Bachelor Thesis

Jonathan Ulmer

May 27, 2024

Contents

1 The Cahn-Hilliard equation **1**

2 Baseline multi-grid solver **4**

3 Numerical evaluation **13**

4 Relaxed problem **18**

5 AI **29**

6 References **31**

This thesis follows reproducible research philosophy, in that we provide all relevant code in the same file as the writing itself. We then use this file to generate exports to html and PDF, as well as extract the code to be used independently. Further details on execution and reading of the original source provided in org-mode format,

1 The Cahn-Hilliard equation

The Cahn-Hilliard(CH) equation is a partial differential equation (PDE) solving the state of a two-phase fluid[2]. The form of the CH equation used in this thesis is

$$\begin{aligned} \partial_t \phi(x,t) &= \nabla \cdot (M(\phi) \nabla \mu) \\ \mu &= -\varepsilon^2 \Delta \phi + W'(\phi) \end{aligned} \tag{1}$$

where, ϕ is a phase-field variable representing the different states of the fluids through an interval $I = [-1, 1]$

$$\phi = \begin{cases} 1 & , \phi = \text{phase 1} \\ -1 & , \phi = \text{ phase 2} \end{cases}$$

ε is a positive constant correlated with boundary thickness and μ is the chemical potential[2].

In this thesis we assume $M(\phi) \equiv 1$, simplifying the CH equation used in [2] [1].

The advantages of the CH approach, as compared to traditional boundary coupling, are for example: "explicit tracking of the interface" [2], as well as "evolution of complex geometries and topological changes [...] in a natural way" [2]. In practice it enables linear interpolation between different formulas on different phases.

1.1 Derivation from paper

1.1.1 The free energy

The authors in [2] define the CH equation using the **Ginzburg-Landau** free energy equation:

$$E^{\text{bulk}} = \int_{\Omega} \frac{\varepsilon^2}{2} |\nabla \phi|^2 + W(\phi)\, dx \tag{2}$$

where $W(\phi)$ denotes the Helmholtz free energy density of mixing [2] that we approximate it in further calculations with $W(\phi) = \frac{(1-\phi^2)^2}{4}$ as in [1]. Additionally $\nabla\phi$ represents the change in phase-field.

The chemical potential, μ, then follows as the variational derivation of the free energy 2.

$$\mu = \frac{\delta E_{bulk}(\phi)}{\delta \phi} = -\varepsilon^2 \Delta \phi + W'(\phi)$$

1.1.2 Derivation of the CH equation from mass balance

The paper [2] motivates us to derive the CH equation as follows:

$$\partial_t \phi + \nabla \cdot J = 0 \tag{3}$$

where J is mass flux. The equation 3 then ensures continuity of mass Using the no-flux boundary conditions:

$$J \cdot n = 0 \qquad \partial\Omega \times (0,T) \tag{4}$$

$$\partial_n \phi = 0 \qquad \partial\Omega \times (0,T) \tag{5}$$

conservation of mass follows see[2].

$$\begin{aligned} \frac{d}{dt}\int_\Omega \phi &= \int_\Omega \frac{\partial \phi}{\partial t} dV \\ &= -\int_\Omega \nabla \cdot J \; dV \\ &= \int_{\partial\Omega} J \cdot n \; dA \\ &= 0 \end{aligned} \tag{6}$$

Therefore mass is conserved over time, as shown in 6. We define the mass flux, J, as the gradient in chemical potential as follows

$$J = -\nabla \mu \tag{7}$$

This results in the CH equation as stated in 1.

$$\begin{aligned} -\nabla \mu &= 0 \\ \partial_n \phi &= 0 \end{aligned} \tag{8}$$

i.e. no flow leaves and potential on the border doesn't change. In order to show the CH equation's consistency with thermodynamics we take the time derivation of the free energy 2 and we show that it decreases in time.

$$\begin{aligned} \frac{d}{dt}E^{bulk}(\phi(t)) &= \int_\Omega (\varepsilon^2 \nabla\phi \cdot \nabla \partial_t \phi + W'(\phi)\partial_t \phi) \; dx \\ &= \int_\Omega (\varepsilon^2 \nabla\phi + W'(\phi))\partial_t \phi \; dx \\ &= \int_\Omega \mu \partial_t \phi \; dx \\ &= \int_\Omega \mu \cdot \Delta\mu \; dx \\ &= -\int_\Omega \nabla\mu \cdot \nabla\mu \; dx + \int_{\partial\Omega} \mu \nabla\phi_t \cdot n \; dS \\ &\overset{\partial_n\phi = 0}{=} -\int_\Omega |\nabla\mu|^2 \; dx, \qquad \forall t \in [0,T) \end{aligned}$$

2 Baseline multi-grid solver

As baseline for numerical experiments we use a two-grid method based on the finite difference method defined in [1].

2.1 The discretization of the CH equation:

Our discretization closely resembles the one taken by the authors in [1]. We discretize our domain Ω to be a Cartesian-grid on a square with side-length $N \cdot h$, where N is the number of grid-points in one direction, and h is the distance between grid-points. In all our initial data h is $3 \cdot 10^{-3}$ and $N = 64$ for all, but in the stability tests in space. We discretize the phase-field ,ϕ, and chemical potential ,μ, into grid-wise functions ϕ_{ij}, μ_{ij}, where ϕ_{ij} represents the evaluation of ϕ at index ij , and at coordinates $(i \cdot h - 1, j \cdot h - 1)$. The authors in [1] use the characteristic function G of the domain Ω to enforce no-flux boundary conditions.

$$G(x, y) = \begin{cases} 1, & (x, y) \in \Omega \\ 0, & (x, y) \notin \Omega \end{cases}$$

We implement the discretized function on our square domain as follows.

$$G_{ij} = \begin{cases} 1, & (i, j) \in [2, N + 1]^2 \\ 0, & \text{else} \end{cases}$$

the domain we calculate on is therefore a square starting at $(2, 2)$ and ending at $(N+1, N+1)$. We use this shifted square to accommodate for zero padding in our numerical implementation.

```
function G(i, j, len, width)
    if 2 <= i <= len + 1 && 2 <= j <= width + 1
        return 1.0
    else
        return 0.0
    end
end
```

