f[species1]=D*(u[species1,1]-u[species1,2])
    end

    # Specify a constant source term
    function source(f,node)
        f[species1]=10
    end

    # Combine flux and source to "physics"
    physics=VoronoiFVM.Physics(flux=flux,source=source)

    # Create system from physics and grid
    system=VoronoiFVM.System(grid,physics)

    # Enable species in cellregion 1
    enable_species!(system,species1,[1])


    # Enable boundary conditions. For those boundary regions
    # which are not specified here, by default, homogeneous
    # Neumann boundary conditions are assumed.
    west=dim_space(grid)==1  ? 1 : 4
    east=2
    boundary_dirichlet!(system, species1, west, 0)
    boundary_dirichlet!(system, species1, east, 1)

    # Solve with given initial value
    solve(unknowns(system,inival=0),system)
end
```

```
solution1d_a =
1×21 Matrix{Float64}:
 6.0e-30  0.2875  0.55  0.7875  1.0  …  1.6875  1.6  1.4875  1.35  1.1875  1.0
```

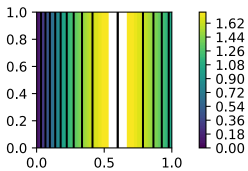
```
solution1d_a=solve_diffproblem_dirichlet(grid1d_a)
```




```
scalarplot(grid1d_a,solution1d_a[1,:],Plotter=PyPlot)
```

```
solution2d_a =
1×121 Matrix{Float64}:
 3.0e-31  0.55  1.0  1.35  1.6  1.75  1.8  …  1.6  1.75  1.8  1.75  1.6  1.35  1.0
```

```
solution2d_a=solve_diffproblem_dirichlet(grid2d_a)
```




```
scalarplot(grid2d_a,solution2d_a[1,:],Plotter=PyPlot)
```

Diffusion with Robin boundary conditions

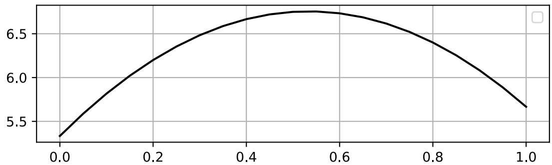
$$-\nabla \cdot D\nabla u = 10$$
$$D\nabla u \cdot \vec{n} + au = 0 \text{ on } \Gamma_{east}$$
$$D\nabla u \cdot \vec{n} + au = a \text{ on } \Gamma_{east}$$
$$D\nabla u \cdot \vec{n}|_{\partial\Omega \setminus (\Gamma_{east} \cup \Gamma_{west})} = 0$$

solve_diffproblem_robin (generic function with 1 method)

```
function solve_diffproblem_robin(grid;D=1.0,a=0.5)
    species1=1

    function flux(f,u0,edge)
        u=unknowns(edge,u0)
        f[species1]=D*(u[species1,1]-u[species1,2])
    end

    function source(f,node)
        f[species1]=10
    end

    physics=VoronoiFVM.Physics(flux=flux,source=source)

    system=VoronoiFVM.System(grid,physics)

    enable_species!(system,species1,[1])

    west=dim_space(grid)==1  ? 1 : 4
    east=2
    boundary_robin!(system, species1, west, a, 0)
    boundary_robin!(system, species1, east, a, a*1)

    solve(unknowns(system,inival=0),system)
end
```

```
solution1d_robin =
1×21 Matrix{Float64}:
 5.33333  5.5875  5.81667  6.02083  6.2  …  6.4  6.25417  6.08333  5.8875  5.66667
```

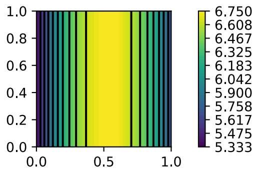
```
solution1d_robin=solve_diffproblem_robin(grid1d_a,a=1)
```




```
scalarplot(grid1d_a,solution1d_robin[1,:],Plotter=PyPlot)
```

```
solution2d_robin =
1×121 Matrix{Float64}:
 5.33333  5.81667  6.2  6.48333  6.66667  …  6.73333  6.61667  6.4  6.08333  5.66667
```

