CahnHilliardSBM.jl

A Case Study of Solving a Stiff, Nonlinear PDE in Custom Geometries using the Smoothed Boundary Method (SBM)

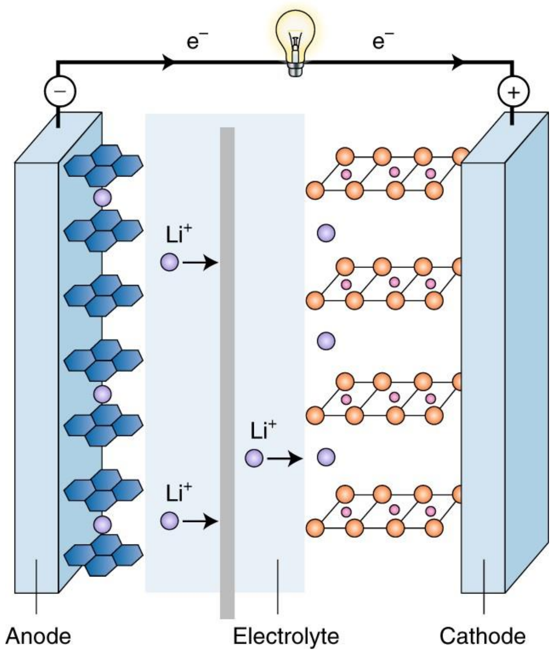
Samuel Degnan-Morgenstern (05/08/2023)

Background

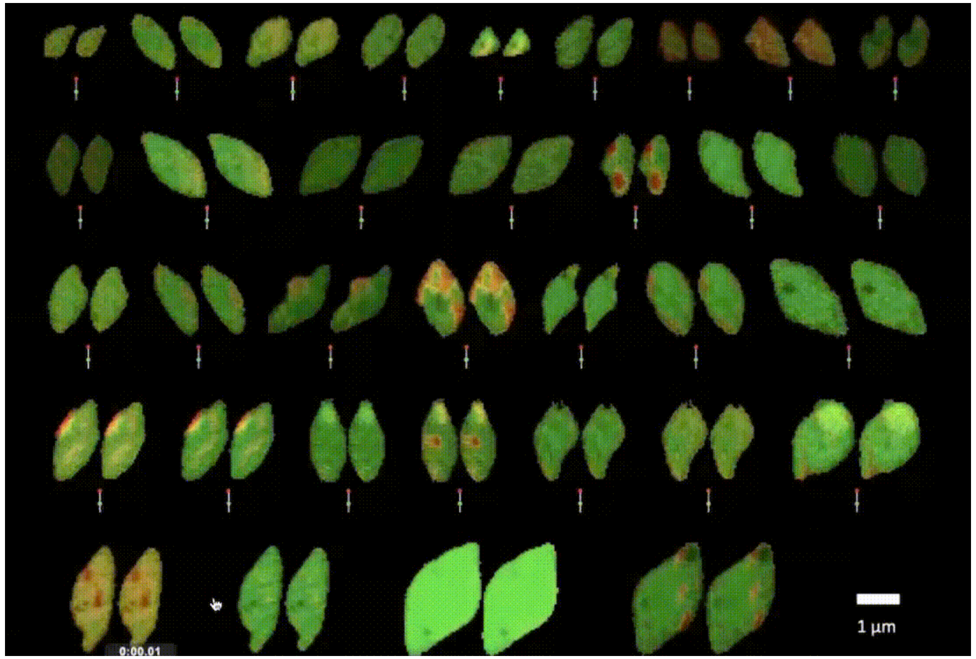
Motivation

Common lithium ion battery electrode materials exhibit complex ,heterogeneous physics characterized by phase separation