We then define the partial derivatives $D_x\phi_{ij}$, $D_y\phi_{ij}$ using centred differences:

$$D_x\phi_{i+\frac{1}{2}j} = \frac{\phi_{i+1j} - \phi_{ij}}{h} \qquad D_y\phi_{ij+\frac{1}{2}} = \frac{\phi_{ij+1} - \phi_{ij}}{h} \tag{9}$$

For $\nabla_d \phi_{ij}, \nabla_d \cdot (G_{ij} \nabla_d \phi_{ij})$ then follows:

$$\nabla_d \phi_{ij} = (D_x \phi_{i+1j},\ D_y \phi_{ij+1}) \tag{10}$$

$$\nabla_d \cdot (G_{ij} \nabla_d \phi_{ij}) = \frac{D_x \phi_{i+\frac{1}{2}j} - D_x \phi_{i-\frac{1}{2}j} + D_y \phi_{ij+\frac{1}{2}} - D_y \phi_{ij-\frac{1}{2}}}{h}, \tag{11}$$

where $\nabla_d \phi_{ij}$ is a discrete gradient, and $\nabla_d \cdot (G_{ij} \nabla_d \phi_{ij})$ is a discrete version of the Laplace operator Δ that takes no-flux boundary conditions into account. The authors in [1] show this to be the case by expanding $\nabla_d \cdot (G_{ij} \nabla_d \phi_{ij})$. In one dimension this expands to:

$$\nabla_d \cdot (G_i \nabla_d \phi_i) = \frac{G_{i+\frac{1}{2}} \phi_{i+1} + G_{i-\frac{1}{2}} \phi_{i-1} - G_{i+\frac{1}{2}} \phi_i - G_{i-\frac{1}{2}} \phi_i}{h^2} \tag{12}$$

notably, when one point lies outside the domain, then $G_{i\frac{\pm 1}{2}} = 0$ and therefore the corresponding discrete gradient $\frac{\phi_{i\pm\frac{1}{2}} - \phi_i}{h}$ is weighted by 0. This corresponds the discrete version of $\partial_n \phi = 0$. The authors in [1]

To simplify the notation for discretized derivatives we use the following abbreviations: Math:

- $\Sigma_G \phi_{ij} = G_{i+\frac{1}{2}j} \phi_{i+1j}^{n+\frac{1}{2},m} + G_{i-\frac{1}{2}j} \phi_{i-1j}^{n+\frac{1}{2},m} + G_{ij+\frac{1}{2}} \phi_{ij+1}^{n+\frac{1}{2},m} + G_{ij-\frac{1}{2}} \phi_{ij-1}^{n+\frac{1}{2},m}$
- $\Sigma_{Gij} = G_{i+\frac{1}{2}j} + G_{i-\frac{1}{2}j} + G_{ij+\frac{1}{2}} + G_{ij-\frac{1}{2}}$

Code:

```
function neighbours_in_domain(i, j, G, len, width)
    (
        G(i + 0.5, j, len, width)
        + G(i - 0.5, j, len, width)
        + G(i, j + 0.5, len, width)
        + G(i, j - 0.5, len, width)
    )

end
function discrete_G_weigted_neigbour_sum(i, j, arr, G, len, width)
    (
        G(i + 0.5, j, len, width) * arr[i+1, j]
        + G(i - 0.5, j, len, width) * arr[i-1, j]
        + G(i, j + 0.5, len, width) * arr[i, j+1]
        + G(i, j - 0.5, len, width) * arr[i, j-1]
    )
end
```

We can then write the modified Laplacian $\nabla_d(G\nabla_d f_{ij})$ as:

$$\nabla_d \cdot (G\nabla_d f_{ij}) = \frac{\Sigma_G f_{ij} - \Sigma_G \cdot f_{ij}}{h^2}$$

We use this modified Laplacian to deal with boundary conditions. Our abbreviations simplify separating implicit and explicit terms in the discretization.

2.2 Initial data

For testing we use initial phase-fields defined by the following equations:

$$\begin{aligned}
\phi_{ij} &= \begin{cases} 1 & ,\|(i,j) - (\frac{N}{2},\frac{N}{2})\|_p < \frac{N}{3} \\ -1 & ,else \end{cases} && \text{where } p \in \{2,\infty\} \\
\phi_{ij} &= \begin{cases} 1 & ,i < \frac{N}{2} \\ -1 & ,else \end{cases} \\
\phi_{ij} &= \begin{cases} 1 & ,\|(i,j) - (\frac{N}{2},2)\|_2 < \frac{N}{3} \\ -1 & ,else \end{cases} \\
\phi_{ij} &= \begin{cases} 1 & ,\|(i,j) - q_k\|_p < \frac{N}{5} \\ -1 & ,else \end{cases} && p \in \{1,2,\infty\}, q_k \in Q
\end{aligned} \tag{13}$$

where q_k are random points inside my domain. Those we generate those using the following rng setup in julia

```
using Random
rng = MersenneTwister(42)
gridsize = 64
radius = gridsize /5
blobs = gridsize ÷ 5
rngpoints = rand(rng,1:gridsize, 2, blobs)
```

```
2×12 Matrix{Int64}:
 48  40  20   1  63  49   8  60  26  58  26  11
 17  13  56  52  15   9  30  14  40   9  40  25
```

```
using Random
function testdata(gridsize , blobs , radius ,norm;rng=MersenneTwister(42))
rngpoints = rand(rng,1:gridsize, 2, blobs)
M = zeros(gridsize,gridsize) .- 1
```

```
for p in axes(rngpoints , 2)
    point = rngpoints[:, p]
    for I in CartesianIndices(M)
        if (LinearAlgebra.norm(point .- I.I  , norm) < radius)
            M[I] = 1
        end
    end
end
M
end
```

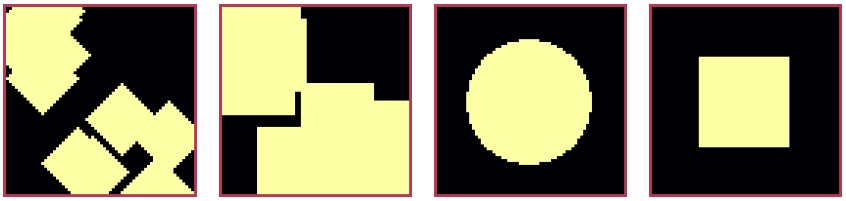


Figure 1: Examples of different phase-fields used as the initial condition in this work.

