Finite Volumes for systems of partial differential equations

Advanced Topics from Scientific Computing
TU Berlin Winter 2024/25
Notebook 15

Jürgen Fuhrmann

CairoMakie

```
begin
    using LinearAlgebra, Printf, LaTeXStrings
    using ExtendableGrids, VoronoiFVM
    using ExtendableSparse, LinearSolve
    using GridVisualize
    using CairoMakie
    CairoMakie.activate!(type = "png")
    default_plotter!(CairoMakie)
end
```

A system of reaction-diffusion equations

Assume, we are given n coupled PDEs in $\Omega \subset \mathbb{R}^d$:

Denote n-vectors by bold face and d-vectors by arrows. Let $\mathbf{u}(\vec{x},t) = (u_1(\vec{x},t) \dots u_n(\vec{x},t))$ be a n-vector function.

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

In vector form, this can be rewritten as:

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

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

Similar for nonlinear Robin boundary conditions on $\partial\Omega$:

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

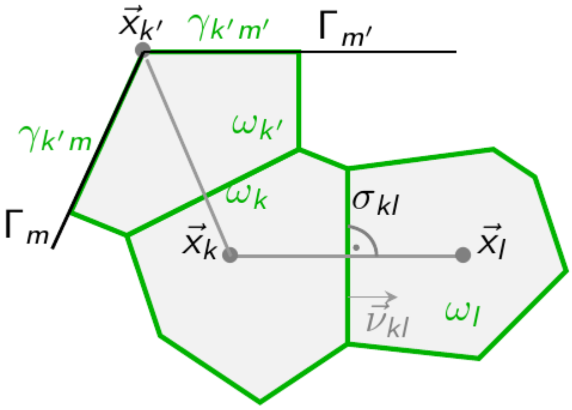
or

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

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

The discrete version

The finite volume discretization is based on the two-point flux Voronoi finite volume method.



Let N be the number of control volumes ω_k / collocation points $\vec{x}_k$. For $k = 1 \dots N$ write

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

- 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.

Example: a reaction-diffusion problem

Two species u_1 and u_2 move by diffusion in $\Omega = (0, 1)$. Both have initial concentration zero, and starting with $t = 0$, at $x = 0$, u_1 enters the domain with concentration 1. u_1 is not allowed to leave the domain at $x = 1$.

Within Ω, u_1 reacts to u_2 with foward reaction constant k^+ and backward reaction constant k^- The boundary $x = 0$ is insulating for u_2, and at $x = 1$, u_2 has forced concentration 0.

$$\begin{aligned}
\partial_t u_1 - \nabla \cdot D_1 \vec{\nabla} u_1 + r_1(u_1, u_2) &= 0 \\
\partial_t u_2 - \nabla \cdot D_2 \vec{\nabla} u_1 + r_2(u_1, u_2) &= 0 \\
r_1(u_1, u_2) &= k^+ u_1 - k^- u_2 \\
r_2(u_1, u_2) &= -r_1(u_1, u_2) \\
u_1|_{x=0} &= 1 \\
D_1 \vec{\nabla} u_1 \cdot \vec{n}|_{x=1} &= 0 \\
D_2 \vec{\nabla} u_2 \cdot \vec{n}|_{x=0} &= 0 \\
u_2|_{x=1} &= 0 \\
u_1|_{t=0} &= 0 \\
u_2|_{t=0} &= 0
\end{aligned}$$

```
begin
    const kp = 1
    const km = 1
    const D_1 = 0.5
    const D_2 = 0.1
end;

```

storage (generic function with 1 method)

```
function storage(f, u, node, data)
    f[1] = u[1]
    f[2] = u[2]
    return nothing
end

```

reaction (generic function with 1 method)

```
function reaction(f, u, node, data)
    r = kp * u[1] - km * u[2]
    f[1] = r
    f[2] = -r
    return nothing
end