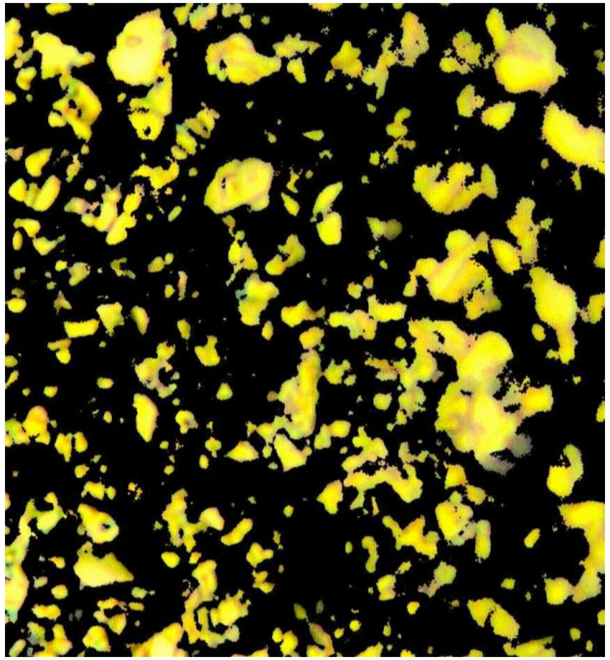
Past



Present



Future



Goodenough, Nat. Electron., 2018

Zhao et al., Preprint, 2022

A Crash Course in Non-Equilibrium Thermo

Mass Conservation & Linear Irreversible Thermodynamics

$$\frac{\partial c}{dt} = -\nabla \cdot F + \cancel{R_v}$$

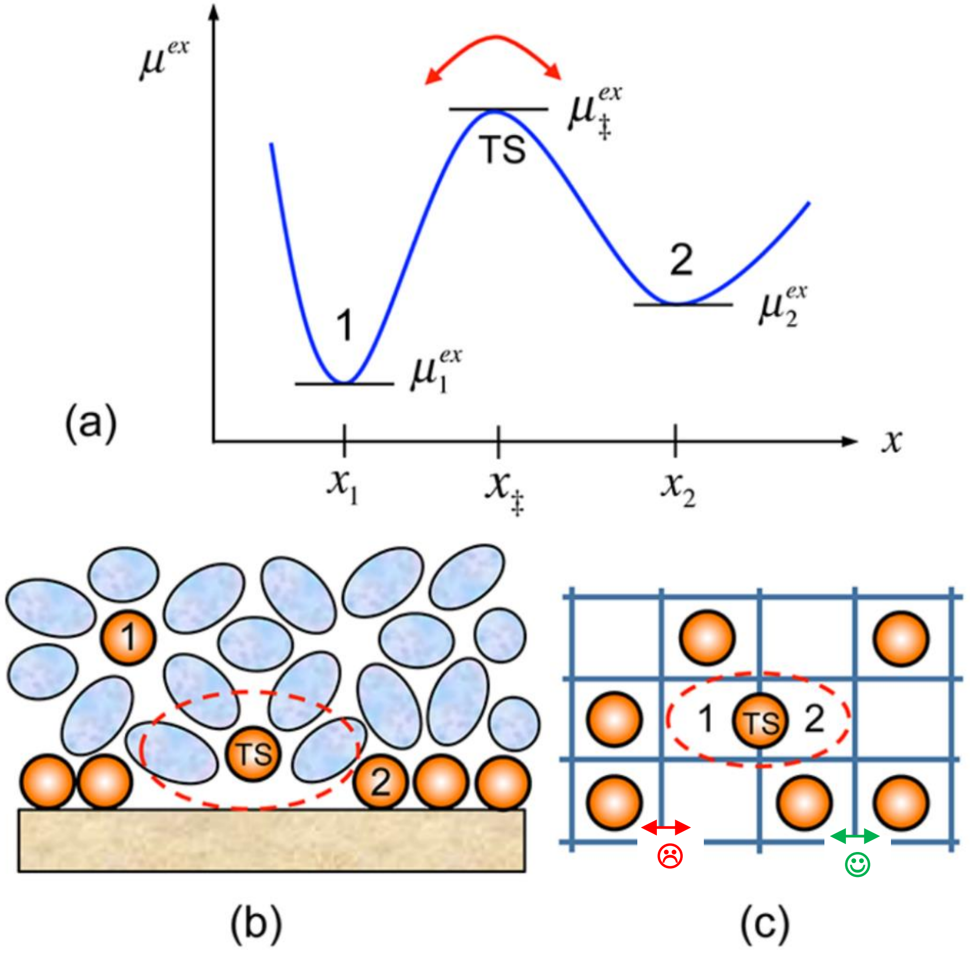
$$F = -D(c) \cdot \nabla \left(\frac{\delta G}{\delta c} \right)$$

$$\frac{\delta G}{\delta c} = \mu = \mu_h - \kappa \nabla^2 c$$

Cahn–Hilliard Partial Differential Equation

$$\frac{\partial c}{dt} = \nabla \cdot (D(c) \cdot \nabla \mu)$$

$$\mu = \underbrace{\log\left(\frac{c}{1-c}\right) + \Omega\,(1-2c)}_{\text{Regular Solution}} - \kappa \nabla^2 c$$



