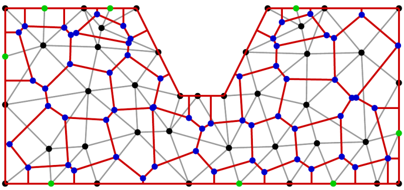
```
begin
    ENV["LANG"]="C"
    using Pkg
    Pkg.activate(mktempdir())
    using Revise
    Pkg.add("Revise")
    Pkg.add(["PyPlot","PlutoUI","ExtendableGrids","GridVisualize", "VoronoiFVM"])
    using PlutoUI,PyPlot,ExtendableGrids,VoronoiFVM,GridVisualize
    PyPlot.svg(true)
end;
```

# Finite volumes: transient problems

## Construction of control volumes

- Start with a triangulation of a polygonal domain (intervals in 1D,triangles in 2D, tetrahedra in 3D).
- Join triangle circumcenters by lines $\to$ create Voronoi cells which can serve as control volumes, akin to representative elementary volumes (REV) used to derive conservation laws.



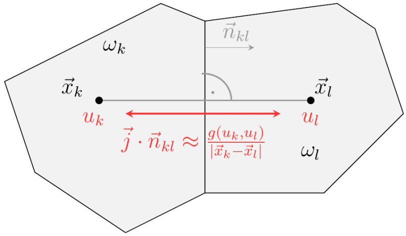
- Black + green: triangle nodes
- Gray: triangle edges
- Blue: triangle circumcenters
- Red: Boundaries of Voronoi cells

## Condition on triangulation

- There is a 1:1 incidence between triangulation nodes and Voronoi cells. Moreover, the angle between the interface between two Voronoi cells and the edge between their corresponding nodes is $\frac{\pi}{2}$.
- Requires (in 2D) that sums of angles opposite to triangle edges are less than $\pi$ and that angles opposite to boudary edges are less than $\frac{\pi}{2}$.
- "boundary conforming Delaunay property". It has different equivalent definitions and analogues in 3D.
- Construction:
  - "by hand" (or script) from tensor product meshes
  - Mesh generators: Triangle, TetGen
  - Julia packages: Triangulate.jl, TetGen.jl; SimplexGridFactory.jl

## The discretization approach

- Use Voronoi cells as REVs aka control volumes aka finite volume cells.




- Given a continuity equation $\nabla \cdot \vec{j} = 0$ in a domain $\Omega$, integrate this over a contol volume $\omega_k$ with associated node $\vec{x}_k$ and apply Gauss theorem:

$$\begin{aligned} 0 &= \int_{\omega_k} \nabla \cdot \vec{j}\, d\omega = \int_{\partial\omega_k} \vec{j} \cdot \vec{n} ds \\ &= \sum_{l \in N_k} \int_{\omega_k \cap \omega_l} \vec{j} \cdot \vec{n} ds + \int_{\partial\omega_k \cap \partial\Omega} \vec{j} \cdot \vec{n} ds \\ &\approx \sum_{l \in N_k} \frac{\sigma_{kl}}{h_{kl}} g(u_k, u_l) + \gamma_k b(u_k) \end{aligned}$$

- Here, $N_k$ is the set of neighbor control volumes, $\sigma_{kl} = |\omega_k \cap \omega_l|$, $h_{kl} = |\vec{x}_k - \vec{x}_l|$, $\gamma_k = |\partial\omega_k \cap \partial\Omega|$, where $|\cdot|$ denotes the measure (length resp. area) of a geometrical entity.

## Flux functions

For instance, for the diffusion flux $\vec{j} = -D\vec{\nabla}u$, we use $g(u_k, u_l) = D(u_k - u_l)$.

For a convective diffusion flux $\vec{j} = -D\vec{\nabla}u + u\vec{v}$, one can chose the upwind flux

$$g(u_k, u_l) = D(u_k - u_l) + v_{kl} \begin{cases} u_k, & v_{kl} > 0 \\ u_l, & v_{kl} \leq 0, \end{cases}$$

where $v_{kl} = \frac{h_{kl}}{\sigma_{kl}} \int_{\omega_k \cap \omega_l} \vec{v} \cdot \vec{n}_{kl}\, ds$ Fluxes also can depend nonlinearly on $u$.