```
solution2d_robin=solve_diffproblem_robin(grid2d_a,a=1)
```




```
scalarplot(grid2d_a,solution2d_robin[1,:],Plotter=PyPlot)
```

Stationary Reaction-Diffusion problem

Here, we regard two species u_1, u_2, and a reaction converting u_1 into u_2. Dirichlet boundary conditions "inject" u_1 an "remove" u_2.

$$
\begin{aligned}
-\nabla \cdot D_1 \nabla u_1 + r(u_1) &= 0 \\
-\nabla \cdot D_2 \nabla u_2 - r(u_1) &= 0 \\
r(u_1) &= k u_1 \\
u_1|_{\Gamma_{west}} &= 1 \\
u_2|_{\Gamma_{east}} &= 0
\end{aligned}
$$

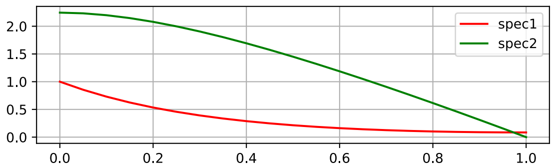
Boundary conditons not specified are assumed to be homogeneous Neumann.

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

```
function solve_readiff(grid;D_1=1.0,D_2=1.0,k=1)
    species1=1
    species2=2

    function flux(f,u0,edge)
        u=unknowns(edge,u0)
        f[species1]=D_1*(u[species1,1]-u[species1,2])
        f[species2]=D_2*(u[species2,1]-u[species2,2])
    end

    function reaction(f,u0,node)
        u=unknowns(node,u0)
        r=k*u[species1]
        f[species1]=r
        f[species2]=-r
    end

    physics=VoronoiFVM.Physics(num_species=2,flux=flux,reaction=reaction)

    system=VoronoiFVM.System(grid,physics)

    enable_species!(system,species1,[1])
    enable_species!(system,species2,[1])

    west=dim_space(grid)==1  ? 1 : 4
    east=2
    boundary_dirichlet!(system, species1, west,1)
    boundary_dirichlet!(system, species2, east,0)

    solve(unknowns(system,inival=0),system)
end
```

```
solution_readiff_1d =
2×21 Matrix{Float64}:
 1.0      0.854464  0.730289  0.624372  …  0.0890498  0.0858439  0.0847841
 2.24551  2.23301   2.19915   2.14703      0.311807   0.156976   3.16072e-30
```

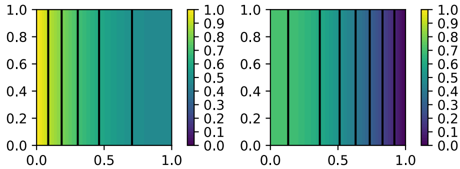
```
solution_readiff_1d=solve_readiff(grid1d_a,k=10, D_2=1)
```




```
let
    v=GridVisualizer(Plotter=PyPlot)
    scalarplot!(v[1,1],grid1d_a, solution_readiff_1d[1,:],label="spec1", color=:red)
    scalarplot!(v[1,1],grid1d_a, solution_readiff_1d[2,:],label="spec2",
color=:green,show=true,clear=false)
end
```

```
solution_readiff_2d =
2×121 Matrix{Float64}:
 1.0      0.884085  0.785851  0.703335  0.634885  …  0.478053  0.464174  0.459578
 0.71873  0.70873   0.681048  0.63765   0.580184     0.233355  0.121319  6.29576e-32
```

```
solution_readiff_2d=solve_readiff(grid2d_a,k=2)
```




```julia
let
    v=GridVisualizer(Plotter=PyPlot,layout=(1,2))
    scalarplot!(v[1,1],grid2d_a, solution_readiff_2d[1,:],label="spec1",flimits=
(0,1))
    scalarplot!(v[1,2],grid2d_a, solution_readiff_2d[2,:],label="spec2",flimits=
(0,1), show=true)
end
```

Transient Reaction-Diffusion problem

Here, we regard two species u_1, u_2, and a reaction converting u_1 into u_2. Dirichlet boundary conditions "inject" u_1 an "remove" u_2.

$$\begin{aligned}\partial_t u_1 - \nabla\cdot D_1 \nabla u_1 + r(u_1) &= 0\\ \partial_t u_2 - \nabla\cdot D_2 \nabla u_2 - r(u_1) &= 0\\ r(u_1) &= k u_1\\ u_1|_{\Gamma_{west}} &= 1\\ u_2|_{\Gamma_{east}} &= 0\\ u_1|_{t=0} &= 0\\ u_2|_{t=0} &= 0\end{aligned}$$

Boundary conditons not specified are assumed to be homogeneous Neumann.

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