Bazant, ACR, 2013

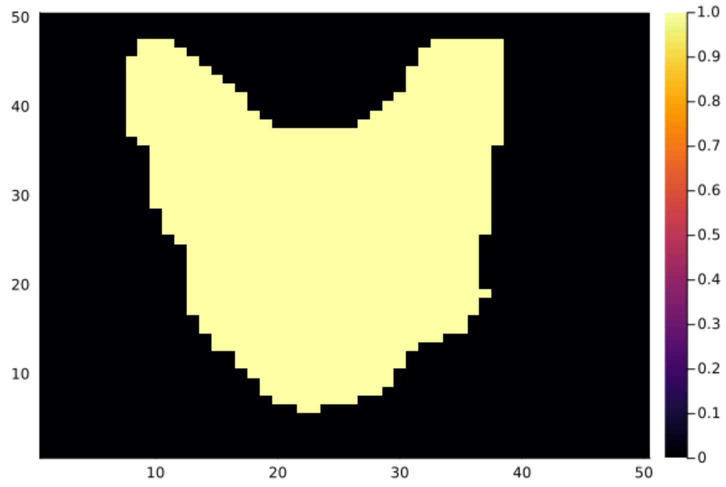
Smoothed Boundary Method

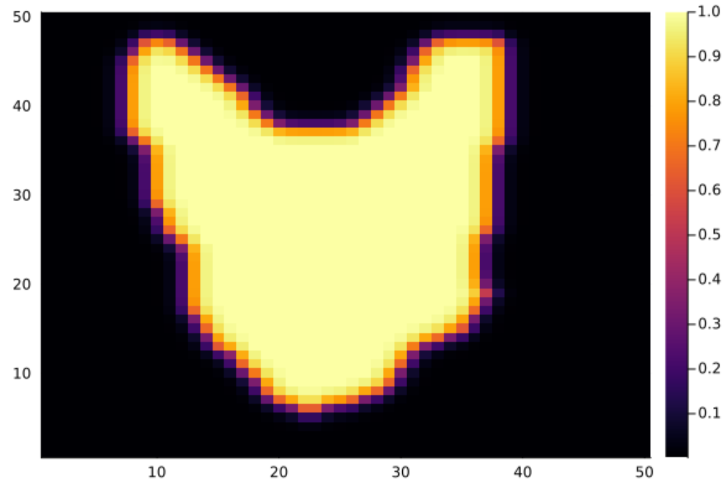
$$\psi(x,y)\frac{\partial c}{dt} = -\psi(x,y)\nabla\cdot(D(c)\cdot\nabla\mu)$$

$$\psi(x,y) = \begin{cases} 1 & \text{if } (x,y)\in\mathbf{S} \\ \approx 0.5 & \text{if } (x,y)\in\partial\mathbf{S} \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\partial c}{dt} = \frac{D(c)}{\psi}\nabla\psi\cdot\nabla\mu + \frac{\partial D}{\partial c}\nabla c\cdot\nabla\mu + D(c)\nabla^2\mu$$

$$\mu = \log\left(\frac{c}{1-c}\right) + \Omega\,(1-2c) - \kappa\left(\frac{\nabla\psi\cdot\nabla c}{\psi} + \nabla^2 c\right)$$





Smoothed Boundary Method

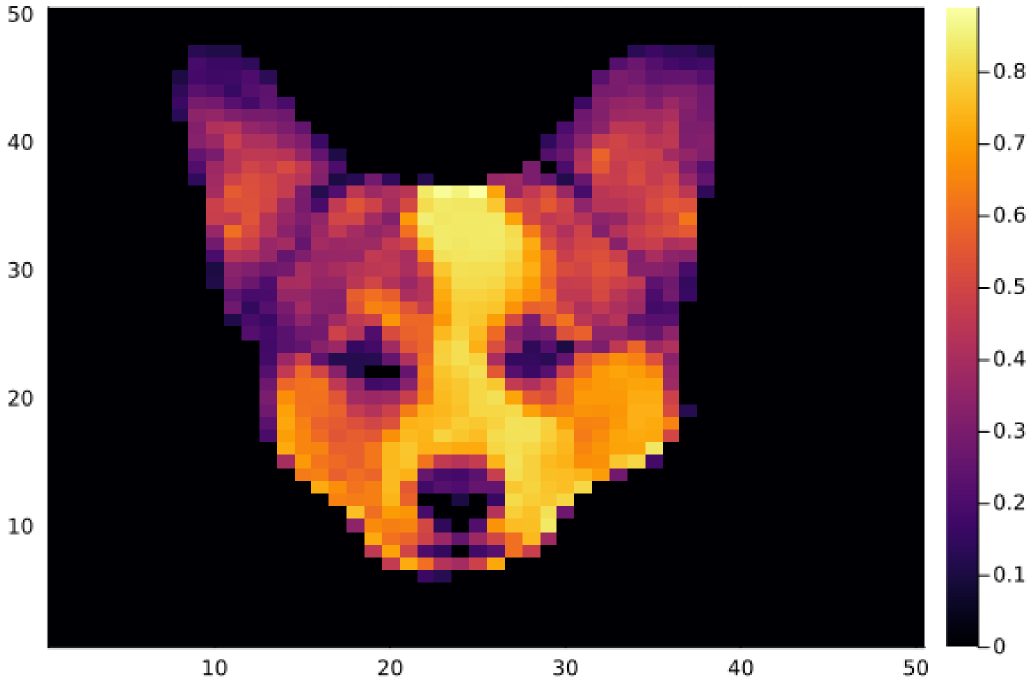
$$\psi(x,y)\frac{\partial c}{dt} = -\psi(x,y)\nabla\cdot\left(D(c)\cdot\nabla\mu\right)$$