# Software API and implementation

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

The entities describing the discrete system can be subdivided into two categories:

- geometrical data: $|\omega_k|, \gamma_k, \sigma_{kl}, h_{kl}$ together with the connectivity information of the triangles
- physical data: the number $m$ and the functions $s, g, r, f$ describing the particular problem, where $g$ is a flux function approximating $\vec{j}$.

This structure allows to describe the problem to be solved by data derived from the discretization grid and by the functions describing the physics, giving rise to a software API.

The solution of the nonlinear systems of equations can be performed by Newton's method combined with various direct and iterative linear solvers.

The generic programming capabilities of Julia allow for an implementation of the method which results in an API which consists in the implementation of functions $s, g, r, f$ without the need to write code for their derivatives.

# Examples

## General settings

Initial value problem with homgeneous Neumann boundary conditions

$$\Omega = (0,1)^d,\ d = 1,2$$

$$T = [0, t_{end}]$$

fpeak (generic function with 2 methods)

```
# Define function for initial value $u_0$ with two methods - for 1D and 2D problems
begin
    fpeak(x)=exp(-100*(x-0.25)^2)
    fpeak(x,y)=exp(-100*((x-0.25)^2+(y-0.25)^2))
end
```

create_grid (generic function with 1 method)

```
# Create discretization grid in 1D or 2D with approximately n nodes
function create_grid(n,dim)
    nx=n
    if dim==2
        nx=ceil(sqrt(n))
    end
    X=collect(0:1.0/nx:1)
    if dim==1
      grid=simplexgrid(X)
    else
      grid=simplexgrid(X,X)
    end
end
```

# Diffusion problem

$$\partial_t u - \nabla \cdot D\nabla u = 0 \text{ in } \Omega$$

$$D\nabla u \cdot \vec{n} = 0 \text{ on } \partial\Omega$$

$$u|_{t=0} = u_0$$

diffusion (generic function with 1 method)

```
function diffusion(;n=100,dim=1,tstep=1.0e-4,tend=1, D=1.0, dtgrowth=1.1)
    grid=create_grid(n,dim)

    ## Diffusion flux between neigboring control volumes
    function flux!(f,u,edge)
        uk=viewK(edge,u)
        ul=viewL(edge,u)
        f[1]=D*(uk[1]-ul[1])
    end

    ## Storage term (under time derivative)
    function storage!(f,u,node)
        f[1]=u[1]
    end

    ## Create a physics structure
    physics=VoronoiFVM.Physics(
        flux=flux!,
        storage=storage!)

    sys=VoronoiFVM.DenseSystem(grid,physics)
    enable_species!(sys,1,[1])

    ## Create a solution array
    inival=unknowns(sys)

    ## Broadcast the initial value
    inival[1,:].=map(fpeak,grid)

    control=VoronoiFVM.NewtonControl()
    control.Δt_min=0.01*tstep
    control.Δt=tstep
    control.Δt_max=0.1*tend
    control.Δu_opt=0.1
    control.Δt_grow=dtgrowth

    tsol=solve(inival,sys,[0,tend];control=control)
    return grid,tsol
end
```

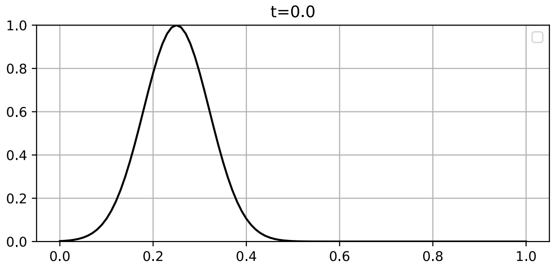
```
grid_diffusion,tsol_diffusion=diffusion(dim=1,n=100);
```

TransientSolution{Float64, 3, Vector{Matrix{Float64}}, Vector{Float64}}

```
typeof(tsol_diffusion)
```

- Documentation for `NewtonControl`
- Documentation for **transient** `solve`
- Documentation for `TransientSolution`

Time step number: 1




# Reaction-diffusion problem

Diffusion + physical process which "eats" species

$$\partial_t u - \nabla \cdot D\nabla u + Ru = 0 \text{ in } \Omega$$

$$D\nabla u \cdot \vec{n} = 0 \text{ on } \partial\Omega$$

$$u|_{t=0} = u_0$$

reaction_diffusion (generic function with 1 method)

