```
PyPlotBackend()
```

```
begin
    using ShortCodes   , ExtendableGrids   , VoronoiFVM   , PlutoVista
    using Plots   , GridVisualize   , PlutoUI   , LaTeXStrings   , PyPlot
    using PyCall   , CSV   , DataFrames
    GridVisualize.default_plotter!(Plots);
    pyplot();
    #TableOfContents();
end
```

Bidomain problem

Table of Contents

Introduction

This document shall document and implement our strategy for solving the "bidomain problem". It introduces the problem at hand, illustrates our discretization strategy, displays our implementation of this strategy, and visualizes the resulting solutions to the problem.

The solution to the problem, once set up, is calculated by use of the VoronoiFVM.jl Julia package [1]. This package is available at **https://github.com/j-fu/VoronoiFVM.jl**.

The bidomain problem was chosen as the project examination topic by our group as part of the Scientific Computing course at the Teschnische Universität Berlin. This work was performed during the Winter Semester 2021/2022.

Bidomain problem

In summary, the bidomain problem is a system of partial differential equations modeling the propagation of electric signals throughout cardiac tissue[2]. Before going into more detail, it is convenient to provide some context on the lower fidelity, but simpler, "monodomain problem".

The monodomain problem models cardiac tissue's electircal properties by treating it as an intracellular region separated from a extracellular region (the electrical ground) by a membrane. This approach neglects current flow external to the cells [2].

The bidomain model then rectifies this issue by treating the intracellular and extracellular regions with their own current flows, linked by transmembrane potential. This allows the regions to be modeled with separate boundary conditions and for the application of external stimuli [2].

This document will use the formulation of the biodomain problem provided in Ether and Bourgalt, *Semi-Implicit Time-Discretization Schemes For The Bidomain Model* [3].

Biodomain Problem Modeled as a Partial Differential Equation

The Bidomain problem for membrane-based models is given by Ethier & Bourgalt [3] as

$$\begin{aligned}\frac{\partial u}{\partial t} &= \frac{1}{\varepsilon} f(u,v) + \nabla \cdot (\sigma_i \nabla u) + \nabla \cdot (\sigma_i \nabla u_e) \\ 0 &= \nabla \cdot (\sigma_i \nabla u + (\sigma_i + \sigma_e) \nabla u_e) \\ \frac{\partial v}{\partial t} &= \varepsilon g(u,v).\end{aligned}$$

where [3]:

- $u = u_i - u_e$ is the transmembrane potential
- σ_i and σ_e are second order tensors representing the intracellular and extracellular tissue's electrical conductivity in each spatial direction
- v is a lumped ionic variable
- ε is a parameter linked to the ratio between the repolarisation rate and the tissue excitation rate

Which can also be written in vector form as

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

where

$$\begin{aligned}\vec{u} &= \begin{bmatrix} u \\ u_e \\ v \end{bmatrix} \\ \vec{s}(\vec{u}) &= \begin{bmatrix} u \\ 0 \\ v \end{bmatrix} \\ \vec{j}(\vec{u}) &= \begin{bmatrix} -\sigma_i(\nabla u + \nabla u_e) \\ \sigma_i \nabla u + (\sigma_i + \sigma_e)\nabla u_e \\ 0 \end{bmatrix} \\ \vec{r}(\vec{u}) &= \begin{bmatrix} -f(u,v)/\varepsilon \\ 0 \\ -\varepsilon g(u,v) \end{bmatrix} \\ \vec{f} &= \vec{0}.\end{aligned}$$

With f and g defined as

$$\begin{aligned} f(u,v) &= u - \frac{u^3}{3} - v \\ g(u,v) &= u + \beta - \varphi v. \end{aligned}$$

The initial condition for this problem in 1D will be set so that

$$\begin{aligned} u = u_{0,1},\ v = v_0 \Rightarrow f(u_{0,1}, v_0) = g(u_{0,1}, v_0) = 0 \ \ \forall x \notin [0, L/20] \\ u = u_{0,2} = 2,\ v = v_0 \ \ \forall x \in [0, L/20] \\ u_e = u_{e,0} = 0 \ \ \forall x. \end{aligned}$$

We can get the exact solution to $f = g = 0$:

$$\begin{aligned} 0 = f(u_{0,1}, v_0) = u_{0,1} - \frac{u_{0,1}^3}{3} - v_0 \Rightarrow v_0 = u_{0,1} - \frac{u_{0,1}^3}{3} \\ 0 = g(u_{0,1}, v_0) = u_{0,1} + \beta - \varphi v_0 \Rightarrow 0 = u_{0,1} + \beta - \varphi\left(u_{0,1} - \frac{u_{0,1}^3}{3}\right) \Rightarrow \\ 0 = u_{0,1}^3 \frac{\varphi}{3} + u_{0,1}(1-\varphi) + \beta \end{aligned}$$

which has three solutions since it is a polynomial of degree 3. Two of them are complex, whilst the third one is real. We find the solution by using Newton's method since the expression for the solution of a 3rd degree polynomial is very verbose.

Finite Volume Discretization

When doing the finite volume discretization we need to split the polygonal domain Ω into finite volumes ω_k such that

$$\begin{aligned} \bar{\Omega} &= \bigcup_{k \in N} \bar{\omega}_k \\ N &= \{1, \dots, n_c\} \\ \partial\Omega &= \bigcup_{m \in G} \Gamma_m \\ G &= \{1, \dots, n_e\} \\ \vec{s}_{kl} &= \bar{\omega}_k \cup \bar{\omega}_l \end{aligned}$$

where n_c is the number of control volumes, n_e the number of edges of the polygonal domain. We have that $|s_{kl}| > 0$ makes ω_k,ω_l neighbours. We define the part of ω_k that is on the boundary of Ω as

$$\gamma_{km} = \partial\omega_k \cup \Gamma_m$$

Discretization of equation 1

Integrate over a control volume ω_k:

$$\int_{\omega_k} \frac{\partial u}{\partial t} = \int_{\omega_k} \frac{1}{\varepsilon} f(u,v) d\omega + \int_{\omega_k} \nabla \cdot (\sigma_i \nabla u) d\omega + \int_{\omega_k} \nabla \cdot (\sigma_i \nabla u_e) d\omega$$

Apply Gauss' thereom to the two divergence terms:

$$\int_{\omega_k} \frac{\partial u}{\partial t} = \int_{\omega_k} \frac{1}{\varepsilon} f(u,v) d\omega + \int_{\partial\omega_k} \sigma_i \nabla u \cdot \vec{n} ds + \int_{\partial\omega_k} \sigma_i \nabla u_e \cdot \vec{n} ds$$

Convert the surface integrals over the edge of the control volume into a sum of an integral over each side, as our Voronoi cell control volumes are polygons. Additionally, introduce a separate term for the surface integral over the specific case of sides that are boundary conditions.

$$\begin{aligned}\int_{\omega_k} \frac{\partial u}{\partial t} = & \int_{\omega_k} \frac{1}{\varepsilon} f(u,v) d\omega \\ & + \sum_{l \in N_k} \int_{s_{kl}} \sigma_i \nabla u \cdot \vec{n}_{kl} ds + \sum_{m \in B_k} \int_{\gamma_{kl}} \sigma_i \nabla u \cdot \vec{n}_m ds \\ & + \sum_{l \in N_k} \int_{s_{kl}} \sigma_i \nabla u_e \cdot \vec{n}_{kl} ds + \sum_{m \in B_k} \int_{\gamma_{kl}} \sigma_i \nabla u_e \cdot \vec{n}_m ds\end{aligned}$$

Exploiting the admissibility condition, approximate the dot products using finite differences:

$$\sigma_i \nabla u \cdot \vec{n} = \sigma_i \frac{u_k - u_l}{|x_k - x_l|} \sigma_i \nabla u_e \cdot \vec{n} = \sigma_i \frac{u_{e_k} - u_{e_l}}{|x_k - x_l|}$$

Substitute this approximation in and replace the integrals with a simple multiplication by the length of the cell border. Additionally expanding $f(u,v)$:

$$\begin{aligned}|\omega_k| \frac{\partial u_k}{\partial t} = & \int_{w_k} \frac{1}{\varepsilon} (u_k - \frac{u_k^3}{3} - v_k) d\omega \\ & + \sum_{l \in N_k} |s_{kl}| \sigma_i \frac{u_k - u_l}{|x_k - x_l|} + \sum_{m \in B_k} |\gamma_{km}| \sigma_i \frac{u_k - u_l}{|x_k - x_l|} \\ & + \sum_{l \in N_k} |s_{kl}| \sigma_i \frac{u_{e_k} - u_{e_l}}{|x_k - x_l|} + \sum_{m \in B_k} |\gamma_{km}| \sigma_i \frac{u_{e_k} - u_{e_l}}{|x_k - x_l|}\end{aligned}$$

Combine terms, and note the first integral simply becomes a multiplication by $|\omega_k|$:

$$|\omega_k|\frac{\partial u_k}{\partial t} = \frac{|\omega_k|}{\varepsilon}\left(u - \frac{u^3}{3} - v\right) + \sum_{l\in N_k}|s_{kl}|\sigma_i\frac{u_k - u_l + u_{e_k} - u_{e_l}}{|x_k - x_l|} + \sum_{m\in B_k}|\gamma_{km}|\sigma_i\frac{u_k - u_l + u_{e_k} - u_{e_l}}{|x_k - x_l|}$$

Move the boundary terms to the other side and organize so it is on the right:

$$|\omega_k|\frac{\partial u_k}{\partial t} + \sum_{l\in N_k}|s_{kl}|\sigma_i\frac{u_k - u_l + u_{e_k} - u_{e_l}}{|x_k - x_l|} + \frac{|\omega_k|}{\varepsilon}\left(u_k - \frac{u_k^3}{3} - v_k\right) = - \sum_{m\in B_k}|\gamma_{km}|\sigma_i\frac{u_k - u_l + u_{e_k} - u_{e_l}}{|x_k - x_l|}$$

Then discretizing in time yields:

$$\frac{|\omega_k|}{\Delta t}(u_k^n - u_k^{n-1}) + \sum_{l\in N_k}|s_{kl}|\sigma_i\frac{u_k^\theta - u_l^\theta + u_{e_k}^\theta - u_{e_l}^\theta}{|x_k - x_l|} + \frac{|\omega_k|}{\varepsilon}\left(u_k^\theta - \frac{(u_k^\theta)^3}{3} - v_k^\theta\right) = - \sum_{m\in B_k}|\gamma_{km}|\sigma_i\frac{u_k^\theta - u_l^\theta + u_{e_k}^\theta - u_{e_l}^\theta}{|x_k - x_l|}$$

Where $u_k^\theta = \theta u_k^n + (1+\theta)u_k^{n-1}$

Discretization of equation 2

Distrbute the divergence and integrate over a volume ω_k:

$$0 = \int_{\omega_k} \nabla \cdot (\sigma_i \nabla u) d\omega + \int_{\omega_k} \nabla \cdot (\sigma_i + \sigma_e) \nabla u_e d\omega$$

Apply Gauss' theorem:

$$0 = \int_{\partial\omega_k} (\sigma_i \nabla u) \cdot \vec{n} ds + \int_{\partial\omega_k} (\sigma_i + \sigma_e) \nabla u_e \cdot \vec{n} ds$$

Again convert these terms to a sum of the integral over each side of the volume, and add terms for the boundary conditions:

$$\begin{aligned} 0 = &\sum_{l \in N_k} \int_{s_{kl}} \sigma_i \nabla u \cdot \vec{n}_{kl} ds + \sum_{m \in B_k} \int_{\gamma_{kl}} \sigma_i \nabla u \cdot \vec{n}_m ds \\ &+ \sum_{l \in N_k} \int_{s_{kl}} (\sigma_i + \sigma_e) \nabla u_e \cdot \vec{n}_{kl} ds + \sum_{m \in B_k} \int_{\gamma_{kl}} (\sigma_i + \sigma_e) \nabla u_e \cdot \vec{n}_m ds \end{aligned}$$

Again approximate the dot products as finite differences, and replace the integrals by the length of the cell border:

$$\begin{aligned} 0 = &\sum_{l \in N_k} |s_{kl}| \sigma_i \left(\frac{u_k - u_l}{|x_k - x_l|} \right) + \sum_{m \in B_k} |\gamma_{km}| \sigma_i \left(\frac{u_k - u_l}{|x_k - x_l|} \right) \\ &+ \sum_{l \in N_k} |s_{kl}| (\sigma_i + \sigma_e) \left(\frac{u_{e_k} - u_{e_l}}{|x_k - x_l|} \right) + \sum_{m \in B_k} |\gamma_{km}| (\sigma_i + \sigma_e) \left(\frac{u_{e_k} - u_{e_l}}{|x_k - x_l|} \right) \end{aligned}$$