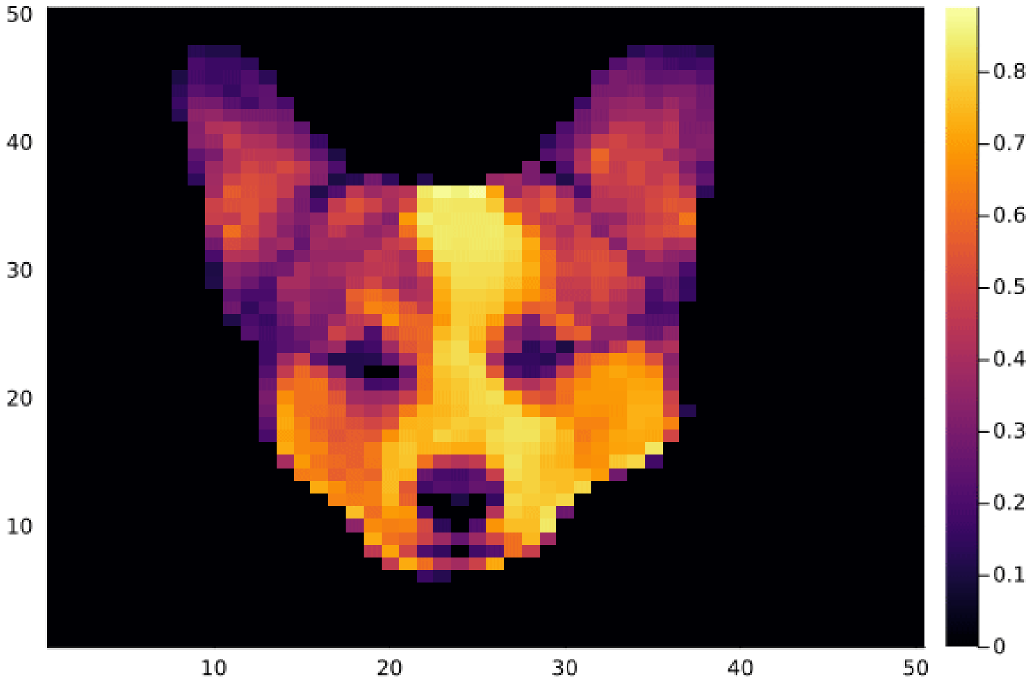
$$\psi(x,y) = \begin{cases} 1 & \text{if } (x,y)\in\mathbf{S} \\ \approx 0.5 & \text{if } (x,y)\in\partial\mathbf{S} \\ 0 & \text{otherwise} \end{cases}$$

$$\longrightarrow$$

$$\frac{\partial c}{dt} = \frac{D(c)}{\psi}\nabla\psi\cdot\nabla\mu + \frac{\partial D}{\partial c}\nabla c\cdot\nabla\mu + D(c)\nabla^2\mu$$

$$\mu = \log\left(\frac{c}{1-c}\right) + \Omega\left(1-2c\right) - \kappa\left(\frac{\nabla\psi\cdot\nabla c}{\psi} + \nabla^2 c\right)$$





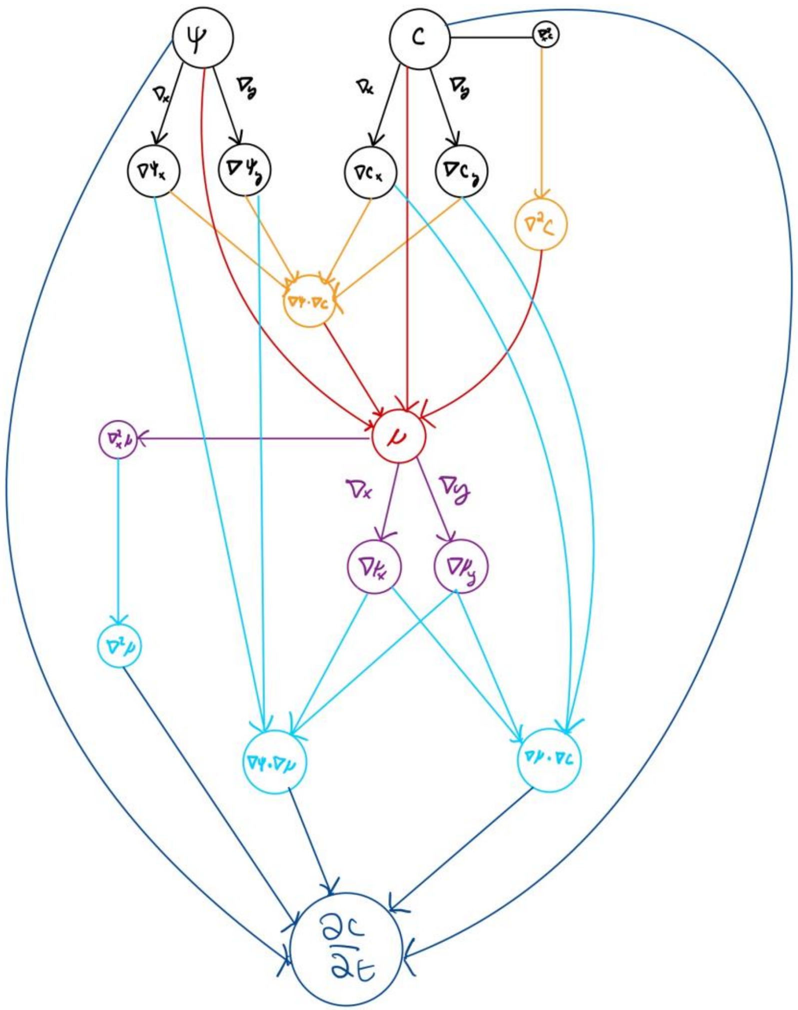
**** No corgis harmed in the making of this project*