2.3 Numerical ansatz

The authors in [1] then define the discrete CH equation adapted for the domain as:

$$\begin{aligned} \frac{\phi_{ij}^{n+1} - \phi_{ij}^{n}}{\Delta t} &= \nabla_d \cdot (G_{ij} \nabla_d \mu_{ij}^{n+\frac{1}{2}}) \\ \mu_{ij}^{n+\frac{1}{2}} &= 2\phi_{ij}^{n+1} - \varepsilon^2 \nabla_d \cdot (G_{ij} \nabla_d \phi_{ij}^{n+1}) + W'(\phi_{ij}^{n}) - 2\phi_{ij}^{n} \end{aligned} \tag{14}$$

and derive a numerical scheme from this implicit equation.

2.4 PDE as operator L

The authors in [1] derive their method by seperating 14 into implicit and linear terms, and explicit non-linear terms. Linear terms are collected in an Operator L, and the explicit terms in $(\zeta_{ij}^{n}, \psi_{ij}^{n})^T$. We derive the iteration operator $L(\phi_{ij}^{n+1}, \mu_{ij}^{n+\frac{1}{2}}) = (\zeta_{ij}^{n}, \psi_{ij}^{n})$as in [1].

$$L \begin{pmatrix} \phi_{ij}^{n+1} \\ \mu_{ij}^{n+\frac{1}{2}} \end{pmatrix} = \begin{pmatrix} \frac{\phi_{ij}^{n+1}}{\Delta t} - \nabla_d \cdot (G_{ij} \nabla_d \mu_{ij}^{n+\frac{1}{2}}) \\ \varepsilon^2 \nabla_d \cdot (G \nabla_d \phi_{ij}^{n+1}) - 2\phi_{ij}^{n+1} + \mu_{ij}^{n+\frac{1}{2}} \end{pmatrix}$$

```
function L(solver::multi_solver,i,j , phi , mu)
    xi = solver.phase[i, j] / solver.dt -
         (discrete_G_weigted_neigbour_sum(i, j, solver.potential, G,
         ↪  solver.len, solver.width)
          -
          neighbours_in_domain(i, j, G, solver.len, solver.width) * mu
          ↪  )/solver.h^2
    psi = solver.epsilon^2/solver.h^2 *
          (discrete_G_weigted_neigbour_sum(i, j, solver.phase, G,
          ↪  solver.len, solver.width)
           -
           neighbours_in_domain(i, j, G, solver.len, solver.width) * phi)
           ↪  - 2 * phi + mu
    return [xi, psi]
end
```

This operator follows from 14 by separating implicit and explicit terms L and $(\zeta_{ij}^n, \psi_{ij}^n)^T$, respectively.

$$\begin{pmatrix} \zeta_{ij}^n \\ \psi_{ij}^n \end{pmatrix} = \begin{pmatrix} \frac{\phi_{ij}^n}{\Delta t} \\ W'(\phi_{ij}^n) - 2\phi_{ij}^n \end{pmatrix}$$

Due to being explicit, we know everything needed to calculate $(\zeta_{ij}^n, \psi_{ij}^n)^T$ at the beginning of each time step. We compute those values once and store them in the solver.

```
function set_xi_and_psi!(solver::T) where T <: Union{multi_solver ,
↪  relaxed_multi_solver}
    xi_init(x) = x / solver.dt
    psi_init(x) = solver.W_prime(x) - 2 * x
    solver.xi[2:end-1, 2:end-1] = xi_init.(solver.phase[2:end-1,2:end-1])
    solver.psi[2:end-1, 2:end-1] =
    ↪  psi_init.(solver.phase[2:end-1,2:end-1])
    return nothing
end
```

Furthermore, as it enables a Newton iteration, we derive its Jacobian with respect to the current grid point $(\phi_{ij}^{n+1}, \mu_{ij}^{n+\frac{1}{2}})^T$:

$$DL \begin{pmatrix} \phi_{ij} \\ \mu_{ij} \end{pmatrix} = \begin{pmatrix} \frac{1}{\Delta t} & \frac{1}{h^2}\Sigma_{Gij} \\ -\frac{\varepsilon^2}{h^2}\Sigma_{Gij} - 2 & 1 \end{pmatrix}$$

```
function dL(solver::multi_solver , i , j)
```

```
    return [ (1/solver.dt)
    ↪  (1/solver.h^2*neighbours_in_domain(i,j,G,solver.len ,
    ↪  solver.width));
            (-1*solver.epsilon^2/solver.h^2 *
            ↪  neighbours_in_domain(i,j,G,solver.len , solver.width) -
            ↪  2) 1]
    end
```

2.5 SMOOTH operator

The authors [1]derived Gauss-Seidel Smoothing from:

$$L \begin{pmatrix} \phi_{ij}^{n+1} \\ \mu_{ij}^{n+\frac{1}{2}} \end{pmatrix} = \begin{pmatrix} \zeta_{ij}^{n} \\ \psi_{ij}^{n} \end{pmatrix} \tag{15}$$