Move the boundary conditions to the opposite side:

$$\begin{aligned} &\sum_{l \in N_k} |s_{kl}| \sigma_i \frac{u_k - u_l}{|x_k - x_l|} + \sum_{l \in N_k} |s_{kl}| (\sigma_i + \sigma_e) \frac{u_{e_k} - u_{e_l}}{|x_k - x_l|} = \\ &- \sum_{m \in B_k} |\gamma_{km}| \sigma_i \frac{u_k - u_l}{|x_k - x_l|} - \sum_{m \in B_k} |\gamma_{km}| (\sigma_i + \sigma_e) \frac{u_{e_k} - u_{e_l}}{|x_k - x_l|} \end{aligned}$$

Then discretizing in time yields:

$$\begin{aligned} &\sum_{l \in N_k} |s_{kl}| \sigma_i \frac{u_k^\theta - u_l^\theta}{|x_k - x_l|} + \sum_{l \in N_k} |s_{kl}| (\sigma_i + \sigma_e) \frac{u_{e_k}^\theta - u_{e_l}^\theta}{|x_k - x_l|} = \\ &- \sum_{m \in B_k} |\gamma_{km}| \sigma_i \frac{u_k^\theta - u_l^\theta}{|x_k - x_l|} - \sum_{m \in B_k} |\gamma_{km}| (\sigma_i + \sigma_e) \frac{u_{e_k}^\theta - u_{e_l}^\theta}{|x_k - x_l|} \end{aligned}$$

Discretization of equation 3

With

$$\frac{\partial v}{\partial t} - \varepsilon(u + \beta - \varphi v) = 0$$

We take the integral as before with respect to the volume ω_k:

$$\int_{\omega_k} \frac{\partial v}{\partial t} d\omega - \int_{\omega_k} \varepsilon(u + \beta - \varphi v) d\omega = 0$$

The integral is simply a multiplication by the area of the volume:

$$|\omega_k| \frac{\partial v_k}{\partial t} - \varepsilon |\omega_k| (u_k + \beta - \varphi v_k) = 0$$

Then discretizing in time yields:

$$\frac{|\omega_k|}{\Delta t}(v_k^n - v_k^{n-1}) - \varepsilon |\omega_k| (u_k^\theta + \beta - \varphi v_k^\theta) = 0$$

This concludes the space discretization.

Time discretization

There are a few possibilities for the time discretization:

Implicit Euler

With an implicit euler we approximate $\frac{\partial v}{\partial t}$ by a finite difference between the current time step and the last time step. That is, we set $\theta = 0$

This provides an easy implementation as all we need to implement this approximation is a storage of the previous $\mathbf{u}$. We can then simply solve the resulting system to find the current $\mathbf{u}$.

Explicit Euler

The explicit or forward Euler yields difficulties when implementing because the time step size must then be much smaller than the grid resultion. In fact, According to the paper by Ethier and Bourgalt, we must have $\Delta t \in O(\min(\varepsilon / L_f, \frac{m_i^3}{M_i^4} h^2))$ where L_f is a Lipschitz constant for f and where m_i and m_e are the 1-ellipticity constants for u and u_e.

Note: This analysis is done with a system discretized using FEM instead of FVM, however, its reasonable to assume the stability results are also valid here, as the system properties remain the same irregardless the discretization method applied.

Discretization summary

In total, for the discretization of the problem, we are left with a nonlinear system of equations that must be solved at each time step:

$$\frac{|\omega_k|}{\Delta t}(u_k^n - u_k^{n-1}) + \sum_{l \in N_k} |s_{kl}|\sigma_i \frac{u_k^\theta - u_l^\theta + u_{e_k}^\theta - u_{e_l}^\theta}{|x_k - x_l|} + \frac{|\omega_k|}{\varepsilon}\left(u_k^\theta - \frac{(u_k^\theta)^3}{3} - v_k^\theta\right) =$$
$$- \sum_{m \in B_k} |\gamma_{km}|\sigma_i \frac{u_k^\theta - u_l^\theta + u_{e_k}^\theta - u_{e_l}^\theta}{|x_k - x_l|}$$

$$\sum_{l \in N_k} |s_{kl}|\sigma_i \frac{u_k^\theta - u_l^\theta}{|x_k - x_l|} + \sum_{l \in N_k} |s_{kl}|(\sigma_i + \sigma_e)\frac{u_{e_k}^\theta - u_{e_l}^\theta}{|x_k - x_l|} =$$
$$- \sum_{m \in B_k} |\gamma_{km}|\sigma_i \frac{u_k^\theta - u_l^\theta}{|x_k - x_l|} - \sum_{m \in B_k} |\gamma_{km}|(\sigma_i + \sigma_e)\frac{u_{e_k}^\theta - u_{e_l}^\theta}{|x_k - x_l|}$$

$$\frac{|\omega_k|}{\Delta t}(v_k^n - v_k^{n-1}) - \varepsilon|\omega_k|(u_k^\theta + \beta - \varphi v_k^\theta) = 0$$

This yields a system of $\sum_{k \in N} \sum_{l \in N_k} 6$ nonlinear equations to be solved in each time step.

Possible solution methods

Possible solution methods for our aquired nonlinear system could be fixpoint iteration, which gives a large area of convergence, but it could be really slow. Other methods include Newton iteration which would need a better initial guess for convergence, but the convergence near the solution is quadratic.

To simplify Newton iteration you can use dual numbers for automatic differentiation, as the VoronoiFVM package does.

Additionally, for performance improvements, you can use a damped Newton scheme or even an embedded newton scheme if the problem is parameter dependant.

Discrete solution using VoronoiFVM

1D Problem

```
begin
    β = 1.0
    γ = 0.5
    ε = 0.1
    σᵢ_normal = 1.0
    σₑ_normal = 1.0
    σᵢ_anisotropic = 25*[0.263 0; 0 0.0263]
    σₑ_anisotropic = 25*[0.263 0; 0 0.1087]
end;
```

```
begin
    L = 70
    T = 30
end;
```

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

```
f(u,v) = u - u^3/3 - v
```

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

```
g(u,v) = u + β - γ*v
```

fg_zeros

```
fg_zeros()
```

Returns the approximate solution to f=g=0.

```
"""
    fg_zeros()

Returns the approximate solution to f=g=0.
"""
function fg_zeros()
    p = 3(1-γ)/γ; q = 3β/γ;
    h(u) = u^3 + p*u + q; ḣ(u) = 3u^2 + p
    uₙ = 1; hₙ = 1
    cnt = 0
    while abs(hₙ) > 1e-15
        uₙ = uₙ - h(uₙ)/ḣ(uₙ)
        hₙ = h(uₙ)
    end
    vₙ = uₙ - uₙ^3/3
    uₙ, vₙ
end
```

```
(-1.28791, -0.57582)
```

```
u₀, v₀ = fg_zeros()
```

create_grid

```
create_grid(N, dim=1)
```

Creates a simplex grid in dimension `dim` with `N` grid points in one dimension with length `L`.

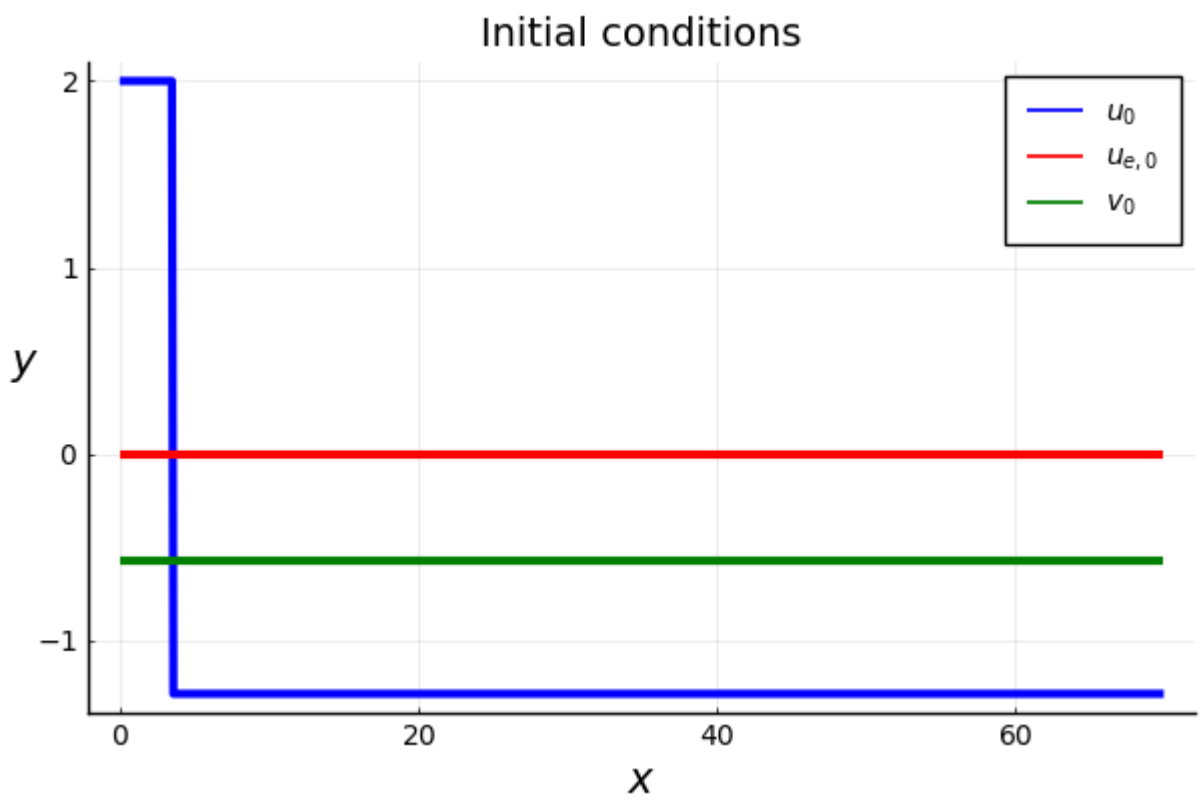
```
"""
    create_grid(N, dim=1)

Creates a simplex grid in dimension `dim` with `N` grid points in one
dimension with length `L`.
"""
function create_grid(N, dim=1)
    if dim == 2
        X = LinRange(0,L,N[1])
        Y = LinRange(0,L,N[2])
        simplexgrid(X,Y)
    else
        X = LinRange(0,L,N)
        simplexgrid(X)
    end
end
```

```
Replacing docs for `Main.workspace#2.create_grid :: Union{Tuple{Any}, Tuple
{Any, Any}}` in module `Main.workspace#2`
```

Functions for getting initial conditions

```
ú₀_2D (generic function with 1 method)
```

```
begin
    function ũ₀(x)
        u = u₀; v = v₀
        if 0 ≤ x ≤ L/20
            u = 2
        end
        uₑ = 0
        return [u, uₑ, v]
    end
    function ũ₀(x,y)
        ũ₀(x)
    end
    function ũ₀_2D(x,y)
        u = u₀; v = v₀
        if 0 ≤ x ≤ 3.5 && 0 ≤ y ≤ 70
            u = 2
        end
        if 31 ≤ x ≤ 39 && 0 ≤ y ≤ 35
            v = 2
        end
        uₑ = 0
        return [u, uₑ, v]
    end
end
```



function for getting the problem physics depending on the setup

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

```
function create_physics(σᵢ_orig, σₑ_orig; initial2D=false, anisotropic=false)
    physics = VoronoiFVM.Physics(
        storage = function(y,u,node)
            y[1] = u[1]
            y[2] = 0
            y[3] = u[3]
        end,
        flux = function(y,u,edge)
            σᵢ, σₑ = anisotropic ? get_σₛ(σᵢ_orig,σₑ_orig,edge) : (σᵢ_orig,σₑ_orig)
            y[1] = -σᵢ*(u[1,2]-u[1,1] + u[2,2]-u[2,1])
            y[2] = σᵢ*(u[1,2]-u[1,1]) + (σᵢ+σₑ)*(u[2,2]-u[2,1])
            y[3] = 0
        end,
        reaction = function(y,u,node)
            y[1] = -f(u[1],u[3])/ε
            y[2] = 0
            y[3] = -ε*g(u[1],u[3])
        end,
        breaction = function(y,u,node)
            if node.coord[:,node.index] == [0, 0]
                y[2] = 0
            end
        end,
    )
end
```