```

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

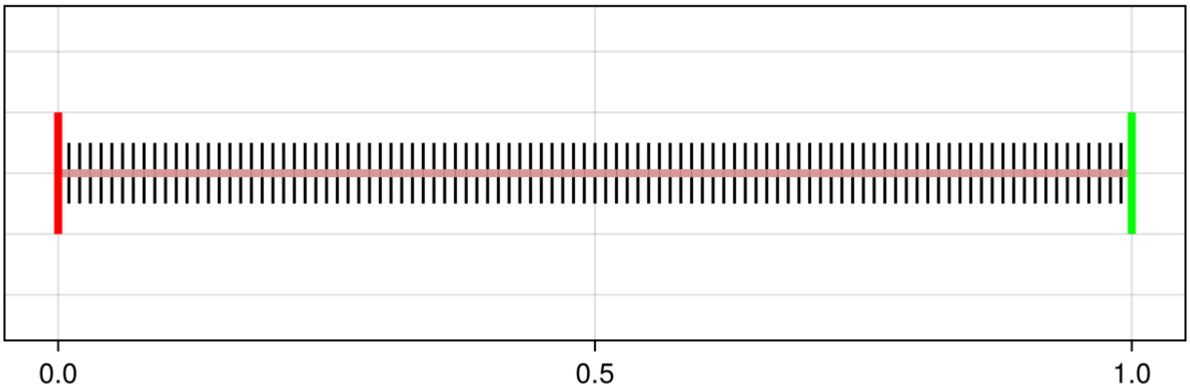
```
function bcondition(f, u, bnode, data)
    v = ramp(bnode.time, du = (0, 1), dt = (0, 1.0e-2))
    boundary_dirichlet!(f, u, bnode, species = 1, region = 1, value = v)
    boundary_neumann!(f, u, bnode, species = 1, region = 2, value = 0)
    boundary_dirichlet!(f, u, bnode, species = 2, region = 2, value = 0)
    boundary_neumann!(f, u, bnode, species = 2, region = 1, value = 0)
    return nothing
end

```

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

```
function flux(f, u, edge, data)
    f[1] = D_1 * (u[1, 1] - u[1, 2])
    f[2] = D_2 * (u[2, 1] - u[2, 2])
    return nothing
end

```

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

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




```
gridplot(grid; size = (600, 200))
```

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

```
system = VoronoiFVM.System(grid; flux, reaction, storage, bcondition, species =
  [1, 2])
```

```
tend = 10
```