```julia
# Function describing evolution of system with initial value inival
# using the Implicit Euler method
function evolution(inival, # initial value
                   sys,     # finite volume system
                   grid,    # simplex grid
                   tstep,   # initial time step
                   tend,    # end time
                   dtgrowth  # time step growth factor
                   )
    time=0.0
    # record time and solution
    times=[time]
    solutions=[copy(inival)]

    solution=copy(inival)
    while time<tend
        time=time+tstep
        solve!(solution,inival,sys,tstep=tstep) # solve implicit Euler time step
        inival.=solution  # copy solution to inivalue
        push!(times,time)
        push!(solutions,copy(solution))
        tstep*=dtgrowth  # increase timestep by factor when approaching stationary
state
    end
    # return result and grid
    (times=times,solutions=solutions,grid=grid)
end

```

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

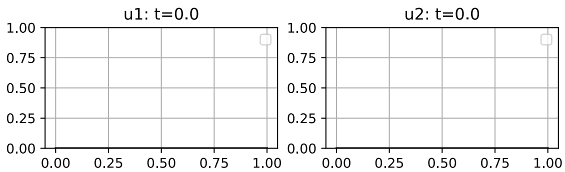
```julia
function transient_reaction_diffusion(grid;D_1=1.0,D_2=1.0,k=1,
    tstep=1.0e-3,tend=1,dtgrowth=1.1)
    species1=1
    species2=2

    function flux(f,u0,edge)
        u=unknowns(edge,u0)
        f[species1]=D_1*(u[species1,1]-u[species1,2])
        f[species2]=D_2*(u[species2,1]-u[species2,2])
    end

    function reaction(f,u0,node)
        u=unknowns(node,u0)
        r=k*u[species1]
        f[species1]=r
        f[species2]=-r
```

```
    end

    function storage(f,u,node)
        f.=u
    end


physics=VoronoiFVM.Physics(num_species=2,flux=flux,reaction=reaction,storage=storage)

    system=VoronoiFVM.System(grid,physics)

    enable_species!(system,species1,[1])
    enable_species!(system,species2,[1])

    west=dim_space(grid)==1  ? 1 : 4
    east=2
    boundary_dirichlet!(system, species1, west,1)
    boundary_dirichlet!(system, species2, east,0)
    ## Create a solution array
    inival=unknowns(system,inival=0)
    evolution(inival,system,grid,tstep,tend,dtgrowth)
end
```

```
tsol_readiff =
 (times = Float64[0.0, 0.001, 0.0021, 0.00331, 0.004641, 0.0061051, 0.00771561, 0.009
```

```
tsol_readiff=transient_reaction_diffusion(grid1d_a,k=1,tend=100)
```

time=




```
let
    vis=GridVisualizer(layout=(1,2),resolution=(600,300),Plotter=PyPlot)
    scalarplot!(vis[1,1],tsol_readiff.grid,
           tsol_readiff.solutions[t_readiff][1,:],
           title="u1: t=$(tsol_readiff.times[t_readiff])",
           flimits=(0,1),colormap=:cool,levels=50,clear=true)
    scalarplot!(vis[1,2],tsol_readiff.grid,
           tsol_readiff.solutions[t_readiff][2,:],
           title="u2: t=$(tsol_readiff.times[t_readiff])",
           flimits=(0,1),colormap=:cool,levels=50,show=true)
end

```

Table of Contents


```
TableOfContents()
```

```
begin
    ENV["LANG"]="C"
    using Pkg
    Pkg.activate(mktempdir())

Pkg.add(["PyPlot","PlutoUI","ExtendableGrids","SimplexGridFactory","VoronoiFVM","Grid
Visualize","Triangulate"])
```

```
    using
PlutoUI,PyPlot,ExtendableGrids,SimplexGridFactory,VoronoiFVM,GridVisualize,Triangulat
e
    PyPlot.svg(true)
end;
```

```
    Status `/tmp/jl_SKqLWP/Project.toml`
  [cfc395e8] ExtendableGrids v0.7.4
  [5eed8a63] GridVisualize v0.1.3
  [7f904dfe] PlutoUI v0.7.2
  [d330b81b] PyPlot v2.9.0
  [57bfcd06] SimplexGridFactory v0.5.1
  [f7e6ffb2] Triangulate v1.0.1
  [82b139dc] VoronoiFVM v0.10.5
```