Function for getting the anisotropic flux

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

```
function get_σₛ(σᵢ,σₑ,edge)
    n1 = edge.coord[:,edge.node[1]]
    n2 = edge.coord[:,edge.node[2]]
    x = abs(n1[1] - n2[1]); y = abs(n1[2] - n2[2])
    θ = atan(y/x)
    l = [1 1]; r = [cos(θ);sin(θ)]
    σᵢ_θ = (l*σᵢ*r)[1]; σₑ_θ = (l*σₑ*r)[1]
    σᵢ_θ, σₑ_θ
end
```

bidomain

```
bidomain(;N=100, dim=1, Δt=1e-4, tₑ=T)
```

Solves the bidomain problem in `dim` dimensions with `N` grid points in each dimension. Uses Δt as time step until final time t_e.

```
"""
    bidomain(;N=100, dim=1, Δt=1e-4, tₑ=T)

Solves the bidomain problem in `dim` dimensions with `N` grid points in each
dimension. Uses Δt as time step until final time tₑ.
"""
function bidomain(;N=100, dim=1, Δt=1e-3, T=30, Plotter=Plots,
        initial2D=false, Tinit_solve=40, use_csv=false, anisotropic=false)
    xgrid = create_grid(N, dim)

    σᵢ, σₑ = anisotropic ? (σᵢ_anisotropic, σₑ_anisotropic) : (σᵢ_normal, σₑ_normal)
    physics = create_physics(σᵢ,σₑ; anisotropic=anisotropic)

    sys = VoronoiFVM.System(
        xgrid,
        physics,
        unknown_storage=:sparse,
    )

    boundaries = (dim == 1 ? 2 : 4)
    enable_species!(sys, species=[1,2,3])
    if !initial2D || true
        boundary_dirichlet!(sys,2,1,0)
    end
    for ispec ∈ [1 3]
        for ibc=1:boundaries
            boundary_neumann!(sys,ispec,ibc,0)
        end
    end

    init = initial_cond(sys, xgrid, Δt, Tinit_solve,
        dim, initial2D, use_csv, anisotropic)
    U = unknowns(sys)

    SolArray = copy(init)
    tgrid = initial2D ? (Tinit_solve:Δt:T+Tinit_solve) : (0:Δt:T)
    for t ∈ tgrid[2:end]
        solve!(U, init, sys; tstep=Δt)
        init .= U
        SolArray = cat(SolArray, copy(U), dims=3)
    end
    vis = GridVisualizer(resolution=(400,300), dim=dim, Plotter=Plotter)
    return xgrid, tgrid, SolArray, vis, sys
end
```

```
Replacing docs for `Main.workspace#68.bidomain :: Tuple{}` in module `Main.w
orkspace#68`
```

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

```
function initial_cond(sys, xgrid, Δt, Tinit_solve,
        dim, initial2D, use_csv, anisotropic)
    init = unknowns(sys)
    U = unknowns(sys)
    filename="initial_2D"*(anisotropic ? "_anisotropic" : "")*".csv"
    if dim==2 && initial2D
        if use_csv
            init = get_initial_data(filename)
        else
            inival = map(ũ₀_2D, xgrid)
            init .= [tuple[k] for tuple in inival, k in 1:3]'
            for t ∈ 0:Δt:Tinit_solve
                solve!(U, init, sys; tstep=Δt)
                init .= U
            end
            save_initial(init, filename)
        end
    else
        inival = map(ũ₀, xgrid)
        init .= [tuple[k] for tuple in inival, k in 1:3]'
    end
    init
end
```

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

```
function save_initial(init, filename)
    mkpath("../csv")
    df = DataFrame(init, :auto)
    CSV.write("../csv/$filename", df)
end
```

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

```
function get_initial_data(filename)
    Array(DataFrame(CSV.File("../csv/$filename")))
end
```

species = $[u,\ u_e,\ v]$

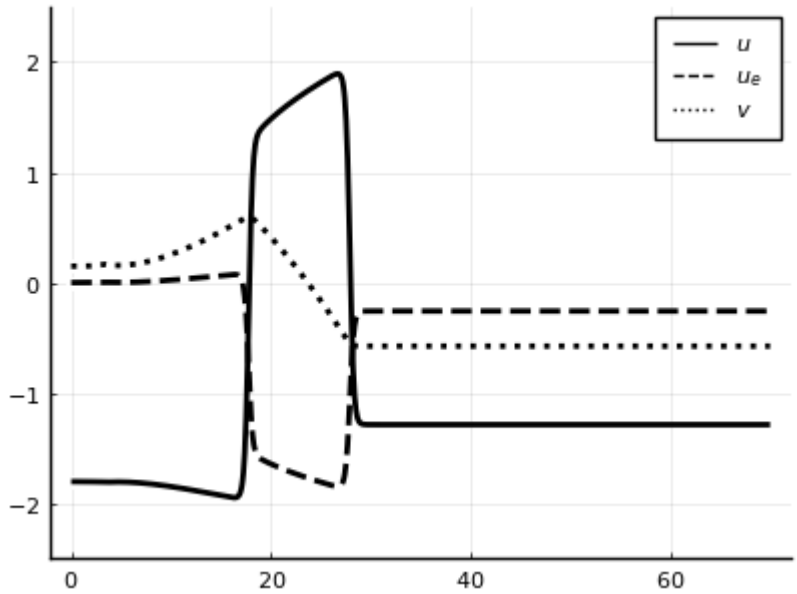
```
species = [L"u", L"u_e", L"v"]
```

```
begin
    dim = 1; N = 1000; Δt = 1e-1;
    xgrid, tgrid, sol, vis = bidomain(dim=dim, N=N, T=T, Δt=Δt);
end;
```

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

```
function plot_at_t(t,vis,xgrid,sol; title="")
    tₛ = Int16(round(t/Δt))+1
    scalarplot!(vis, xgrid, sol[1,:,tₛ], linestyle=:solid)
    scalarplot!(vis, xgrid, sol[2,:,tₛ], linestyle=:dash)
    plotted_sol = scalarplot!(
        vis, xgrid, sol[3,:,tₛ], linestyle=:dot, legend=:best, show=true, title=title)
    for (i,spec) ∈ zip(2 .* (1:length(species)), species)
        plotted_sol[1][i][:label] = spec
    end
    Plots.plot!(labels=species)
    Plots.plot!(ylims=(-2.5,2.5))
    plotted_sol
end
```

Solution to 1D problem for t=11




```
plot_at_t(11,vis,xgrid,sol;title="1D problem at t=11")
```

```
begin
    anim = @animate for t in 0:Δt*5:T
        plot_at_t(t,vis,xgrid,sol)
    end
    gif(anim, "../movies/1D_solution.gif", fps=15)
end;
```

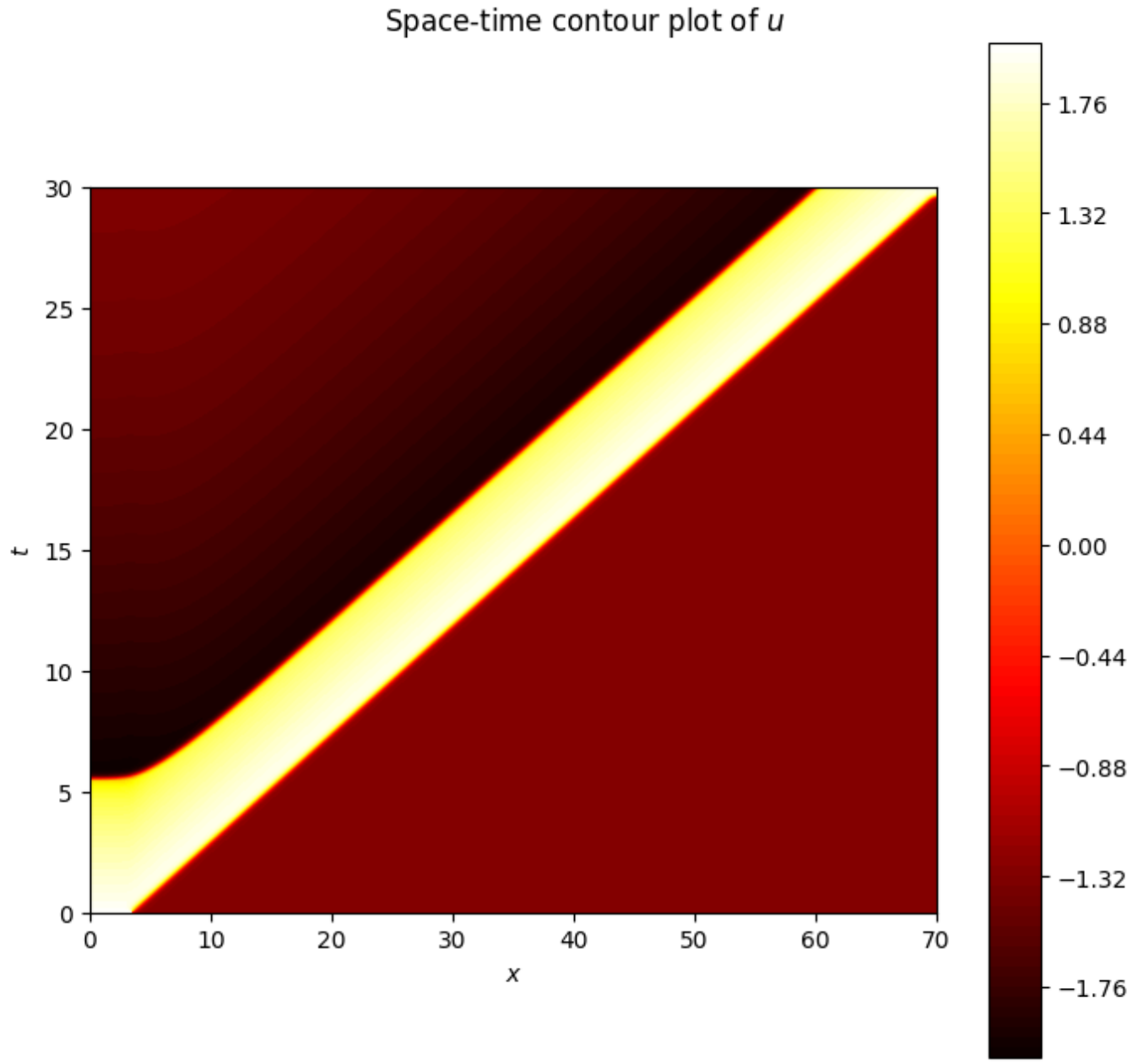
```
Saved animation to
fn:
    "/mnt/c/Users/sanmi/Desktop/masters/semester1/scicomp/SciCompFinalProject/movies/1D_sc
```

```
plot_species_3d (generic function with 1 method)
```