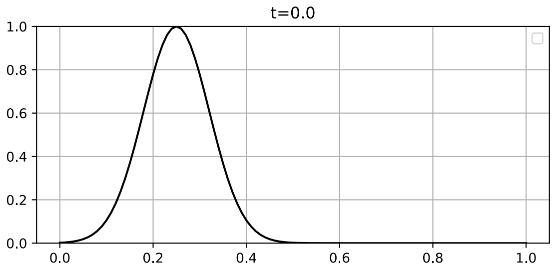
```
function reaction_diffusion(;
        n=1000,
        dim=1,
        tstep=1.0e-4,
        tend=1,
        D=1.0,
        R=10.0,
        dtgrowth=1.1)

    grid=create_grid(n,dim)
    ## Diffusion flux between neigboring control volumes
    function flux!(f,u,edge)
        uk=viewK(edge,u)
        ul=viewL(edge,u)
        f[1]=D*(uk[1]-ul[1])
    end

    ## Storage term (under time derivative)
    function storage!(f,u,node)
        f[1]=u[1]
    end

    ## Reaction term
    function reaction!(f,u,node)
        f[1]=R*u[1]
    end


    ## Create a physics structure
    physics=VoronoiFVM.Physics(
        flux=flux!,
        reaction=reaction!,
        storage=storage!)

    sys=VoronoiFVM.DenseSystem(grid,physics)
    enable_species!(sys,1,[1])
    ## Create a solution array
    inival=unknowns(sys)

    ## Broadcast the initial value
    inival[1,:].=map(fpeak,grid)



    control=VoronoiFVM.NewtonControl()
```

```
    control.Δt_min=0.01*tstep
    control.Δt=tstep
    control.Δt_max=0.1*tend
    control.Δu_opt=0.1
    control.Δt_grow=dtgrowth

    tsol=solve(inival,sys,[0,tend];control=control)
    return grid,tsol
end
```

```
grid_rd,tsol_rd=reaction_diffusion(dim=1,n=100,R=10);
```

Time step number: 1

```
md"""
Time step number:$(@bind t_reaction_diffusion
Slider(1:length(tsol_rd),default=1,show_value=true))
"""
```




# Convection-Diffusion problem

$$\partial_t u - \nabla \cdot (D \nabla u - u\vec{v}) = 0 \text{ in } \Omega$$

$$(D \nabla u - u\vec{v}) \cdot \vec{n} = 0 \text{ on } \partial\Omega$$

$$u|_{t=0} = u_0$$

convection_diffusion (generic function with 1 method)

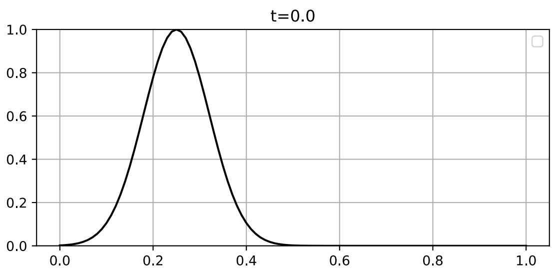
```
function convection_diffusion(;
        n=20,
        dim=1,
        tstep=1.0e-4,
        tend=1,
        D=0.01,
        vx=10.0,
        vy=10.0,
        dtgrowth=1.1,
        scheme="expfit")
    grid=create_grid(n,dim)
    # copy vx, vy into vector
    if dim==1
        V=[vx]
    else
        V=[vx,vy]
    end
    # Bernoulli function


     B(x)=x/(exp(x)-1)

     function flux_expfit!(f,u,edge)
         uk=viewK(edge,u)
         ul=viewL(edge,u)
         vh=project(edge,V)  # Calculate projection v * (x_L-x_K)
         f[1]=D*(B(-vh/D)*uk[1]- B(vh/D)*ul[1])
     end

     function flux_centered!(f,u,edge)
         uk=viewK(edge,u)
         ul=viewL(edge,u)
         vh=project(edge,V)
         f[1]=D*(uk[1]-ul[1])+ vh*0.5*(uk[1]+ul[1])
     end
```

```
    function flux_upwind!(f,u,edge)
        uk=viewK(edge,u)
        ul=viewL(edge,u)
        vh=project(edge,V)
        f[1]=D*(uk[1]-ul[1])+ ( vh>0.0 ? vh*uk[1] : vh*ul[1] )
    end

    flux! =flux_upwind!
    if scheme=="expfit"
        flux! =flux_expfit!
    elseif scheme=="centered"
        flux! =flux_centered!
    end


    ## Storage term (under time derivative)
    function storage!(f,u,node)
        f[1]=u[1]
    end

    ## Create a physics structure
    physics=VoronoiFVM.Physics(
        flux=flux!,
        storage=storage!)

    sys=VoronoiFVM.DenseSystem(grid,physics)
    enable_species!(sys,1,[1])
    ## Assume homogeneous Neumann boundary conditions, so do nothig

    ## Create a solution array
    inival=unknowns(sys)

    ## Broadcast the initial value
    inival[1,:].=map(fpeak,grid)


    control=VoronoiFVM.NewtonControl()
    control.Δt_min=0.01*tstep
    control.Δt=tstep
    control.Δt_max=0.1*tend
    control.Δu_opt=0.1
    control.Δt_grow=dtgrowth

    tsol=solve(inival,sys,[0,tend];control=control)
    return grid,tsol
end
```