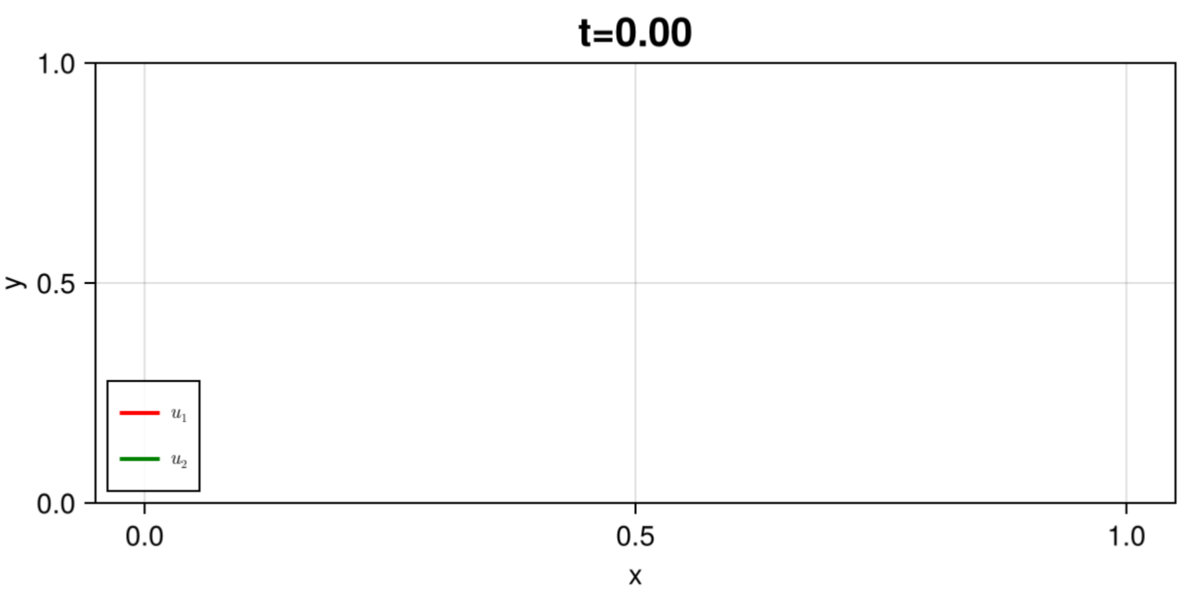
```
tend = 10
```

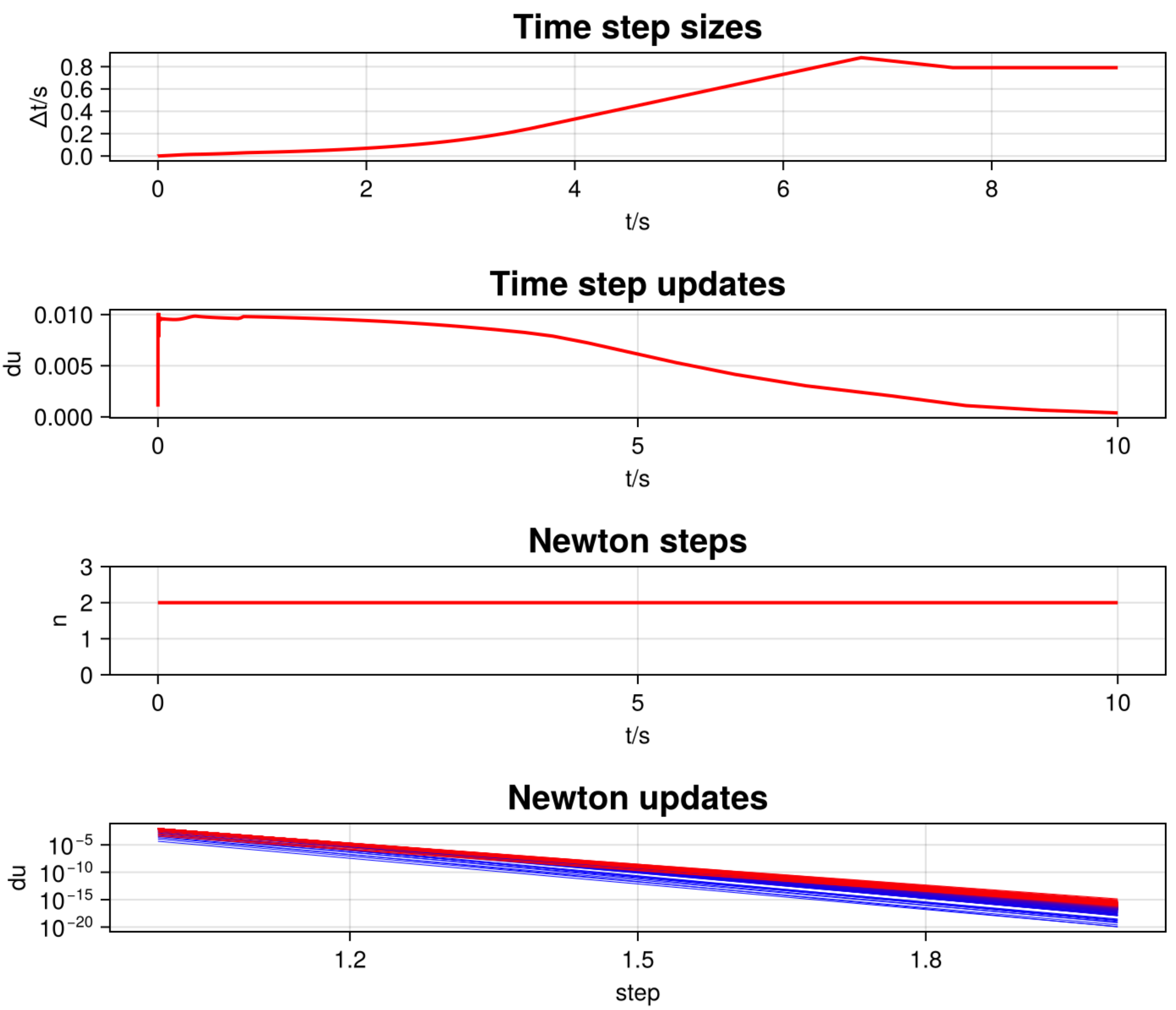
```
tsol =
t: 291-element Vector{Float64}:
  0.0
  1.0e-5
  2.2e-5
  3.64e-5
  5.3679999999999994e-5
  7.441599999999999e-5
  9.929919999999999e-5
  ⋮
  6.01378830802794
  6.747503004584978
  7.6279606404534235
  8.418640426968949
  9.209320213484474
 10.0
u: 291-element Vector{Matrix{Float64}}:
 [0.0 0.0 … 0.0 0.0; 0.0 0.0 … 0.0 0.0]
 [0.001 4.554843410322946e-5 … 1.5480027823016597e-136 1.407262463342749e-137; 9.81447
 [0.0021999999999999997 0.00015908130705043495 … 1.4009955298788406e-128 1.50105056903
 [0.0036399999999999996 0.0003701927174498243 … 5.196095466939289e-121 6.54045755236787
 [0.0053679999999999995 0.0007170798487162418 … 1.0109039306116876e-113 1.489440927662
 [0.007441599999999999 0.0012480797898037478 … 1.1121249039591822e-106 1.9100042604902
 [0.009929919999999998 0.002023311225746713 … 7.06416191759849e-100 1.407518950270931e
 ⋮
 [1.0 0.9951828455360225 … 0.6874277376172829 0.6873586244812638; 0.8128968654045824 0
 [1.0 0.9952259856410248 … 0.6891158547224582 0.689046719975269; 0.8159125544273736 0.
 [1.0 0.9952552929218204 … 0.6902622563812656 0.6901931068670524; 0.817962531028575 0.
 [1.0 0.9952708836996275 … 0.6908719847120693 0.6908028273158894; 0.819053459466289 0.
 [1.0 0.9952801198208002 … 0.6912331459324508 0.6911639838573514; 0.8196998777537228 0