SMOOTH consists of point-wise Gauss-Seidel relaxation, by solving 15 for all i, j with the initial guess for $\zeta_{ij}^n, \psi_{ij}^n$. Since L is linear we can write 15 as

$$\begin{pmatrix} \zeta_{ij}^{n} \\ \psi_{ij}^{n} \end{pmatrix} = DL \begin{pmatrix} \phi_{ij}^{n+1} \\ \mu_{ij}^{n+\frac{1}{2}} \end{pmatrix} \cdot \begin{pmatrix} \phi_{ij}^{n+1} \\ \mu_{ij}^{n+\frac{1}{2}} \end{pmatrix} + \begin{pmatrix} -\frac{1}{h^2}\Sigma_{Gij}\mu_{ij}^{n+\frac{1}{2}} \\ +\frac{\varepsilon^2}{h^2}\Sigma_{Gij}\phi_{ij}^{n+1} \end{pmatrix} \tag{16}$$

where

- $\Sigma_G \phi_{ij}^{n+1} = G_{i+\frac{1}{2}j}\phi_{i+1j}^{n+1,m} + G_{i-\frac{1}{2}j}\phi_{i-1j}^{n+1,m} + G_{ij+\frac{1}{2}}\phi_{ij+1}^{n+1,m} + G_{ij-\frac{1}{2}}\phi_{ij-1}^{n+1,m}$,
- $\Sigma_G \mu_{ij} = G_{i+\frac{1}{2}j}\mu_{i+1j}^{n+\frac{1}{2},m} + G_{i-\frac{1}{2}j}\mu_{i-1j}^{n+\frac{1}{2},m} + G_{ij+\frac{1}{2}}\mu_{ij+1}^{n+\frac{1}{2},m} + G_{ij-\frac{1}{2}}\mu_{ij-1}^{n+\frac{1}{2},m}$,

since values for $\phi_{kl}^{n+1,m}, \mu_{kl}^{n+\frac{1}{2},m}$ are unknown, the authors in [1] and we use initial approximations, if $k > i, l > j$, and the values of the current smooth iteration else. As initial approximation we use the values of $\phi_{kl}^{n+1,m}, \mu_{kl}^{n+\frac{1}{2},m}$ from the last smoothing iteration. We then and solve 16 for $\phi_{ij}^{n+1}, \mu_{ij}^{n+\frac{1}{2}}$.

```
function SMOOTH!(
    solver::T,
    iterations,
    adaptive
) where T <: Union{multi_solver, adapted_multi_solver ,
↪  gradient_boundary_solver}
    for k = 1:iterations
        old_phase = copy(solver.phase)
        for I in CartesianIndices(solver.phase)[2:end-1, 2:end-1]
            i, j = I.I
```

```
            bordernumber = neighbours_in_domain(i, j, G, solver.len,
            ↪  solver.width)
            coefmatrix = dL(solver, i,j )

            b = [(
                        solver.xi[i, j]
                        +
                        discrete_G_weigted_neigbour_sum(
                            i, j, solver.potential, G, solver.len,
                            ↪  solver.width
                        ) / solver.h^2
                    ), (
                        solver.psi[i, j]
                        -
                        (solver.epsilon^2 / solver.h^2) *
                        ↪  discrete_G_weigted_neigbour_sum(
                            i, j, solver.phase, G, solver.len,
                            ↪  solver.width
                        ))]

            res = coefmatrix \ b
            solver.phase[i, j] = res[1]
            solver.potential[i, j] = res[2]

        end

        #if adaptive && LinearAlgebra.norm(old_phase - solver.phase) <
        ↪  1e-8
        #    #println("SMOOTH terminated at $(k) succesfully")
        #    break
        #end
    end
end
```

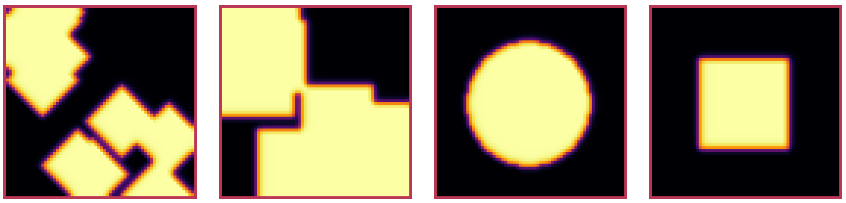


Figure 2: inputs from 2.2 after SMOOTH

2.6 V-cycle approach

The numerical method proposed in [1] consists of a V-cycle multi-grid method derived from previously stated operators. Specificly we use a two-grid implementation consisting of

1. a Gauss-Seidel relaxation for smoothing 4.3.
2. restriction and prolongation methods between grids $h \leftrightarrow H$.
3. a Newton iteration to solve $L(\phi_{ij,H}^{n+1,m}, \mu_{ij,H}^{n+\frac{1}{2},m})_H = L(\bar{\phi}_{ij,H}^{n+1,m}, \bar{\mu}_{ij,H}^{n+\frac{1}{2},m}) + (d_{ij,H}^{n+1,m}, r_{ij,H}^{n+1,m})$. we solve using the same iteration as in 4.3 however we replace $(\zeta_{ij}^n, \psi_{ij}^n)$ with $L(\bar{\phi}_{ij,H}^{n+1,m}, \bar{\mu}_{ij,H}^{n+\frac{1}{2},m}) + (d_{ij,H}^{n+1,m}, r_{ij,H}^{n+1,m})$. in the iteration, where $\bar{\phi}_{ij,H}^{n+1,m}, \bar{\mu}_{ij,H}^{n+\frac{1}{2},m}$ are the values after the smooth restricted to the coarser grid and $d_{ij,H}^{n+1,m}, r_{ij,H}^{n+1,m}$ is the residual from the smooth iteration on the fine grid restricted onto the coarse grid.