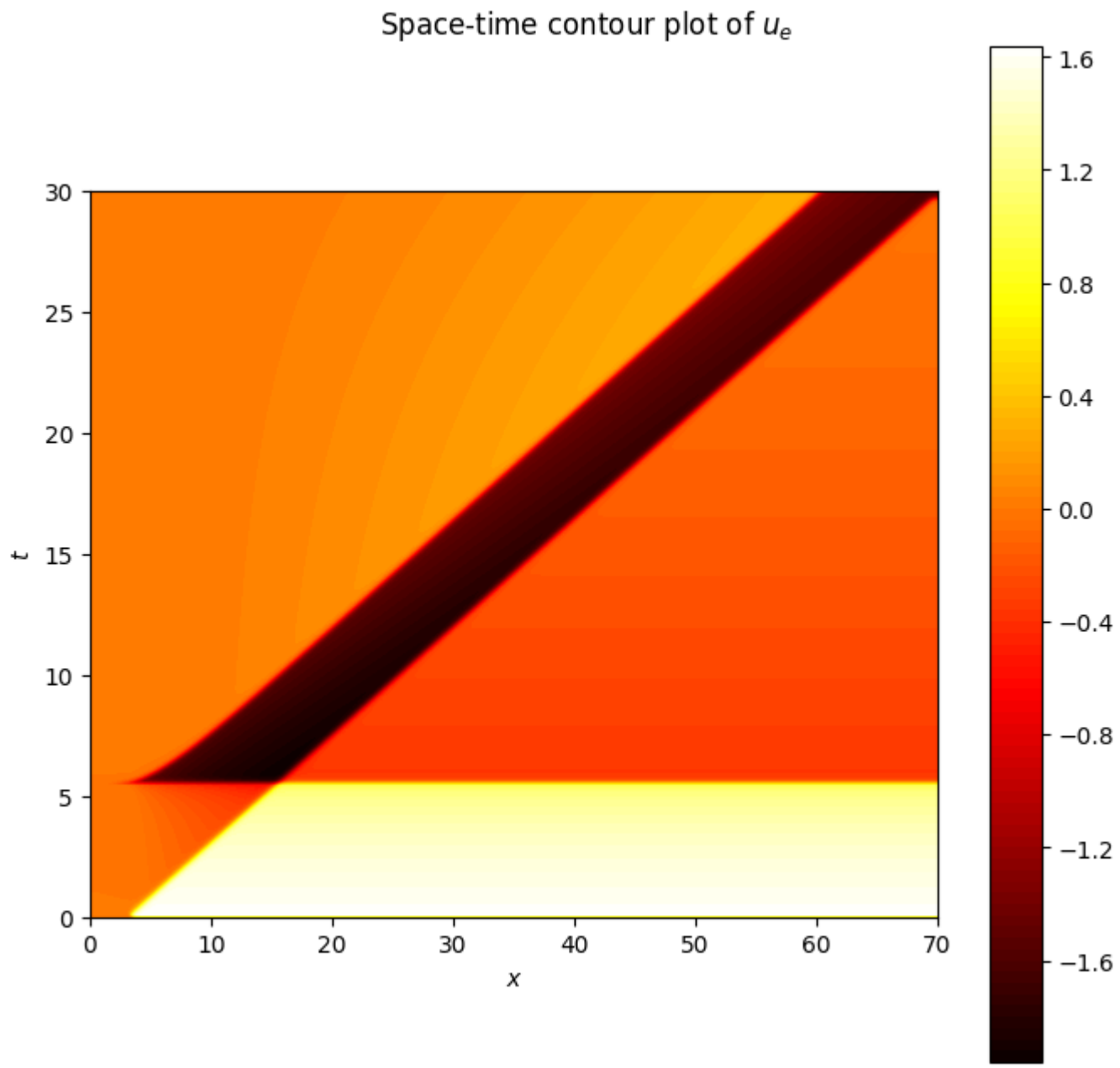
```
function plot_species_3d(spec)
    Xgrid = xgrid[Coordinates][:]; Tgrid = tgrid; Zgrid = sol[spec,:,:];
    xstep = 1; tstep = 1;
    Tgrid = Tgrid[1:xstep:end];
    Xgrid = Xgrid[1:tstep:end]
    Zgrid = Zgrid[1:tstep:end,1:xstep:end];
    PyPlot.clf()
    PyPlot.suptitle("Space-Time plot for "*species[spec])
    PyPlot.surf(Xgrid,Tgrid,Zgrid',cmap=:coolwarm) # 3D surface plot
    ax=PyPlot.gca(projection="3d")  # Obtain 3D plot axes
    α = 30; β = 240
    ax.view_init(α,β) # Adjust viewing angles

    PyPlot.xlabel(L"X")
    PyPlot.ylabel(L"T")
    PyPlot.zlabel(L"u")
    figure=PyPlot.gcf()
    figure.set_size_inches(7,7)
    figure
end
```

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

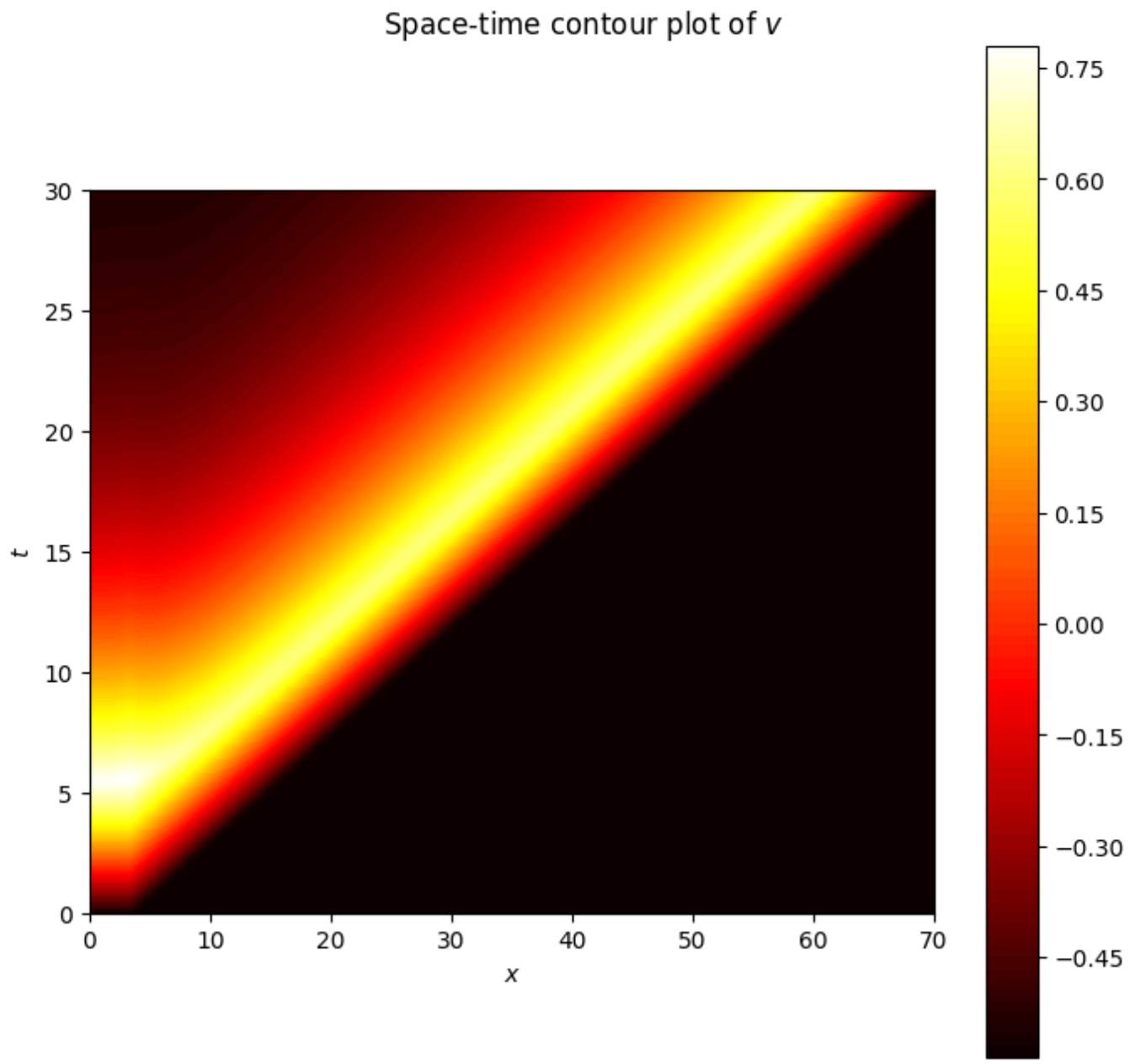
```
function contour_plot(spec; initial2D=false, anisotropic=false)
    PyPlot.clf()
    Xgrid = xgrid[Coordinates][:]
    Tgrid = tgrid
    Zgrid = sol[spec,:,:]
    PyPlot.suptitle("Space-time contour plot of "*species[spec])

    PyPlot.contourf(Xgrid,Tgrid,Zgrid',cmap="hot",levels=100)
    axes=PyPlot.gca()
    axes.set_aspect(2)
    PyPlot.colorbar()

    PyPlot.size(600,600)
    PyPlot.xlabel(L"x")
    PyPlot.ylabel(L"t")
    figure=PyPlot.gcf()
    figure.set_size_inches(7,7)
    extra = initial2D ? ("_inital2D") : (anisotropic ? "_anisotropic" : "")
    PyPlot.savefig("../img/st_contour_plot_spec_"*string(spec)*extra)
    figure
end
```




```
contour_plot(1)
```




```
contour_plot(2)
```




```
contour_plot(3)
```

2D Problem with 1D problem setup

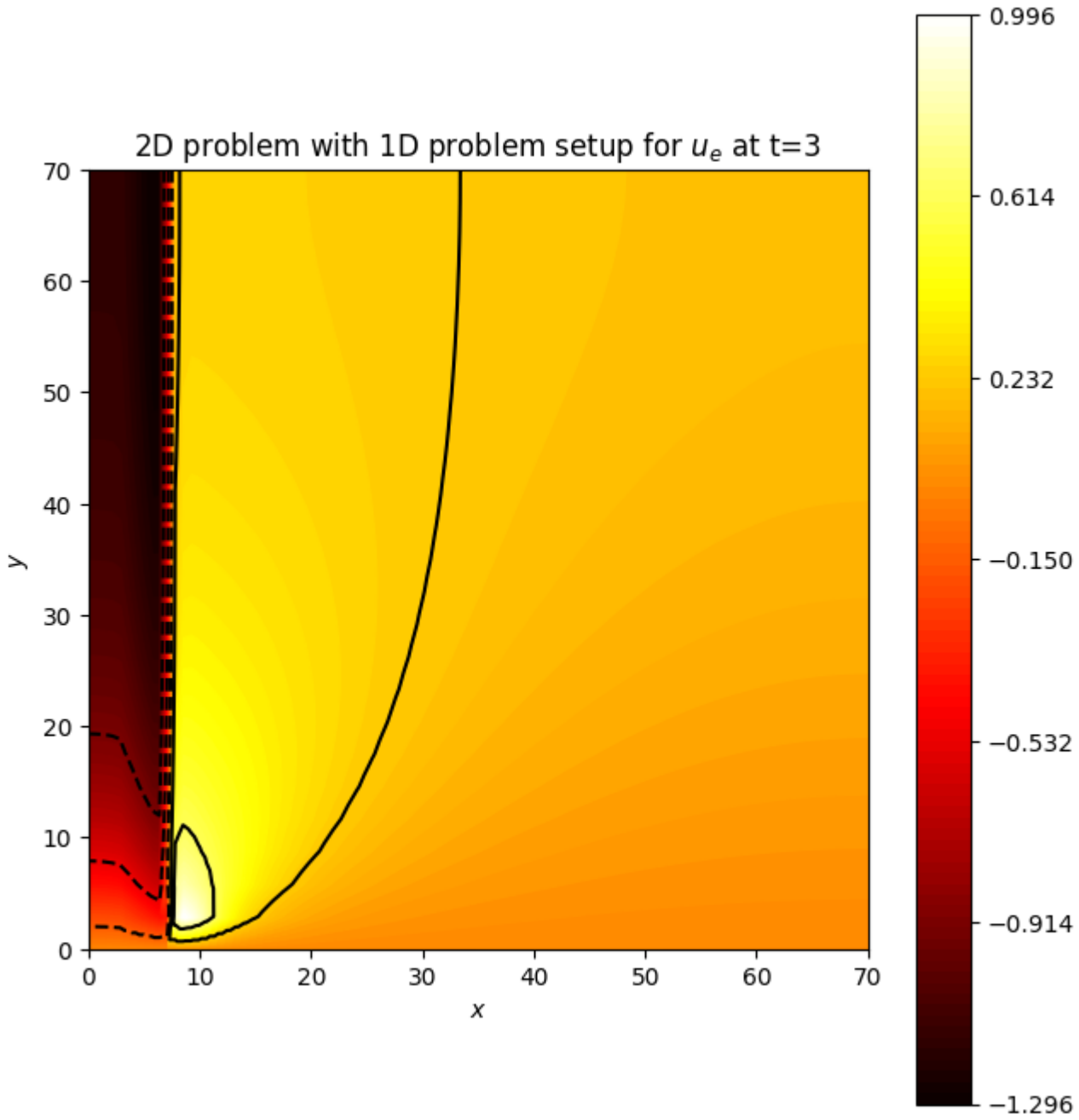
```
begin
    dim₂=2; N₂ = (100,25); Δt₂ = 1e-1;
    xgrid₂, tgrid₂, sol₂, vis₂ = bidomain(dim=dim₂, N=N₂,T=T, Δt=Δt₂, Plotter=PyPlot);
end;
```

```
contour_2d_at_t (generic function with 1 method)
```

```
function contour_2d_at_t(spec, t, Δt, xgrid, sol, title)
    tₛ = Int16(round(t/Δt))+1
    p = scalarplot(
        xgrid,sol[spec,:,tₛ], Plotter=PyPlot, colormap=:hot,
        title=title, colorlevels=100, isolines=0)
    p.set_size_inches(7,7)
    PyPlot.xlabel(L"x")
    PyPlot.ylabel(L"y")
    p
end
```