 [1.0 0.9952855915529846 … 0.6914470904421581 0.6913779255916498; 0.8200828856140855 0
```

```
tsol = solve(system, times = (0, tend), Δu_opt = 0.01, Δt_min = 1.0e-5, Δt =
1.0e-5, log = true)
```

```
animatesolution1d (generic function with 1 method)
```

```julia
function animatesolution1d(
        system, tsol;
        video = "tmp.gif",
        nframes = 201,
        limits = (0, 1),
        size = (600, 300),
        pscale_bar = 1.0,
        Plotter = GridVisualize.default_plotter(),
        vis = nothing
    )
    title = ""
    vis = GridVisualizer(; size, limits, title, Plotter, legend = :lb)
    trange = range(extrema(tsol.t)...; length = nframes)
    movie(vis; file = video) do vis
        for t in trange
            title = @sprintf("t=%.2f", t)
            sol = tsol(t)
            scalarplot!(vis, system, sol; title, species = 1, label = L"u_1", color = :red)
            scalarplot!(vis, system, sol; species = 2, label = L"u_2", color = :green, clear = false)
            reveal(vis)
        end
    end
    return video
end

```






```
plothistory(tsol)
```

Example: the Brusselator system

Two species interacting via a reaction

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

with homogeneous Neumann boundary conditons

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

```
function bruss_storage(f, u, node, data)
    f[1] = u[1]
    f[2] = u[2]
    return nothing
end