The V-cycle of a two-grid method using pre and post smoothing is then stated by:

```
function v_cycle!(grid::Array{T}, level) where T <: solver
    solver = grid[level]
    #pre SMOOTHing:
    SMOOTH!(solver, 400, true)

    d = zeros(size(solver.phase))
    r = zeros(size(solver.phase))

    # calculate error between L and expected values
    for I in CartesianIndices(solver.phase)[2:end-1, 2:end-1]
        d[I], r[I] = [solver.xi[I], solver.psi[I]] .- L(solver, I.I...,
        ↪  solver.phase[I], solver.potential[I])
    end

    restrict_solver!(grid[level], grid[level+1])
    solver = grid[level+1]
    solution = deepcopy(solver)

    d_large = restrict(d, G)
    r_large = restrict(r, G)


    u_large = zeros(size(d_large))
    v_large = zeros(size(d_large))
```

```
    #Newton Iteration for solving smallgrid
    for i = 1:300
        for I in CartesianIndices(solver.phase)[2:end-1, 2:end-1]

            diffrence = L(solution, I.I..., solution.phase[I],
            ↪  solution.potential[I]) .- [d_large[I], r_large[I]] .-
            ↪  L(solver, I.I..., solver.phase[I], solver.potential[I])
            #diffrence = collect(L(solution, I.I...)) .- collect(L(solver,
            ↪  I.I...))
            #diffrence = [d_large[I] , r_large[I]]

            local ret = dL(solution, I.I...) \ diffrence

            u_large[I] = ret[1]
            v_large[I] = ret[2]
        end
        solution.phase .-= u_large
        solution.potential .-= v_large
    end
    u_large = solver.phase .- solution.phase
    v_large = solver.potential .- solution.potential

    solver = grid[level]

    solver.phase .+= prolong(u_large , G)
    solver.potential .+= prolong(v_large, G)
    SMOOTH!(solver, 800, true)
end
```

So let's take a closer look at the internals, namely the phase field after pre-SMOOTHing $\bar{\phi}$, the phase residuals of $\left[L(\bar{\phi_{ij}}, \bar{\mu_{ij}}) - (\zeta_{ij}, \psi_{ij})\right]_{ij\in\Omega}$ and the result of the Newton iteration on coarsest level. After a few iterations, V-cycle exhibits the following behavior:

```
<<init>>
using JLD2
using DataFrames
results = jldopen("experiments/iteration.jld2")["result"]
anim = @animate for res in eachrow(results)
    heatmap(res.solver.phase , xlims = (2,size(res.solver.phase , 1)-1) ,
    ↪  ylim=(2,size(res.solver.phase , 1)-1) , aspectratio=:equal)
end
gif(anim , "images/iteration.gif" , fps = 10)
```

3 Numerical evaluation

The analytical CH equation conserves mass 3 and the free energy ,E_{bulk}, 2 decreases in time, i.e. consistence with the second law of thermodynamics. Therefore, we use discrete variants of those concepts as necessary conditions for a "good" solution. Furthermore, since E_{bulk} is closely correlated with chemical potential, μ, we evaluate this difference as quality of convergence.

3.1 Energy evaluations

As discrete energy measure we use:

$$\begin{aligned} E_d^{\text{bulk}} &= \sum_{i,j\in\Omega} \frac{\varepsilon^2}{2}|G\nabla\phi_{ij}|^2 + W\left(\phi_{ij}\right)\, dx \\ &= \sum_{i,j\in\Omega} \frac{\varepsilon^2}{2} G_{i+\frac{1}{2}j}(D_x\phi_{i+\frac{1}{2}j})^2 + G_{ij+\frac{1}{2}}(D_y\phi_{ij+\frac{1}{2}})^2 + W\left(\phi_{ij}\right)\, dx \end{aligned}$$

```
function bulk_energy(solver::T) where T <: Union{multi_solver ,
↪  relaxed_multi_solver}
    energy = 0
    dx = CartesianIndex(1,0)
    dy = CartesianIndex(0,1)
    W(x) = 1/4 * (1-x^2)^2
    for I in CartesianIndices(solver.phase)[2:end-1,2:end-1]
        i,j = I.I
        energy += solver.epsilon^2 / 2 * G(i+ 0.5,j ,solver.len,
        ↪  solver.width) * (solver.phase[I+dx] - solver.phase[I])^2 +
        ↪  G(i,j+0.5,solver.len ,solver.width) * (solver.phase[I+dy] -
        ↪  solver.phase[I])^2 + W(solver.phase[I])
        end
   return energy
end
```

3.2 Mass balance

Instead of a physical mass we use the average of ϕ over the domain Ω written as:

$$\frac{1}{|\Omega|}\int_\Omega \phi\, dx \tag{17}$$

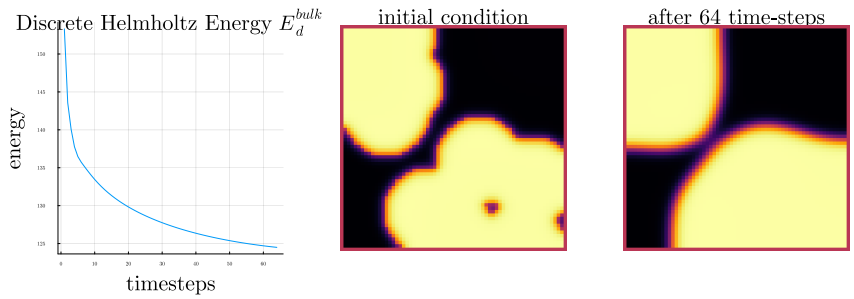


Figure 3: behaviour of energy E_{bulk} over time for one initial condition ϕ_0

We calculate this balance as:

$$b = \frac{\sum_{i,j \in \Omega} \phi_{ij}}{|\{(i,j) \in \Omega\}|}$$

such that $b = 1$ means there is only phase 1, $\phi \equiv 1$, and $b = -1$ means there is only phase 2, $\phi \equiv -1$.

```
function massbal(arr)
    num_cells= *((size(arr).-2)...)
    return sum(arr[2:end-1, 2:end-1])/num_cells
    end
```