scheme: centered / expfit / upwind

```
grid_conv,tsol_conv=convection_diffusion(n=100,
    dim=1,
    scheme=scheme[1],
    vx=10,vy=10);
```

Time step number: 1




## Brusselator system

Two species interacting via a reaction:

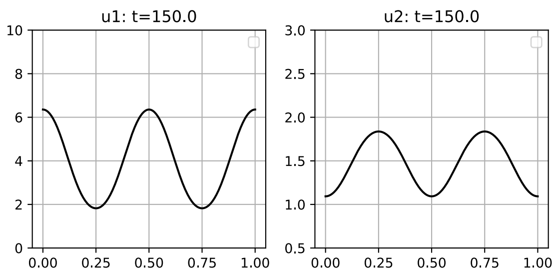
$$\partial_t u_1 - \nabla \cdot (D_1 \nabla u_1) + (B+1)u_1 - A - u_1^2 u_2 = 0$$
$$\partial_t u_2 - \nabla \cdot (D_2 \nabla u_2) + u_1^2 u_2 - B u_1 = 0$$

brusselator (generic function with 1 method)

```
function brusselator(;n=100,dim=1,A=4.0,B=6.0,D1=0.01,D2=0.1,perturbation=0.1,
    tstep=0.01, tend=150,dtgrowth=1.05)

    grid=create_grid(n,dim)
    function storage!(f,u,node)
        f.=u
    end


function bruss_diffusion!(f,_u,edge)
        u=unknowns(edge,_u)
    f[1]=D1*(u[1,1]-u[1,2])
    f[2]=D2*(u[2,1]-u[2,2])
end
# Reaction:
function bruss_reaction!(f,u,node)
    f[1]= (B+1.0)*u[1]-A-u[1]^2*u[2]
    f[2]= u[1]^2*u[2]-B*u[1]
end
# Create system
bruss_physics=VoronoiFVM.Physics(flux=bruss_diffusion!,storage=storage!,
                                 num_species=2,reaction=bruss_reaction!)
brusselator_system=VoronoiFVM.DenseSystem(grid,bruss_physics)
enable_species!(brusselator_system,1,[1])
enable_species!(brusselator_system,2,[1])

    inival=unknowns(brusselator_system)
for i=1:num_nodes(grid)
    inival[1,i]=1.0+perturbation*randn()
    inival[2,i]=1.0+perturbation*randn()
end


    control=VoronoiFVM.NewtonControl()
    control.Δt_min=0.0001*tstep
    control.Δt=tstep
    control.Δt_max=0.1*tend
    control.Δu_opt=0.1
    control.Δt_grow=dtgrowth

    tsol=solve(inival,brusselator_system,[0,tend];control=control)
    return grid,tsol
end
```

```
grid_bruss,tsol_bruss=brusselator(n=500,dim=1);
```

Time step number: 365




## Table of Contents

```
    Status `/tmp/jl_1ZzYoO/Project.toml`
  [cfc395e8] ExtendableGrids v0.7.4
  [5eed8a63] GridVisualize v0.1.5
  [7f904dfe] PlutoUI v0.7.4
  [d330b81b] PyPlot v2.9.0
  [295af30f] Revise v3.1.14
  [82b139dc] VoronoiFVM v0.10.9
```