```

```
ExtendableGrids.ExtendableGrid{Float64, Int32}
      dim =       2
   nnodes =    1681
   ncells =    3200
  nbfaces =     160
```

```
begin
    const A = 2.25
    const B = 7.0
    const bruss_D_1 = 0.005
    const bruss_D_2 = 0.1
    const pert = 0.1
    const bruss_tend = 50
    const dim = 1
    bruss_X = -1:0.01:1
    bruss_grid = simplexgrid(bruss_X)
    bruss_X2 = -1:0.05:1
    bruss_grid2 = simplexgrid(bruss_X2, bruss_X2)
end

```

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

```
function bruss_diffusion(f, u, edge, data)
    f[1] = bruss_D_1 * (u[1, 1] - u[1, 2])
    return f[2] = bruss_D_2 * (u[2, 1] - u[2, 2])
end

```

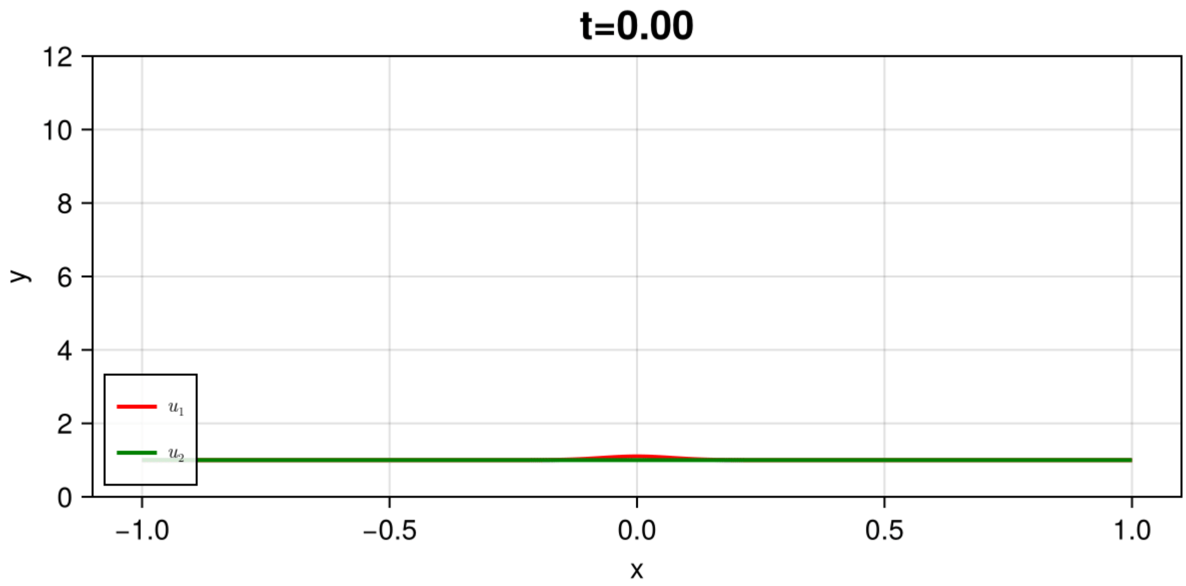
```
bruss_reaction (generic function with 1 method)
```