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

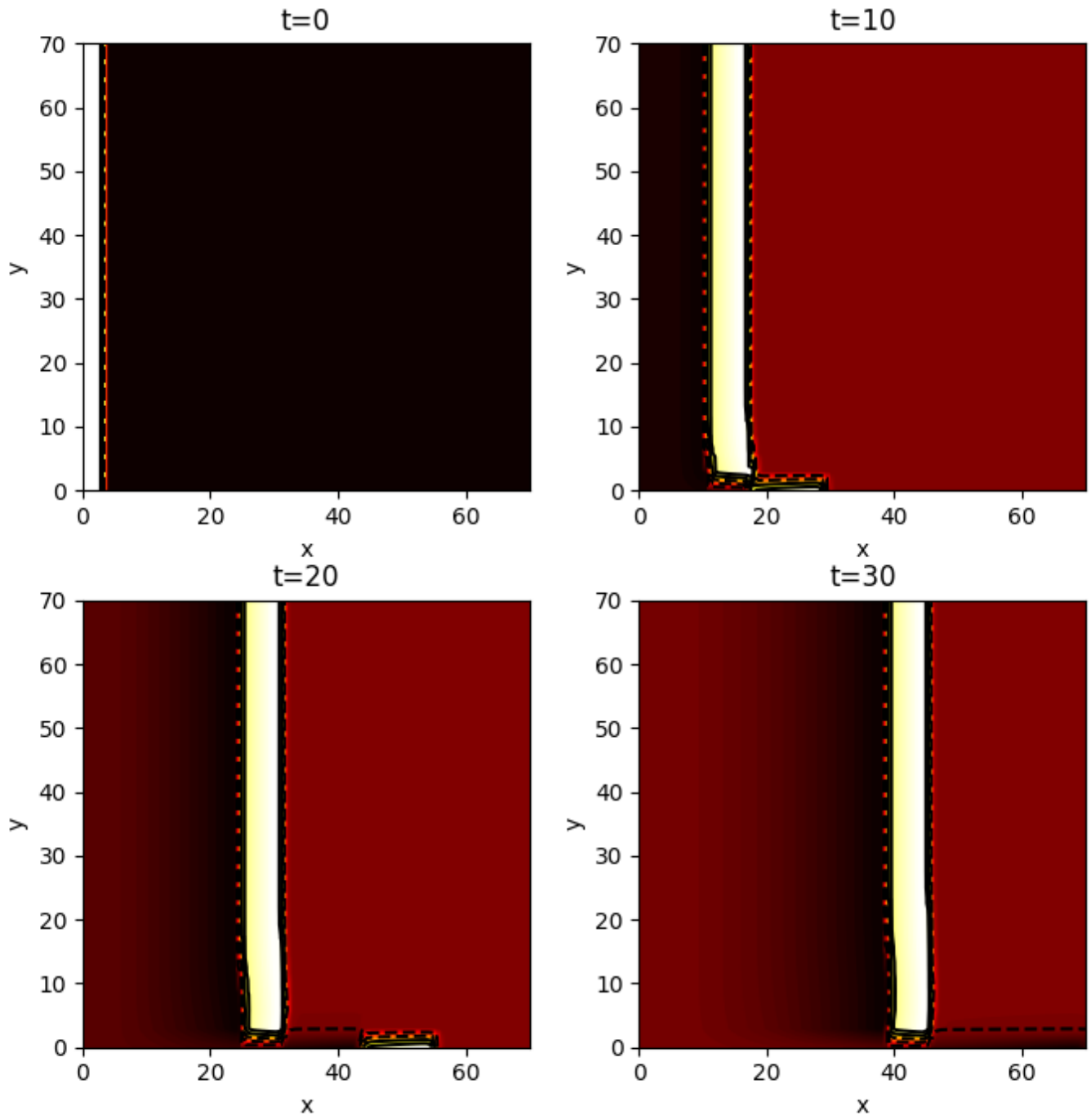
```
function contour_subplots(spec, times, xgrid, sol; Δt=1e-1, save=false)
    subplots = [(1,1),(1,2),(2,1),(2,2)]
    p = GridVisualizer(resolution=(700,700),dim=2,Plotter=PyPlot,layout=
(2,2),title="Solution with anisotropic conductivity")
    cnt = 0
    for (t,sp) ∈ zip(times,subplots)
        cnt += 1
        tₛ = Int16(round(t/Δt))+1
        scalarplot!(p[sp...],
            xgrid,sol[spec,:,tₛ], Plotter=PyPlot, colormap=:hot,
            title="t=$t", colorlevels=100, colorbar=false, framepos=cnt)
    end
    p.context[:title] = "Solution with anisotropic conductivity"
    save ? p : reveal(p)
end
```




```
contour_2d_at_t(2,3,Δt₂,xgrid₂,sol₂,
    "2D problem with 1D problem setup for "*species[2]*" at t="*string(3))
```

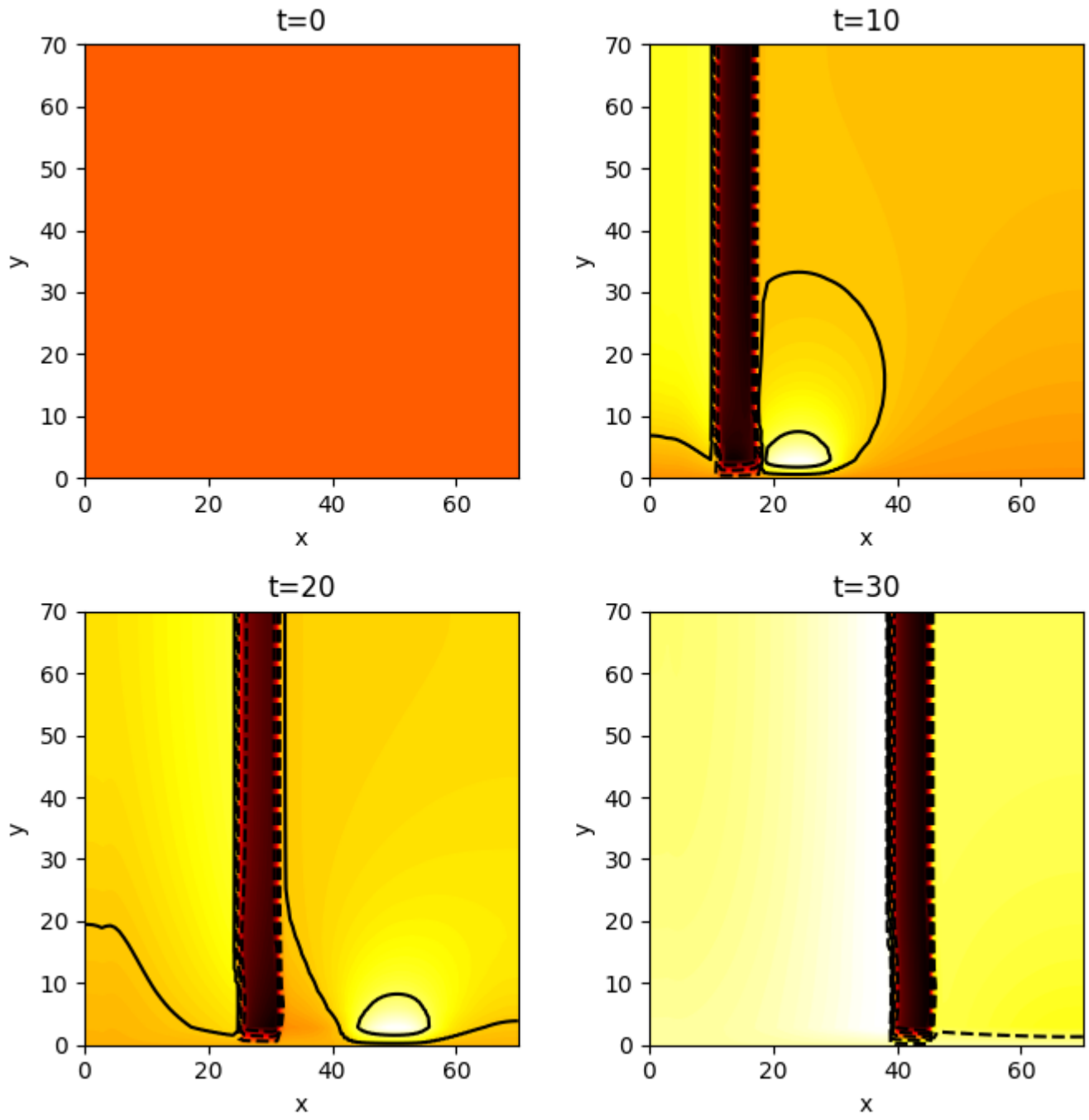
```
times = [0,10,20,30];
```

Solution to 2D problem with 1D problem setup for u:



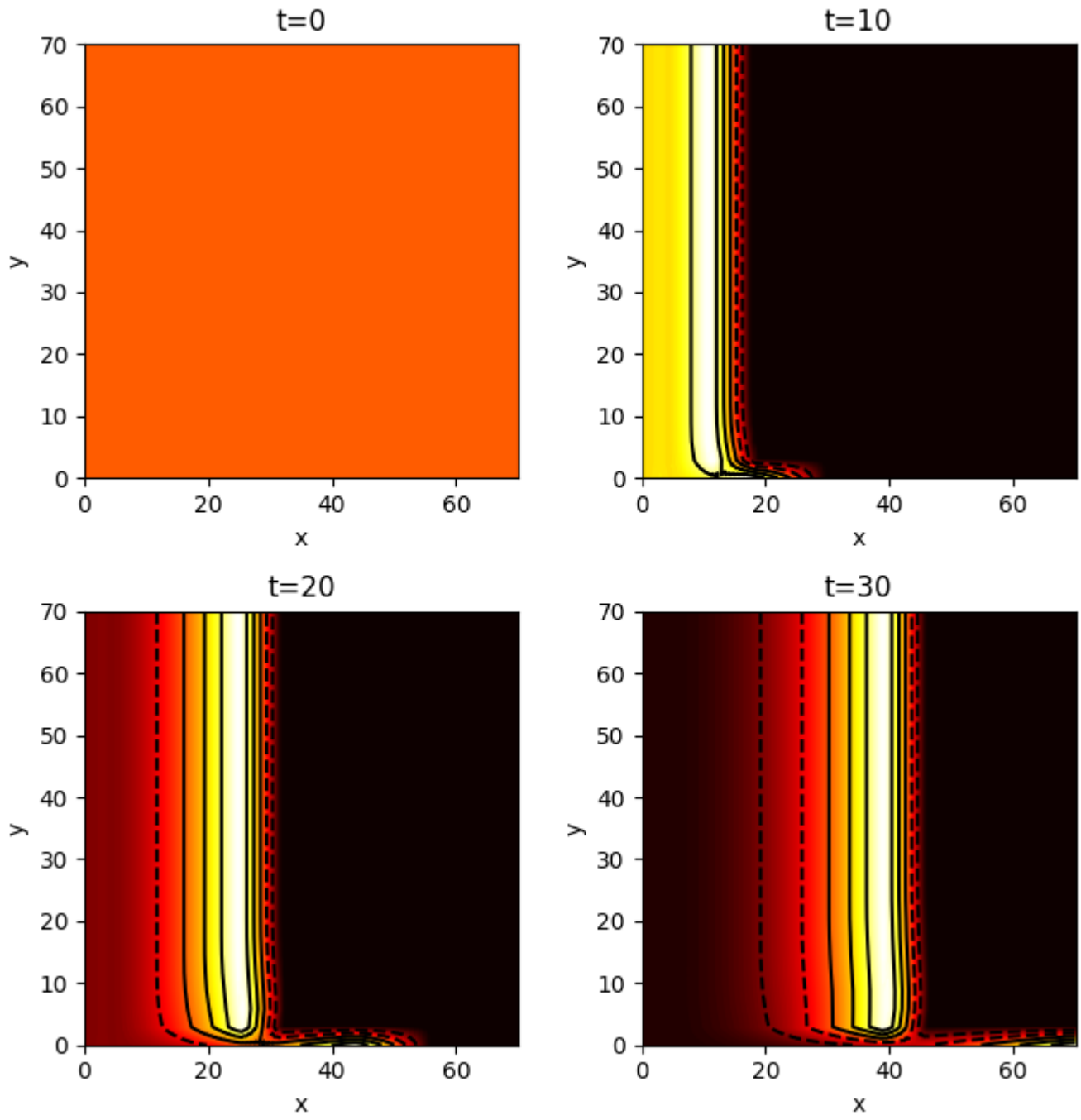

```
contour_subplots(1,times,xgrid₂,sol₂;Δt=Δt₂)
```