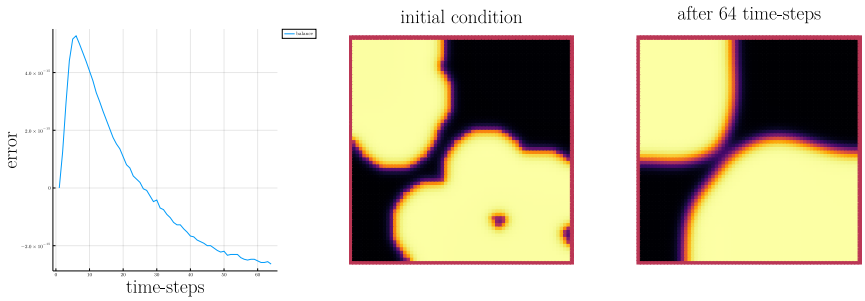


Figure 4: behaviour of phase change over time for one initial condition ϕ_0

3.3 TODO stability

3.3.1 stability of a sub iteration v-cycle

in order to evaluate convergence we observe the change in phase

$$\|\phi^n - \phi^{n+1,m}\|_{Fr} \tag{18}$$

where $\|\cdot\|_{Fr}$ represents a Frobenious norm over the tensors representing $\phi.^n, \phi^{n+1,m}$. In addition we track the change of bulk energy:

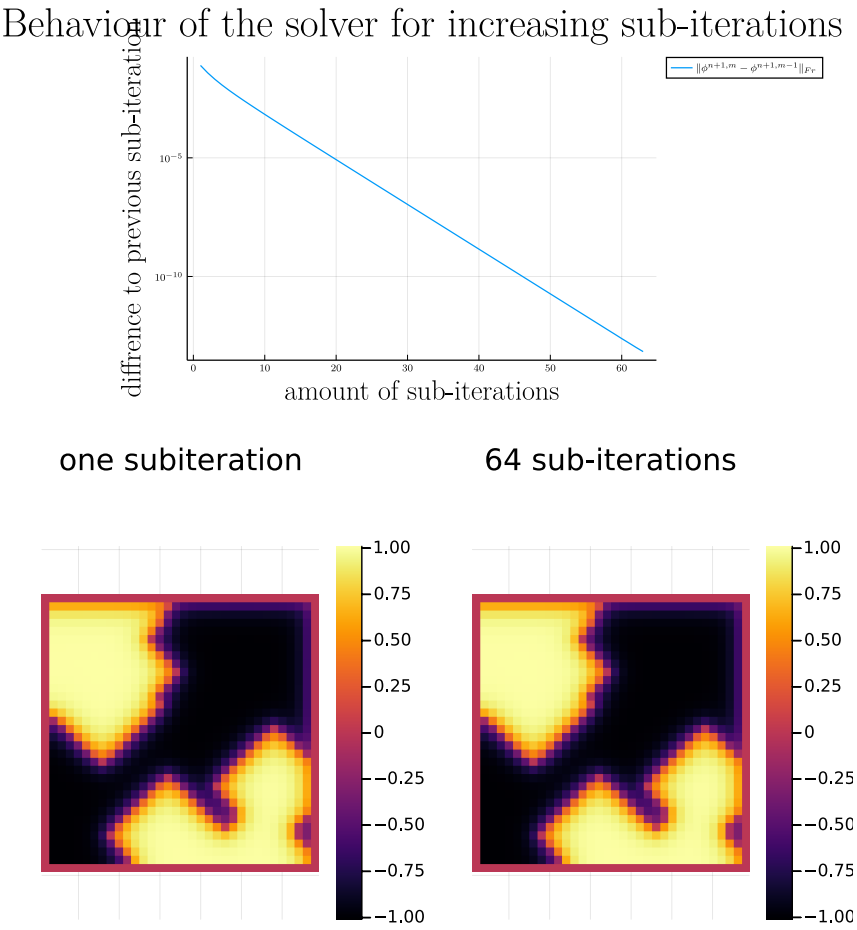
$$\frac{d}{dt}E^{bulk} = -\int_\Omega |\nabla\mu|^2 \, dx \tag{19}$$

discetized as follows:

$$\Delta E^{bulk} = -\sum_{ij\in\Omega} |\nabla_d\mu|^2 \tag{20}$$

```
function bulk_energy_potential(solver::T) where T <: solver
    energy = 0
    dx = CartesianIndex(1,0)
    dy = CartesianIndex(0,1)
    W(x) = 1/4 * (1-x^2)^2
    for I in CartesianIndices(solver.phase)[2:end-1,2:end-1]
        i,j = I.I
        energy +=  G(i+ 0.5,j ,solver.len, solver.width) *
        ↪  (solver.potential[I+dx] - solver.potential[I])^2 +
        ↪  G(i,j+0.5,solver.len ,solver.width) * (solver.potential[I+dy]
        ↪  - solver.potential[I])^2
        end
   return energy
end
```

we expect our solver to converge if we do more sub-iterations. To test this we compare the phase-field $\phi_{ij}^{n+1,m-1}$ after $m-1$ sub-iterations with the phase-field $\phi_{ij}^{n+1,m}$ after m sub-iterations. As sub-iterations increase , $m \to \infty$ we expect the difference between both phase-fields to go to zero $\|\phi^{n+1,m} - \phi^{n+1,m-1}\|_{Fr} \to 0$



3.3.2 stability under refinment in time

we test the bahaviour unter refinement in time by succesivly subdeviding the original time interval $[0, T]$ in finer parts

3.3.3 stability under refinement in space

We expect our methods to be stable in space. Therefore we expect

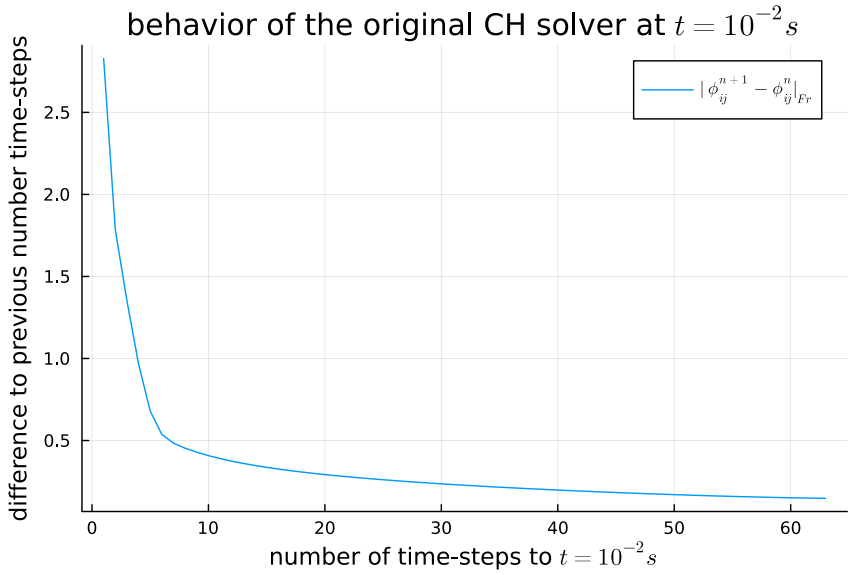


Figure 5: behavior of the baseline solver while solving the time interval $T = \left[0, 10^{-2}\right]$ with increasing number of time-steps