Numerical Implementation

**** All simulations run on Windows 11 Machine with AMD Ryzen 9 7950 16 Core / 32 threads 4.5GHz CPU , NVIDIA RTX 4070 TI GPU*

A Naïve Implementation

```julia
function GCH_2D_mask(dc,c,p,t)
    D, κ, Ω,dx,dy,Nx,Ny,Ψ = p
    ψ = @view Ψ[:,:]
    @inline function ∇2c(ix,iy)
        left = ix > 1 ? c[ix-1,iy] : c[ix+1,iy]
        right = ix < Nx ? c[ix+1,iy] : c[ix-1,iy]
        bottom = iy > 1 ? c[ix,iy-1] : c[ix,iy+1]
        top = (iy < Ny ? c[ix,iy+1] : c[ix,iy-1])
        return ((right + left - 2.0*c[ix,iy])/dx^2 + (top + bottom - 2.0*c[ix,iy])/dy^2)
    end
    @inline function ∇ψ∇c(ix,iy)
        ψleft = ix > 1 ? ψ[ix-1,iy] : ψ[ix+1,iy]
        ψright = ix < Nx ? ψ[ix+1,iy] : ψ[ix-1,iy]
        ψbottom = iy > 1 ? ψ[ix,iy-1] : ψ[ix,iy+1]
        ψtop = iy < Ny ? ψ[ix,iy+1] : ψ[ix,iy-1]

        cleft = ix > 1 ? c[ix-1,iy] : c[ix+1,iy]
        cright = ix < Nx ? c[ix+1,iy] : c[ix-1,iy]
        cbottom = iy > 1 ? c[ix,iy-1] : c[ix,iy+1]
        ctop = iy < Ny ? c[ix,iy+1] : c[ix,iy-1]

        return ((ψleft-ψright)/(2*dx))*((cleft-cright)/(2*dx)) + ((ψtop-ψbottom)/(2*dx))*((ctop-cbottom)/(2*dy))
    end
    @inline function μₕ(ix,iy)
        return log(max(1e-10,c[ix,iy]/(1-c[ix,iy]))) + Ω*(1.0-2.0*c[ix,iy])
    end
    @inline function μ(ix,iy)
        return μₕ(ix,iy) - κ*(∇ψ∇c(ix,iy)/ψ[ix,iy] + ∇2c(ix,iy))
    end
    @inline function ∇ψ∇μ(ix,iy)
        ψleft = ix > 1 ? ψ[ix-1,iy] : ψ[ix+1,iy]
        ψright = ix < Nx ? ψ[ix+1,iy] : ψ[ix-1,iy]
        ψbottom = iy > 1 ? ψ[ix,iy-1] : ψ[ix,iy+1]
        ψtop = iy < Ny ? ψ[ix,iy+1] : ψ[ix,iy-1]

        μleft = ix > 1 ? μ(ix-1,iy) : μ(ix+1,iy)
        μright = ix < Nx ? μ(ix+1,iy) : μ(ix-1,iy)
        μbottom = iy > 1 ? μ(ix,iy-1) : μ(ix,iy+1)
        μtop = iy < Ny ? μ(ix,iy+1) : μ(ix,iy-1)

        return ((ψleft-ψright)/(2*dx))*((μleft-μright)/(2*dx)) + ((ψtop-ψbottom)/(2*dx))*((μtop-μbottom)/(2*dy))
    end
     @inline function ∇c∇μ(ix,iy)
         cleft = ix > 1 ? c[ix-1,iy] : c[ix+1,iy]
         cright = ix < Nx ? c[ix+1,iy] : c[ix-1,iy]
         cbottom = iy > 1 ? c[ix,iy-1] : c[ix,iy+1]
         ctop = iy < Ny ? c[ix,iy+1] : c[ix,iy-1]

         μleft = ix > 1 ? μ(ix-1,iy) : μ(ix+1,iy)
         μright = ix < Nx ? μ(ix+1,iy) : μ(ix-1,iy)
         μbottom = iy > 1 ? μ(ix,iy-1) : μ(ix,iy+1)
         μtop = iy < Ny ? μ(ix,iy+1) : μ(ix,iy-1)

         return ((cleft-cright)/(2*dx))*((μleft-μright)/(2*dx)) + ((ctop-cbottom)/(2*dx))*((μtop-μbottom)/(2*dy))
     end
     @inline function ∇2μ(ix,iy)
         left = ix > 1 ? μ(ix-1,iy) : μ(ix+1,iy)
         right = ix < Nx ? μ(ix+1,iy) : μ(ix-1,iy)
         bottom = iy > 1 ? μ(ix,iy-1) : μ(ix,iy+1)
         top = iy < Ny ? μ(ix,iy+1) : μ(ix,iy-1)
         return ((right + left - 2.0*μ(ix,iy))/dx^2 + (top + bottom - 2.0*μ(ix,iy))/dy^2)
     end
     @inline function getD(ix::Int,iy::Int)
         return D*(1.0-c[ix,iy])*c[ix,iy];
     end
     @inline function ∂D∂c(ix,iy)
         return D*(1.0-2*c[ix,iy]);
     end
     @inline function normψ(ix,iy)
         if ((ix > 1) && (ix < Nx)) && ((iy > 1) && (iy < Ny))
             return sqrt(((c[ix+1,iy]-c[ix-1,iy])/(2*dx))^2 + ((c[ix,iy+1]-c[ix,iy-1])/(2*dx))^2)
         else
             return 0.0
         end
     end
     @inbounds @views for I in CartesianIndices((Nx, Ny))
         ix, iy = Tuple(I);
         dc[ix,iy]=(getD(ix,iy)/ψ[ix,iy])*∇ψ∇μ(ix,iy)+ ∂D∂c(ix,iy)*∇c∇μ(ix,iy) + getD(ix,iy)*∇2μ(ix,iy)
     end
     return nothing
 end
```