Solution to 2D problem with 1D problem setup for u_e:




```
contour_subplots(2,times,xgrid₂,sol₂;Δt=Δt₂)
```

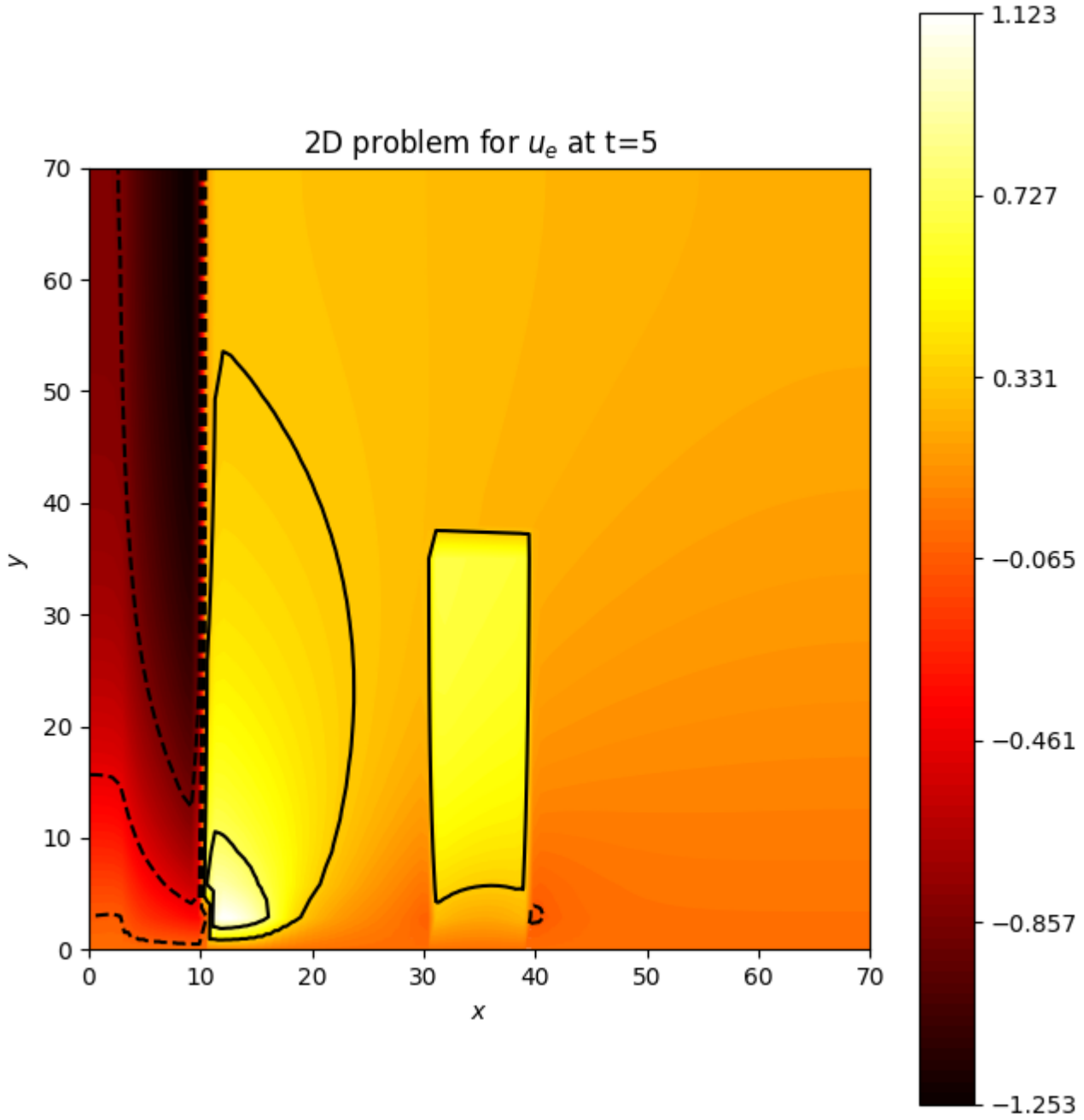
Solution to 2D problem with 1D problem setup for v:




```
contour_subplots(3,times,xgrid₂,sol₂;Δt=Δt₂)
```

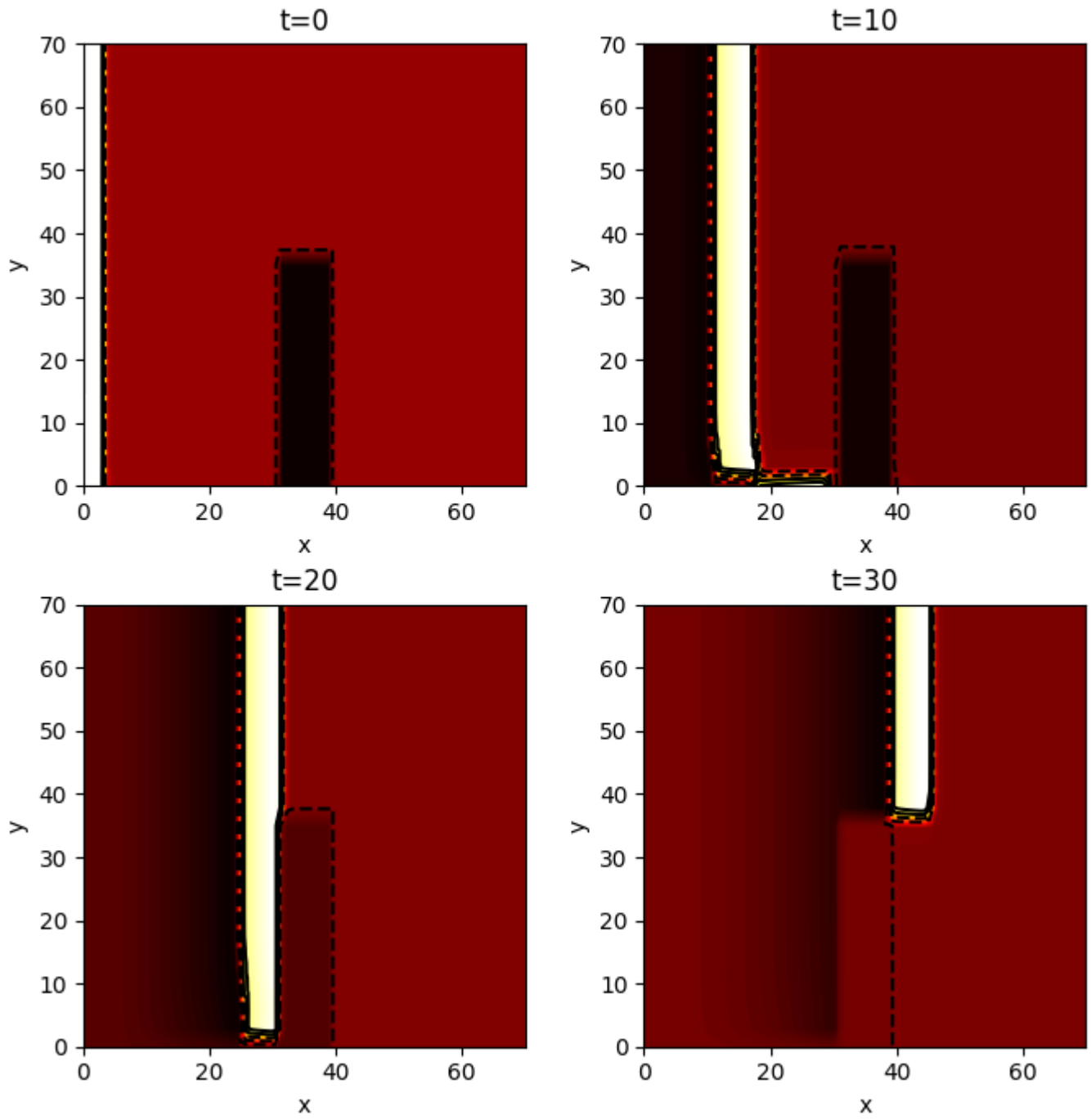
2D Problem

```
begin
    dim₃=2; N₃ = (100,25); Δt₃ = 1e-1;
    xgrid₃, tgrid₃, sol₃, vis₃ = bidomain(
        dim=dim₃, N=N₃, T=T, Δt=Δt₃, Plotter=PyPlot, initial2D=true, Tinit_solve=0)
end;
```



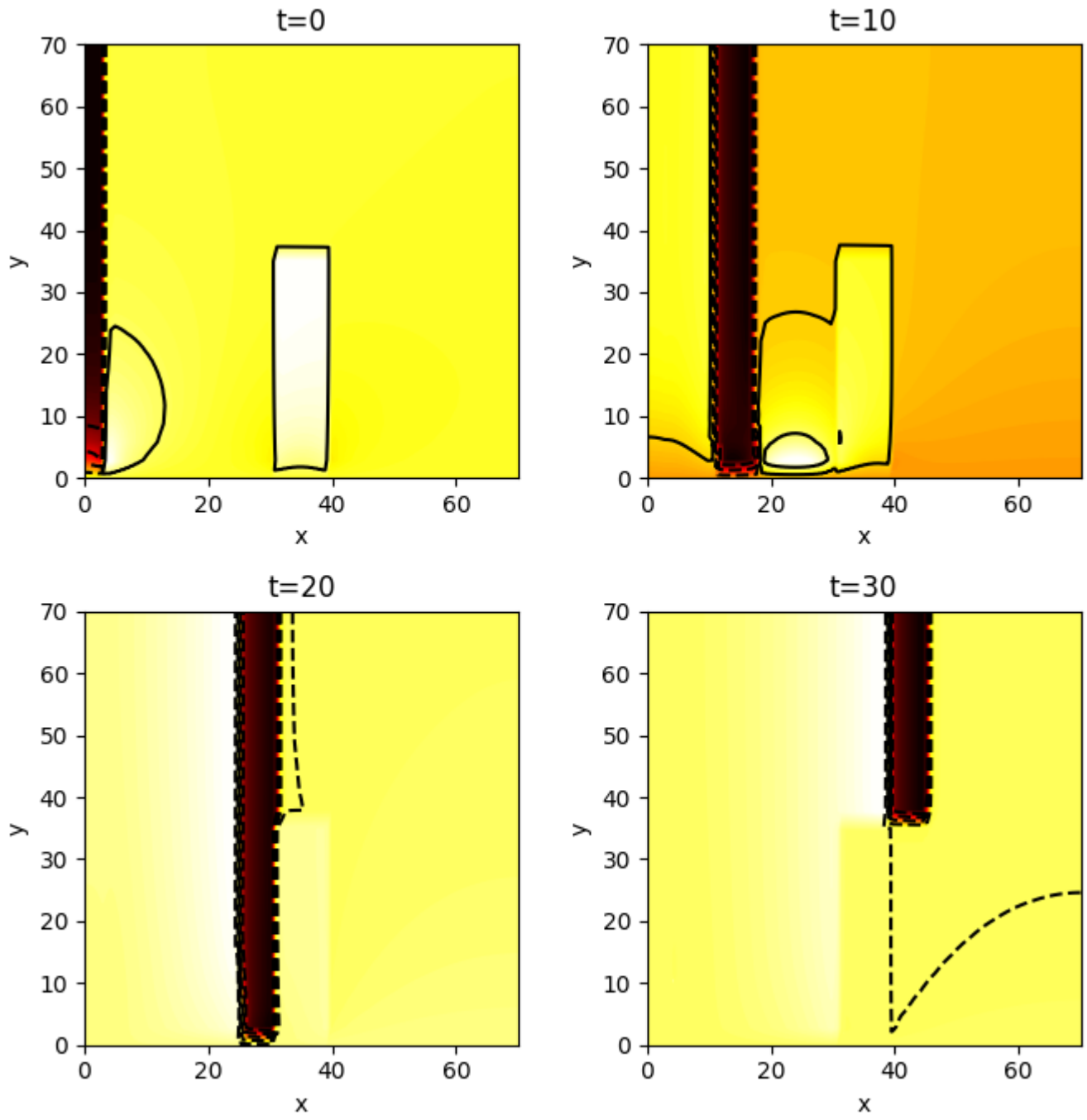

```
contour_2d_at_t(2,5,Δt₃,xgrid₃,sol₃,
    "2D problem for "*species[2]*" at t="*string(5))
```