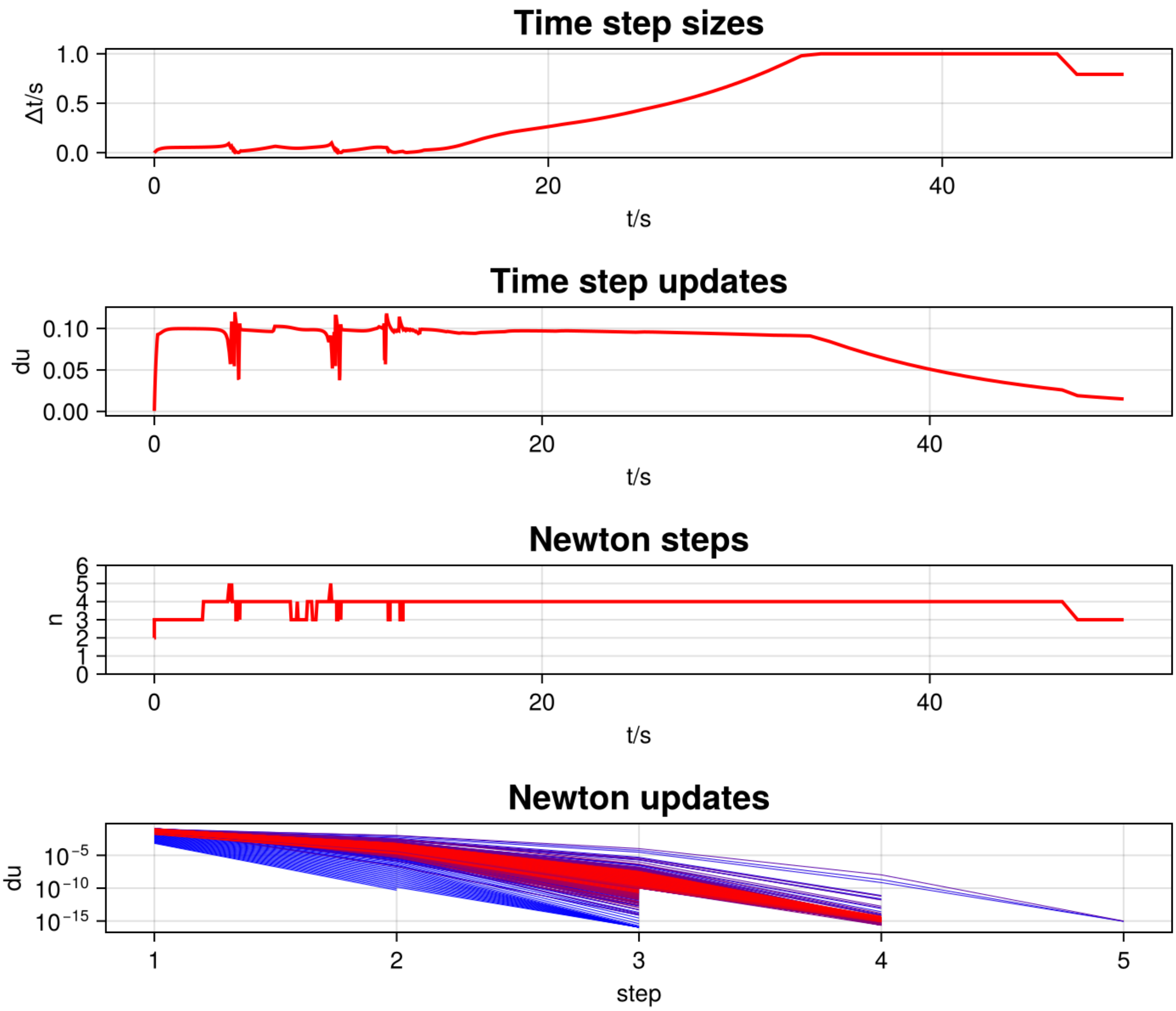
```
function bruss_reaction(f, u, node, data)
    f[1] = (B + 1.0) * u[1] - A - u[1]^2 * u[2]
    return f[2] = u[1]^2 * u[2] - B * u[1]
end

```

```
bruss_system =
VoronoiFVM.System{Float64, Float64, Int32, Int64, Matrix{Int32}}(
  grid = ExtendableGrids.ExtendableGrid{Float64, Int32}(dim=1, nnodes=201, ncells=200,
  nbfaces=2),
  physics = Physics(flux=bruss_diffusion, storage=bruss_storage,
  reaction=bruss_reaction, ),
  num_species = 2)
```

```
bruss_system = VoronoiFVM.System(
    bruss_grid,
    flux = bruss_diffusion,
    storage = bruss_storage,
    reaction = bruss_reaction,
    species = [1, 2]
)