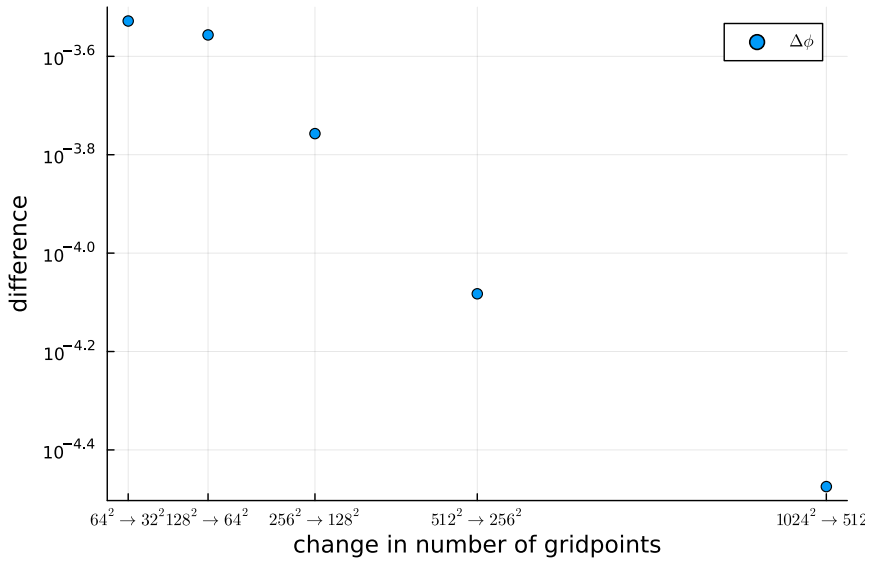


Figure 6: behavior of the baseline solver while solving on successively finer grids

4 Relaxed problem

In effort to decrease the order of complexity, from fourth order derivative to second order, we propose an elliptical relaxation approach, where the relaxation variable c is the solution of the following elliptical PDE:

$$-\Delta c^\alpha + \alpha c^a = \alpha \phi^\alpha, \tag{21}$$

where α is a relaxation parameter. We expect to approach the original solution of the CH equation 1 as $\alpha \to \infty$. This results in the following relaxation for the classical CH equation 1:

$$\begin{aligned} \partial_t \phi^\alpha &= \Delta \mu \\ \mu &= \varepsilon^2 \alpha (c^\alpha - \phi^\alpha) + W'(\phi) \end{aligned} \tag{22}$$

It requires solving the elliptical PDE each time-step to calculate c.

As ansatz for the numerical solver we propose:

$$\begin{aligned} \frac{\phi_{ij}^{n+1,\alpha} - \phi_{ij}^{n,\alpha}}{\Delta t} &= \nabla_d \cdot (G_{ij} \nabla_d \mu_{ij}^{n+\frac{1}{2},\alpha}) \\ \mu_{ij}^{n+\frac{1}{2},\alpha} &= 2\phi_{ij}^{n+1,\alpha} - \varepsilon^2 a (c_{ij}^{n+1,\alpha} - \phi_{ij}^{n+1,\alpha}) + W'(\phi_{ij}^{n,\alpha}) - 2\phi_{ij}^{n,\alpha} \end{aligned} \tag{23}$$

This approach is inspired by 14 adapted to the relaxed CH equation 23. We then adapt the multi-grid solver proposed in 2 to the relaxed problem by replacing the differential operators by their discrete counterparts as defined in 10, and expand them.

4.1 Elliptical PDE:

In order to solve the relaxed CH equation we solve the following PDE in each time step:

$$-\nabla \cdot (G \nabla c^\alpha) + \alpha c^\alpha = \alpha \phi^\alpha$$

Similarly to the first solver we solve this PDE with a finite difference scheme using the same discretization as before.

4.1.1 Discretization

The discretization of the PDE expands the differential operators in the same way and proposes an equivalent scheme for solving the elliptical equation 21.

$$-\nabla_d \cdot (G_{ij} \nabla_d c_{ij}^\alpha) + \alpha c_{ij}^\alpha = \alpha \phi_{ij}^\alpha$$

$$\Longrightarrow$$

$$\begin{aligned}-(\frac{1}{h}(G_{i+\frac{1}{2}j}\nabla c^{\alpha}_{i+\frac{1}{2}j} + G_{ij+\frac{1}{2}}\nabla c^{\alpha}_{ij+\frac{1}{2}})\\ -(G_{i-\frac{1}{2}j}\nabla c^{\alpha}_{i-\frac{1}{2}j} + G_{ij-\frac{1}{2}}\nabla c^{\alpha}_{ij-\frac{1}{2}})) + \alpha c^{\alpha}_{ij} = \alpha\phi^{\alpha}_{ij}\end{aligned}$$

$$\Longrightarrow$$

$$\begin{aligned}-\frac{1}{h^2}(G_{i+\frac{1}{2}j}(c^{\alpha}_{i+1j} - c^{\alpha}_{ij})\\ +G_{ij+\frac{1}{2}}(c^{\alpha}_{ij+1} - c^{\alpha}_{ij})\\ +G_{i-\frac{1}{2}j}(c^{\alpha}_{i-1j} - c^{\alpha}_{ij})\\ +G_{ij-\frac{1}{2}}(c^{\alpha}_{ij-1} - c^{\alpha}_{ij})) + \alpha c^{\alpha}_{ij} = \alpha\phi^{\alpha}_{ij}\end{aligned}$$