Solution to 2D problem for u:



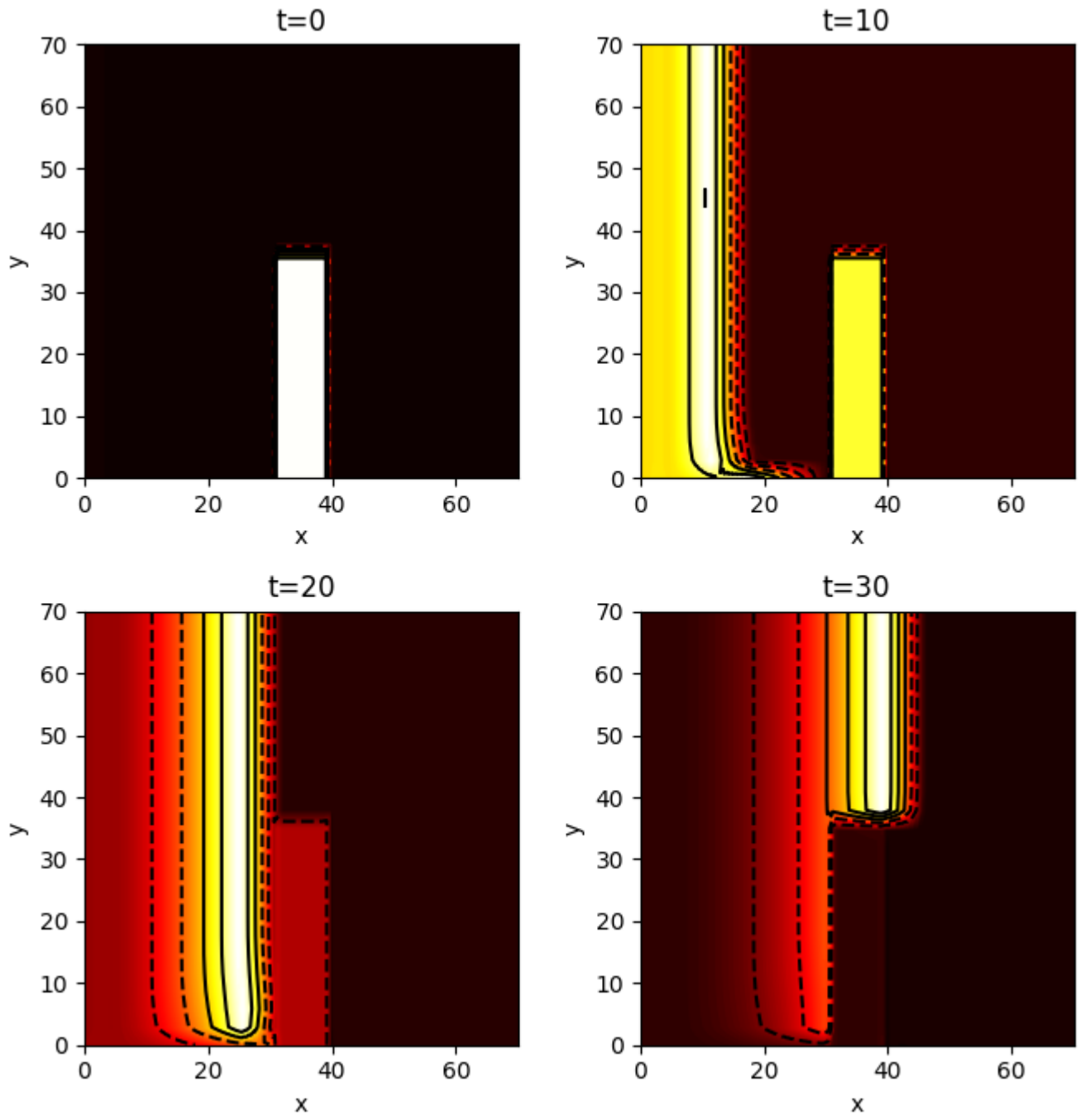

```
contour_subplots(1,times,xgrid₃,sol₃;Δt=Δt₃)
```

Solution to 2D problem with 1D problem setup for u_e:




```
contour_subplots(2,times,xgrid₃,sol₃;Δt=Δt₃)
```

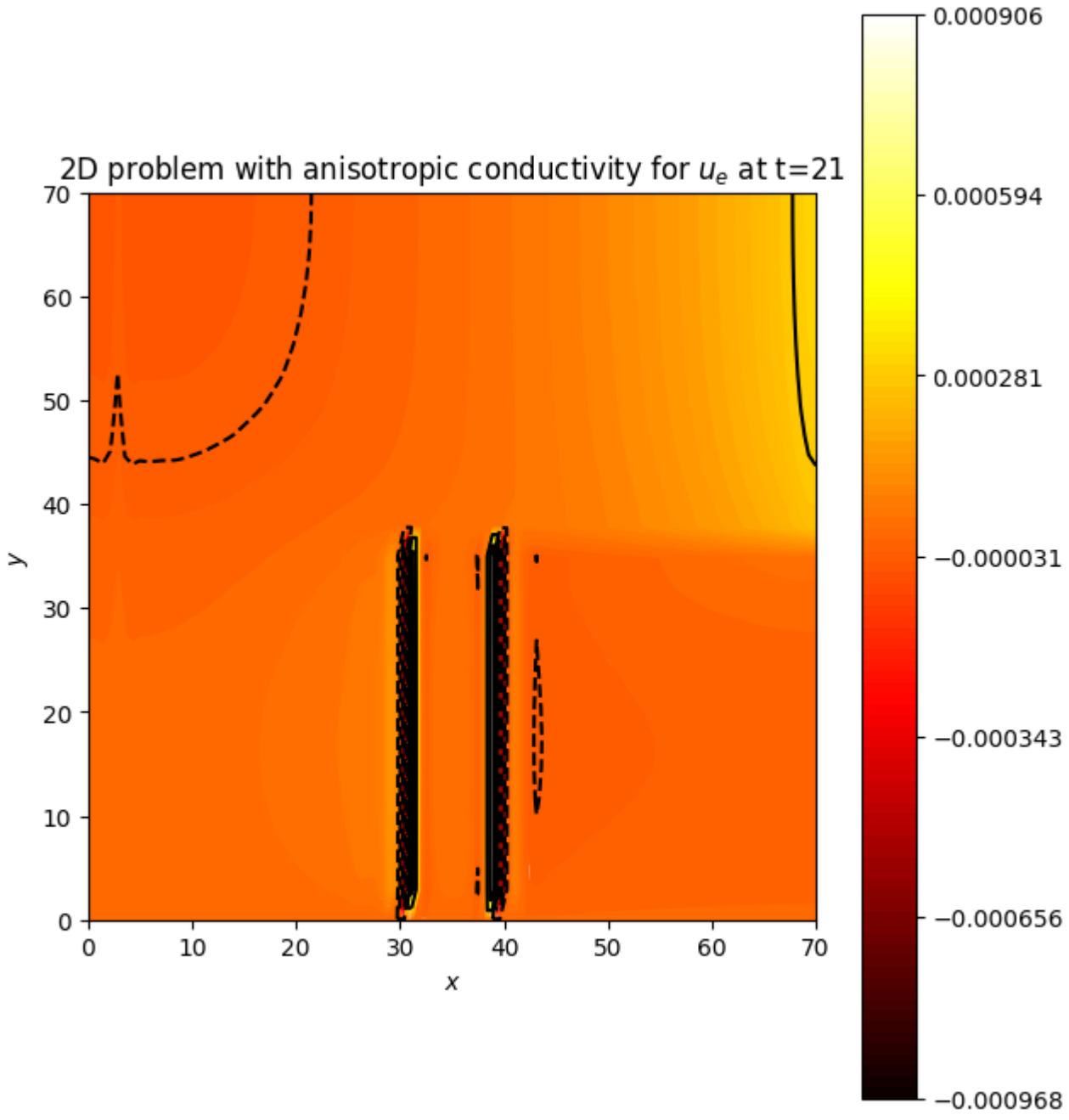
Solution to 2D problem for v:




```
contour_subplots(3,times,xgrid₃,sol₃;Δt=Δt₃)
```

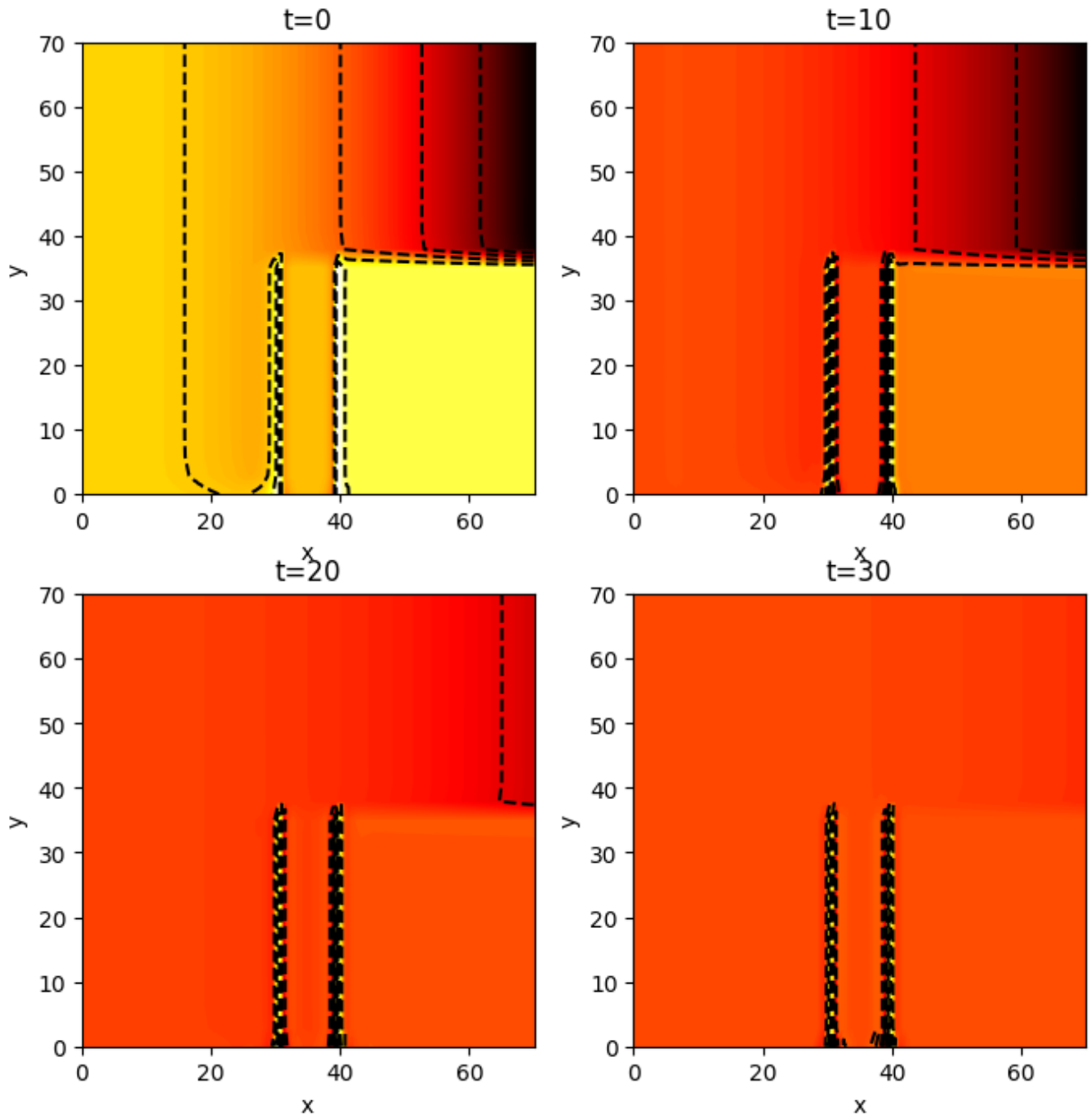
2D Problem with anisotropic conductivity

```
begin
    dim₄=2; N₄=(100,25); Δt₄=1e-1;
    xgrid₄, tgrid₄, sol₄, vis₄ = bidomain(
        dim=dim₄, N=N₄, T=T, Δt=Δt₄, Plotter=PyPlot, initial2D=true,
        anisotropic=true);
end;
```