```

```
begin
    inival = unknowns(bruss_system)
    coord = bruss_grid[Coordinates]
    fpeak(x) = exp(-norm(10 * x)^2)
    for i in 1:size(inival, 2)
        inival[1, i] = 1.0 + 0.1 * fpeak(coord[:, i])
        inival[2, i] = 1.0
    end
    bruss_tsol = solve(
        bruss_system; inival, times = (0, bruss_tend),
        Δu_opt = 0.1,
        Δt = 1.0e-4,
        Δt_min = 1.0e-6, Δt_max = tend / 10, log = true
    )
end;
```






```
plothistory(bruss_tsol)
```

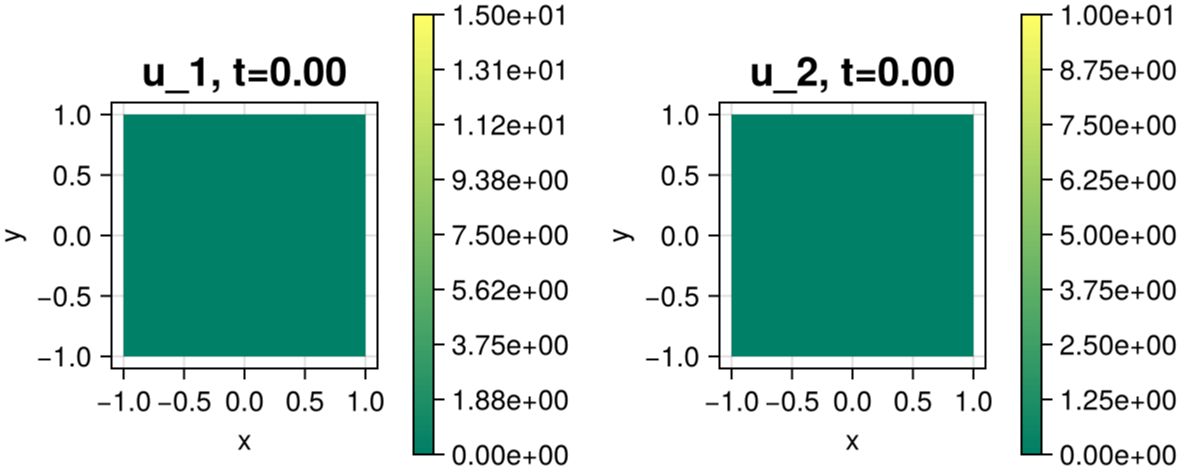
```
bruss_system2 =
VoronoiFVM.System{Float64, Float64, Int32, Int64, Matrix{Int32}}(
  grid = ExtendableGrids.ExtendableGrid{Float64, Int32}(dim=2, nnodes=1681, ncells=320
  nbfaces=160),
  physics = Physics(flux=bruss_diffusion, storage=bruss_storage,
  reaction=bruss_reaction, ),
  num_species = 2)
```