A Naïve Implementation

40 x 40 system, t ϵ (0,5)

```
@benchmark sol_l=solve($prob,CVODE_BDF(linear_solver=:GMRES),save_everystep=false)
✓ 14.8s

BenchmarkTools.Trial: 9 samples with 1 evaluation.
 Range (min … max):  594.319 ms … 610.479 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     602.146 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   601.627 ms ±   5.963 ms  ┊ GC (mean ± σ):  0.00% ± 0.00%

 594 ms           Histogram: frequency by time          610 ms <

 Memory estimate: 768.41 KiB, allocs estimate: 15489.
```

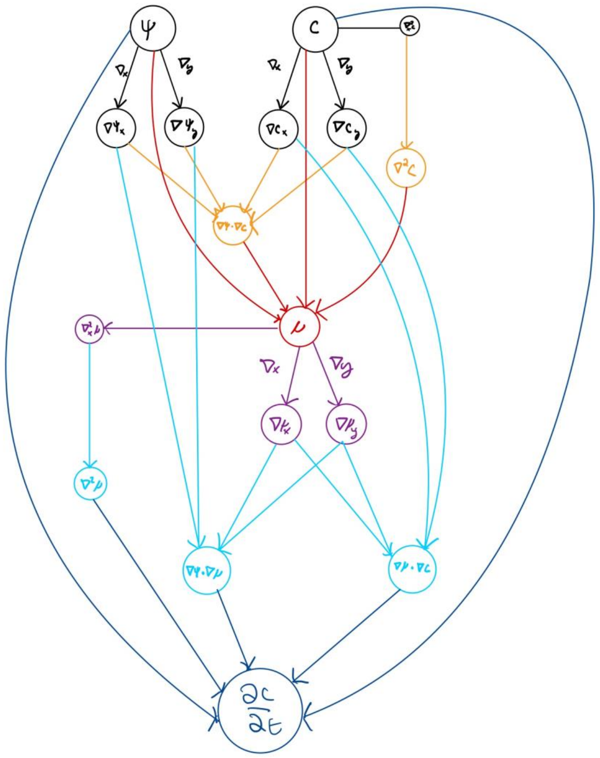
We can do better!

Several Iterations of Code Optimization…

```julia
function GCH_2D_mul_full2(du, u, p, t,ψ,∇x,∇y,∇2x,∇2y,∇ψ_x,∇ψ_y,∇c_x,∇c_y,∇2c,μ,∇2μ,∇μ_x,∇μ_y)
    D, κ, Ω =p
    c = @view u[:,:]
    dc = @view du[:,:]

    #Set up caches from DiffCache
    ∇c_x_t = get_tmp(∇c_x,u)
    ∇c_y_t = get_tmp(∇c_y,u)
    ∇2c_t = get_tmp(∇2c,u)
    μ_t = get_tmp(μ,u)
    ∇2μ_t = get_tmp(∇2μ,u)
    ∇μ_x_t = get_tmp(∇μ_x,u)
    ∇μ_y_t = get_tmp(∇μ_y,u)

    #Compute ∇c
    mul!(∇c_x_t,∇x,c) # Compute (∇c)ₓ = ∇x*c
    mul!(∇c_y_t,c,∇y) # Compute (∇c)_y = c*∇y

    #Compute ∇2c
    mul!(∇2c_t,∇2x,c) # Compute (∇2c)ₓ = c*∇2x
    mul!(∇2c_t,c,∇2y,1.0,1.0) #∇2c = 1*(∇2c)ₓ + 1*(∇2y)*c

    @. μ_t = log(max(1e-10,c./(1.0 - c)))+ Ω*(1.0 - 2.0*c) .- κ*((∇c_x_t*∇ψ_x  + ∇c_y_t*∇ψ_y)./ψ + ∇2c_t);

    #Compute ∇2μ
    mul!(∇2μ_t,∇2x,μ_t) # Compute (∇2μ)ₓ = μ*∇2x
    mul!(∇2μ_t,μ_t,∇2y,1.0,1.0) #∇2μ = 1*(∇2μ)ₓ + 1*(∇2y)*μ
    #Compute ∇μ
    mul!(∇μ_x_t,∇x,μ_t) # Compute (∇μ)ₓ = ∇x*μ
    mul!(∇μ_y_t,μ_t,∇y) # Compute (∇μ)_y = μ*∇y
    @. dc = D*(c*(1.0-c)*((∇ψ_x*∇μ_x_t + ∇ψ_y*∇μ_y_t)./ψ + ∇2μ_t) + (1.0-2.0*c)*(∇c_x_t*∇μ_x_t + ∇c_y_t*∇μ_y_t))
    return nothing
end
```