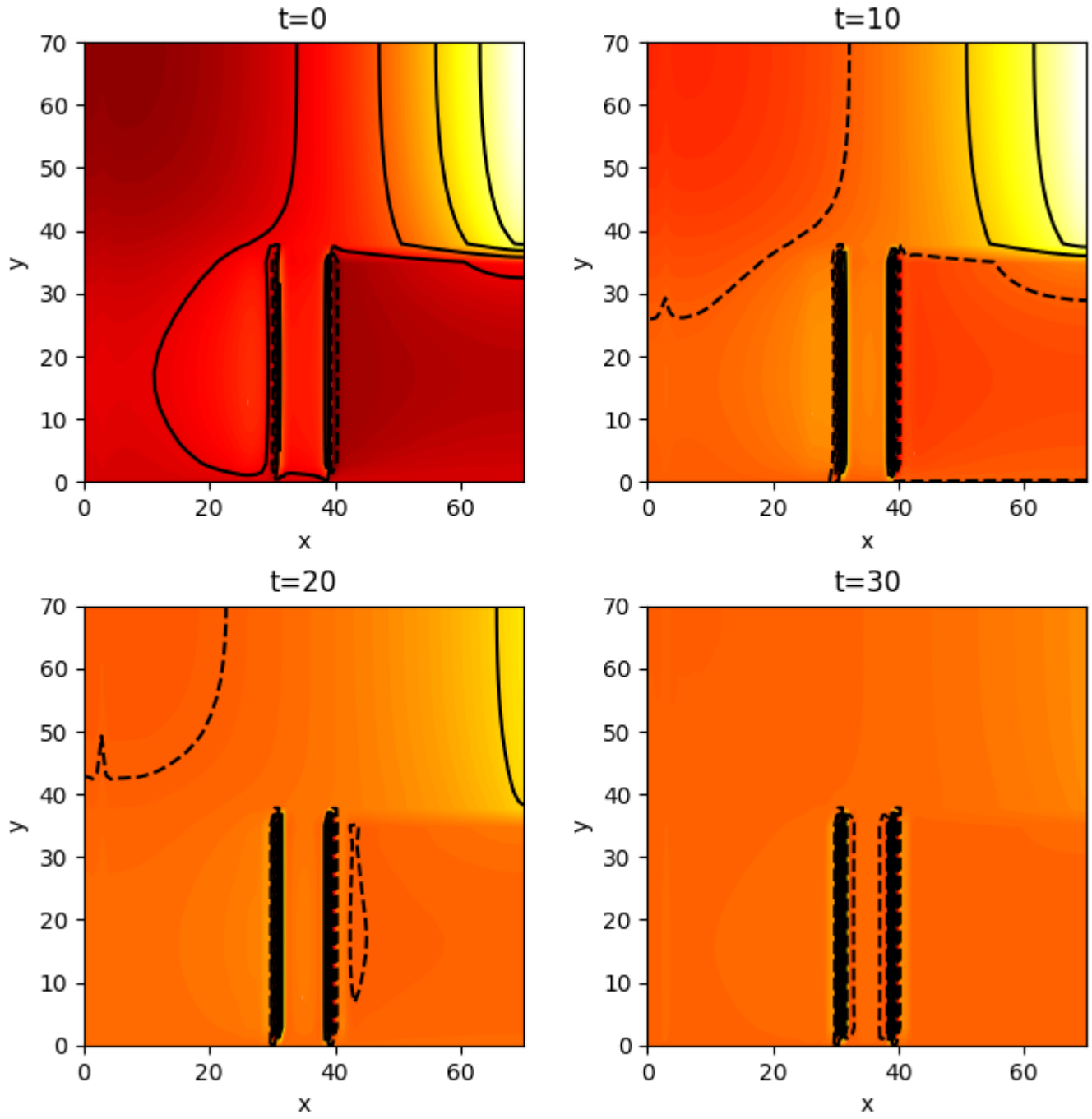

```
contour_2d_at_t(2,21,Δt₄,xgrid₄,sol₄,
    "2D problem with anisotropic conductivity for "*species[2]*" at t="*string(21))
```

Solution to 2D problem with anisotropic conductivity for u:



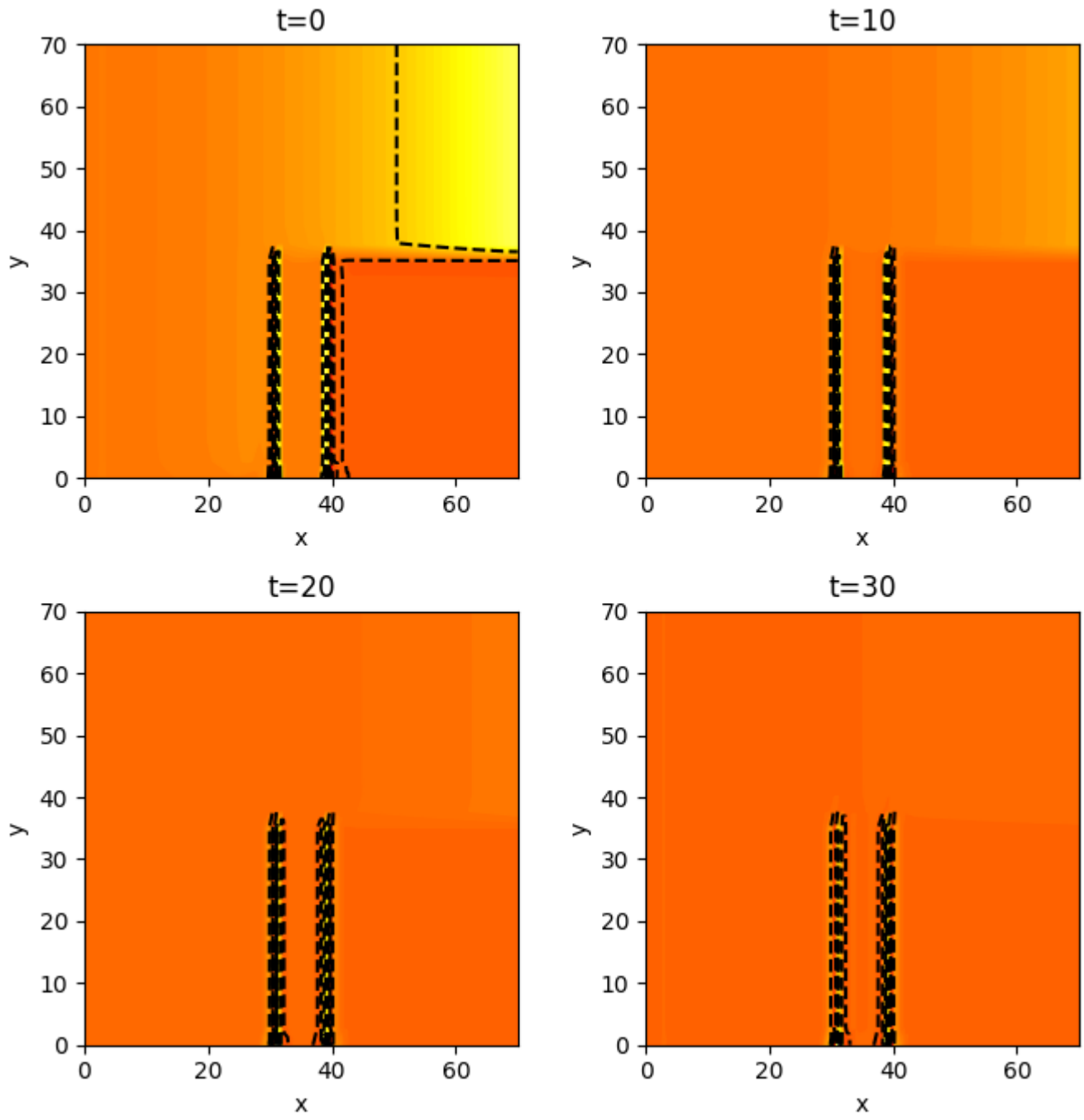

```
contour_subplots(1,times,xgrid₄,sol₄;Δt=Δt₄)
```

Solution to 2D problem with anisotropic conductivity for u_e:




```
contour_subplots(2, times, xgrid₄, sol₄; Δt=Δt₄)
```

Solution to 2D problem with anisotropic conductivity for v:




```
contour_subplots(3,times,xgrid₄,sol₄;Δt=Δt₄)
```

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

```
function save_all_subplots()
    plot_times = [0,10,20,30]
    grids = [xgrid₂,xgrid₃,xgrid₄]
    solutions = [sol₂,sol₃,sol₄]
    Δtₛ = [Δt₂,Δt₃,Δt₄]
    fnames = ["2D_with_1D_setup", "2D", "2D_with_anisotropic"]
    for i ∈ length(grids)
        grid = grids[i]; sol = solutions[i]; Δt = Δtₛ[i]; fname = fnames[i]
        for spec ∈ 1:3
            p = contour_subplots(spec, plot_times, grid, sol; Δt=Δt, save=true)
            GridVisualize.save("../img/"*fname*"_spec"*string(spec)*".png", p)
        end
    end
end
```

```
save_all_subplots()
```

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

```
function images_for_gif(spec, folder, xgrid, sol; steps=5)
    folder = "../img/"*folder*"_"*string(spec)*"/"
    mkpath(folder)

    grid_vis = GridVisualizer(resolution=(500,500), dim=2, Plotter=PyPlot)
    for ts=2:steps:size(sol)[3]
        scalarplot!(grid_vis,
            xgrid, sol[spec,:,ts], Plotter=PyPlot, colormap=:hot, colorlevels=100,
            colorbar=false)
        GridVisualize.save(folder*string(ts)*".png", grid_vis)
    end
end

```

```
for spec=1:3
    images_for_gif(spec, "contour_plot_species", xgrid₂, sol₂; steps=5)
    images_for_gif(spec, "contour_plot_2D_species", xgrid₃, sol₃; steps=5)
    images_for_gif(spec, "contour_plot_2Danisotropic_species", xgrid₄, sol₄; steps=5)
end

```

Problems

We have not gotten the anisotropic solution or the 2D problem setup to give the same solution as they have in the paper, we never figured out what the problem was. Possible problems includes setting the dirichlet boundary for u_e:

$$u_e(0,0) = 0$$

Performance improvement

To improve performance and stability, as an important first step, we could use more sophisticated time-stepping approaches.

In the Ethier and Borgoult paper, they show a number of schemes and their performance. From this, we can conclude that employing 2nd order methods is most effective. However, we also note that IMEX yields a linear system so each time step, so even if less stable the performance advantage could be very significant when properly implemented.

When it comes to parallelization, the ability to parallelize the solution of the problem at each time step would yield massive performance gains, as it amounts to performing a sequence of sparse matrix-vector multiplications. This is because solving the linear system that results in each iteration of the Newton method can be done by an iterative scheme such as CG, and these schemes also amount to performing a series of sparse matrix-vector multiplications.

Optional topics, Anisotropic conductivity

For the anisotropic conductivity we have different conductivity in each direction. So in the multiplication of σ_i and σ_e for the fluxes we need to get the contribution to the correct direction. We then need to get the angle between the two edges x_k and x_l:

$$\theta = \cos^{-1}\left(\frac{(x_k - x_l)\cdot e_x}{|x_k - x_l|}\right)$$

which we then take the corresponding values of the tensor, and we multiply the flux with $\sigma_{i,new}$, $\sigma_{e,new}$ which we get as

$$\sigma_{new} = \sigma_{1,1}\cos(\theta) = \sigma_{2,2}\sin(\theta).$$

References

1. Fuhrmann, Jurgen *VoronoiFVM.jl: Finite Volume Solver For Coupled Nonlinear Partial Differential Equations*, Zenodo (2022) 10/gpgb9n, cited by 0
2. Hooke, N.; Henriquez, C.S.; Lanzkron, P.; Rose, D. *Linear Algebraic Transformations Of The Bidomain Equations: Implications For Numerical Methods*, Mathematical Biosciences (1994) 10/cxq6cb, cited by 45
3. Ethier, Marc; Bourgault, Yves *Semi-Implicit Time-Discretization Schemes For The Bidomain Model*, Siam Journal On Numerical Analysis (2008) 10/cq2xvr, cited by 72