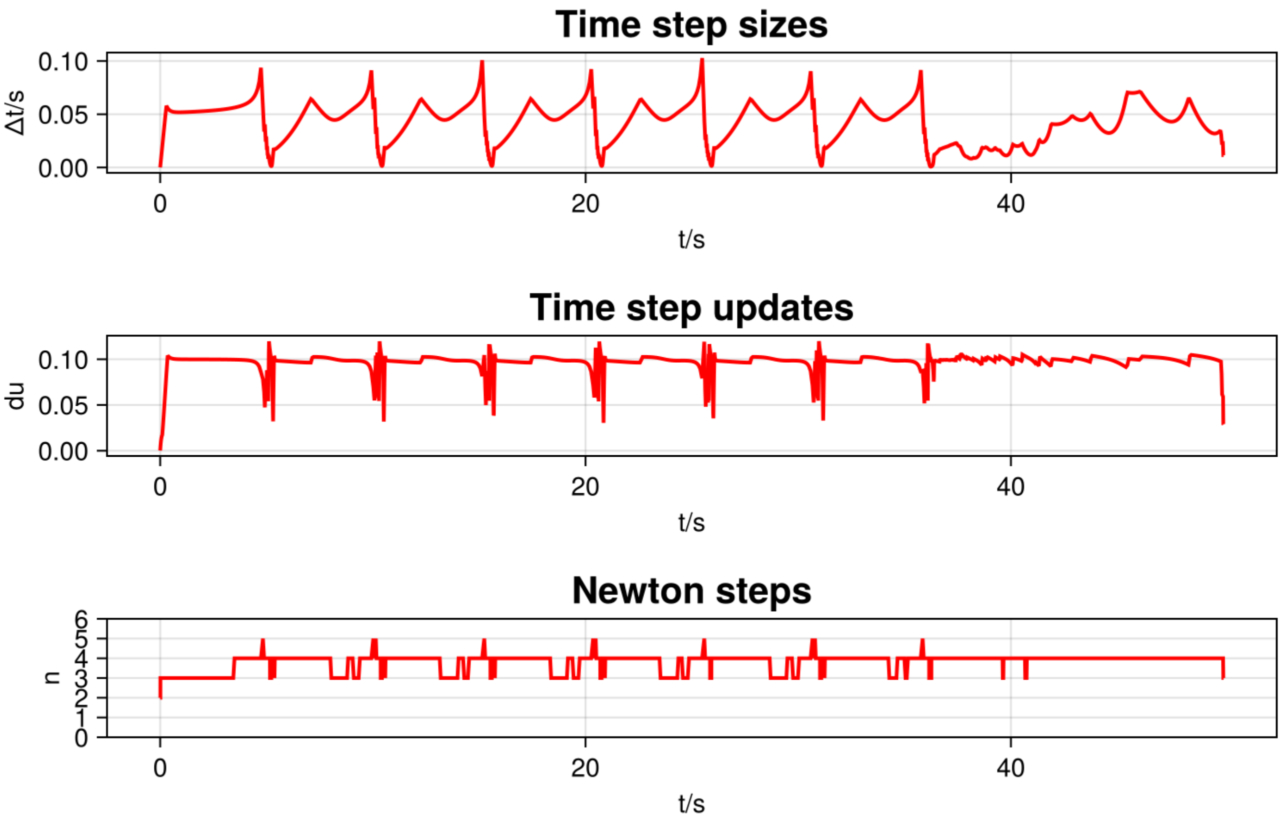
```
bruss_system2 = VoronoiFVM.System(
    bruss_grid2,
    flux = bruss_diffusion,
    storage = bruss_storage,
    reaction = bruss_reaction,
    species = [1, 2]
)
```

```
begin
    inival2 = unknowns(bruss_system2)
    coord2 = bruss_grid2[Coordinates]
    for i in 1:size(inival2, 2)
        inival2[1, i] = 1.0 + 0.1 * fpeak(coord2[:, i])
        inival2[2, i] = 1.0
    end
    bruss_tsol2 = solve(
        bruss_system2; inival2, times = (0, bruss_tend),
        Δu_opt = 0.1,
        reltol = 1.0e-10,
        Δt = 1.0e-4,
        Δt_min = 1.0e-6, Δt_max = tend / 10, log = true
    )
end;
```

animatesolution2d (generic function with 1 method)

```
function animatesolution2d(
        system, tsol;
        video = "tmp.gif",
        nframes = 201,
        size = (600, 300),
        pscale_bar = 1.0,
        Plotter = GridVisualize.default_plotter(),
        vis = nothing
    )
    title = ""
    vis = GridVisualizer(; layout = (1, 2), size, title, Plotter)
    trange = range(extrema(tsol.t)...; length = nframes)
    movie(vis; file = video) do vis
        for t in trange
            sol = tsol(t)
            title = @sprintf("u_1, t=%.2f", t)
            scalarplot!(vis[1, 1], system, sol; title, species = 1, label = "u_1", colormap = :summer, limits = (0, 15))
            title = @sprintf("u_2, t=%.2f", t)
            scalarplot!(vis[1, 2], system, sol; title, species = 2, label = "u_2", colormap = :summer, limits = (0, 10))
            reveal(vis)
        end
    end
    return video
end
```






```
1 plothistory(bruss_tsol2, plots = [:timestepsizes, :timestepupdates, :newtonsteps])
```

Table of Contents