Key Differences:

- Finite differencing redone with matrix stencil operators
- ForwardDiff.jl compatible mul! caches
- Efficient use of broadcasting



Several Iterations of Code Optimization…

Dense Jacobian:

```
@benchmark sol=solve($prob,ROCK2(),save_everystep=false)
✓ 13.5s
BenchmarkTools.Trial: 42 samples with 1 evaluation.
 Range (min … max):  119.738 ms … 121.523 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     119.959 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   120.169 ms ± 465.521 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%

 120 ms          Histogram: frequency by time          122 ms <

 Memory estimate: 594.03 KiB, allocs estimate: 11767.
```

```
@benchmark sol=solve($prob,CVODE_BDF(linear_solver=:GMRES),save_everystep=false)
✓ 13.8s
BenchmarkTools.Trial: 68 samples with 1 evaluation.
 Range (min … max):  73.467 ms … 75.297 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     74.370 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   74.422 ms ± 325.800 μs ┊ GC (mean ± σ):  0.00% ± 0.00%

 73.5 ms          Histogram: frequency by time          75.2 ms <

 Memory estimate: 915.22 KiB, allocs estimate: 20201.
```

Sparse Jacobian:

```
@benchmark solve($sparse_prob,ROCK2(),save_everystep=false)
✓ 13.0s
BenchmarkTools.Trial: 167 samples with 1 evaluation.
 Range (min … max):  29.764 ms … 30.778 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     29.984 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   30.029 ms ± 142.859 μs ┊ GC (mean ± σ):  0.00% ± 0.00%

 29.8 ms          Histogram: frequency by time          30.7 ms <

 Memory estimate: 645.19 KiB, allocs estimate: 2990.
```

```
@benchmark solve($sparse_prob,CVODE_BDF(linear_solver=:GMRES),save_everystep=false)
✓ 13.3s
BenchmarkTools.Trial: 254 samples with 1 evaluation.
 Range (min … max):  19.363 ms … 20.678 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     19.661 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   19.680 ms ± 109.976 μs ┊ GC (mean ± σ):  0.00% ± 0.00%

 19.4 ms          Histogram: frequency by time          20 ms <

 Memory estimate: 625.94 KiB, allocs estimate: 5378.
```

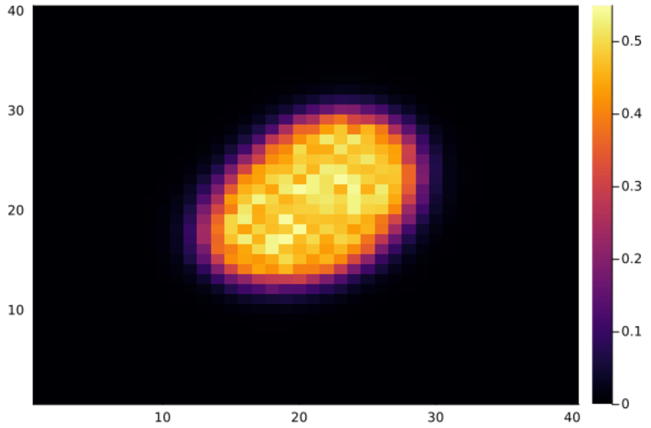
30x Speedup!!

SciML Tooling

Parameter Estimation via ForwardDiff

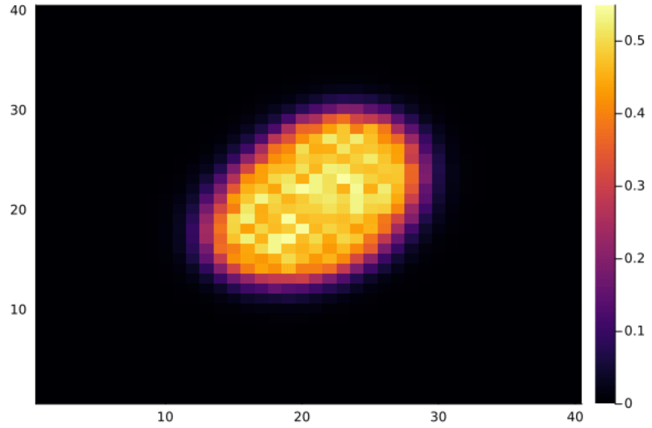
```julia
function proto_loss(θ,prob,tsteps,ode_data,ψ_binary)
    tmp_prob = remake(prob, p = θ)
    tmp_sol = solve(tmp_prob, TRBDF2(), saveat =tsteps,sensealg =ForwardDiffSensitivity())
    #if tmp_sol.retcode == ReturnCode.Success
    if size(tmp_sol) == size(ode_data)
        return sum(abs2, (ψ_binary.*(Array(tmp_sol) - ode_data)))
    else
        return Inf
    end
end

function set_pe(ψ,c0_messy,ptruth,tspan,Nsteps; tcleaning=1e-4)
    tsteps = collect(range(tspan[1], tspan[2], length = Nsteps))
    x,y,rhsfunc =setup_CH(ψ; gpuflag = false,levels=3)
    prob = makesparseprob(rhsfunc,c0,(0,tcleaning),ptruth)
    tmpsol=solve(prob,TRBDF2(),save_everystep=false);
    newc0 =tmpsol.u[end];
    prob = remake(prob,tspan=tspan,c0=newc0);
    ode_data = Array(solve(prob, TRBDF2(), saveat = tsteps));
    return  tsteps,ode_data,prob
end
function callback(p, l)
    global  iter
    iter += 1
    display("Iteration $(iter), loss = $(l)")
    return false
end
function solve_pe(pinit,loss;iter_max=200)
    optfun = OptimizationFunction((u,_)->loss(u), Optimization.AutoForwardDiff())
    optprob = OptimizationProblem(optfun, pinit)
    @time optsol = solve(optprob, Optim.NewtonTrustRegion(),callback=callback;maxiters=iter_max)
    return optsol
end


function run_pe(ψ,ψ_binary,c0_messy,ptruth;tspan=(0.0,1.0),Nsteps=100,itmax=200)
    tsteps,ode_data,prob=set_pe(ψ,c0_messy,ptruth,tspan,Nsteps)
    loss(θ) = proto_loss(θ,prob,tsteps,ode_data,ψ_binary)
    optsol=solve_pe(pinit,loss;iter_max=itmax)
    return optsol
end
```

Initial Guess

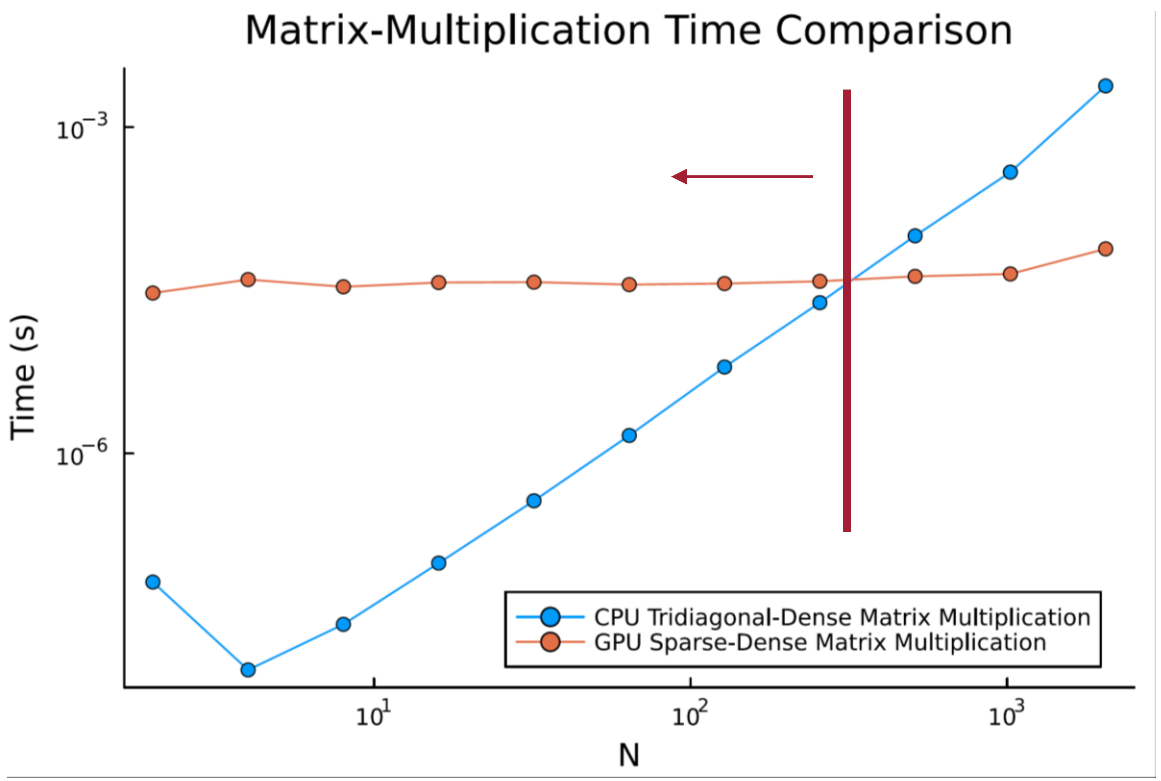


NewtonTrustRegion()   37 iterations

Recovers true solution



Within Method GPU Parallelization



To successfully implement GPU parallelization:

- Ease numerical instability
  - Larger systems lead to exploding Laplacian terms
- Move the needle to the left by writing custom GPU kernel

Questions? Collaboration?

- Opportunity to demonstrate capability of Julia SciML Ecosystem on a very complex physical problem
- Hoping to set up support for faster reverse mode AD
- Looking to improve performance & stability of within method GPU parallelization