As before we abbreviate $\Sigma_G c^{\alpha}_{ij} = G_{i+\frac{1}{2}j}c^{\alpha}_{i+1j} + G_{i-\frac{1}{2}j}c^{\alpha}_{i-1j} + G_{ij+\frac{1}{2}}c^{\alpha}_{ij+1} + G_{ij-\frac{1}{2}}c^{\alpha}_{ij-1}$ and $\Sigma_{Gij} = G_{i+\frac{1}{2}j} + G_{i-\frac{1}{2}j} + G_{ij+\frac{1}{2}} + G_{ij-\frac{1}{2}}$. Then the discrete elliptical PDE can be stated as:

$$-\frac{\Sigma_G c^{\alpha}_{ij}}{h^2} + \frac{\Sigma_G}{h^2}c^{\alpha}_{ij} + \alpha c^{\alpha}_{ij} = \alpha\phi^{\alpha}_{ij} \tag{24}$$

solving 24 for c^{α}_{ij} then results in.

$$\begin{aligned}\left(\frac{\Sigma_{Gij}}{h^2} + \alpha\right)c^{\alpha}_{ij} &= \alpha\phi^{\alpha}_{ij} + \frac{\Sigma_G c^{\alpha}_{ij}}{h^2}\\ c^{\alpha}_{ij} &= \frac{\alpha\phi^{\alpha}_{ij} + \frac{\Sigma_G c^{\alpha}_{ij}}{h^2}}{\frac{\Sigma_G}{h^2} + \alpha}\\ c^{\alpha}_{ij} &= \frac{\alpha h^2\phi^{\alpha}_{ij}}{\Sigma_{Gij} + \alpha h^2} + \frac{\Sigma_G c^{\alpha}_{ij}}{\Sigma_{Gij} + \alpha h^2}\end{aligned}$$

and can be translated to code as follows

```
function elyps_solver!(solver::T, n) where T  <:
↪  Union{relaxed_multi_solver , adapted_relaxed_multi_solver}
    for k in 1:n
        for i = 2:(solver.len+1)
            for j = 2:(solver.width+1)
                bordernumber = neighbours_in_domain(i, j,G, solver.len,
                ↪  solver.width)
                solver.c[i, j] =
                    (
                        solver.alpha * solver.phase[i, j] +
```

```
                        discrete_G_weigted_neigbour_sum(i, j, solver.c, G,
                        ↪  solver.len, solver.width) / solver.h^2
                    ) / (bordernumber / solver.h^2 + solver.alpha)

            end
        end
    end
end
```

4.2 Relaxed PDE as operator L

We reformulate the discretization 23 in terms of the relaxed operator L as follows:

$$L_r \begin{pmatrix} \phi^{n+1,\alpha} \\ \mu^{n+\frac{1}{2},\alpha} \end{pmatrix} = \begin{pmatrix} \frac{\phi_{ij}^{n+1,m,\alpha}}{\Delta t} - \nabla_d \cdot (G_{ji} \nabla_d \mu_{ji}^{n+\frac{1}{2},m,\alpha}) \\ \varepsilon^2 \alpha (c^\alpha - \phi_{ij}^{n+1,m,\alpha}) - 2\phi_{ij}^{n+1,m,\alpha} - \mu_{ji}^{n+\frac{1}{2},m,\alpha} \end{pmatrix}$$

```
function L(solver::relaxed_multi_solver,i,j , phi , mu)
    xi = solver.phase[i, j] / solver.dt -
         (discrete_G_weigted_neigbour_sum(i, j, solver.potential, G,
         ↪  solver.len, solver.width)
          -
          neighbours_in_domain(i, j, G, solver.len, solver.width) * mu
          ↪  )/solver.h^2
    psi = solver.epsilon^2 * solver.alpha*(solver.c[i,j] - phi) - 2 *
    ↪  solver.phase[i,j] - solver.potential[i,j]
    return [xi, psi]
end
```

and its Jacobian:

$$DL_r \begin{pmatrix} \phi \\ \mu \end{pmatrix} = \begin{pmatrix} \frac{1}{\Delta t} & \frac{1}{h^2}\Sigma_G \\ -\varepsilon^2\alpha - 2 & 1 \end{pmatrix}$$

```
function dL(solver::relaxed_multi_solver , i , j)
    return [ (1/solver.dt)
    ↪  (1/solver.h^2*neighbours_in_domain(i,j,G,solver.len ,
    ↪  solver.width));
             (-1*solver.epsilon^2 * solver.alpha  - 2) 1]
    end
```

4.3 SMOOTH operator

The relaxed solver uses the same approach as the original solver, where we solve $L_r(\phi_{ij}^{n+1,m,\alpha}, \mu_{ij}^{n+\frac{1}{2},m,\alpha}) = (\zeta_{ij}^n, \psi_{ij}^n)^T$ for each grid-point $\phi_{ij}^{n+1,m,\alpha}$.

Notably $(\zeta_{ij}^n, \psi_{ij}^n)^T$ is the same as in the original part. As in the original smoothing evalations of $\mu_{kl}^{n+\frac{1}{2},m,\alpha}$ for $k,l > i,j$ are replaced with their values from the previous SMOOTH iteration. Correspondingly the SMOOTH operation expands to:

$$\begin{aligned} -\frac{\Sigma_{Gij}}{h^2}\overline{\mu_{ji}^{n+\frac{1}{2},m,\alpha}} &= \frac{\phi_{ij}^{n+1,m,\alpha}}{\Delta t} - \zeta_{ij}^{n,\alpha} - \frac{\Sigma_G \mu_{ij}}{h^2} \\ \varepsilon^2\alpha\overline{\phi_{ij}^{n+1,m,\alpha}} + 2\phi_{ij}^{n+1,m,\alpha} &= \varepsilon^2\alpha c_{ij}^{n,\alpha} - \overline{\mu_{ji}^{n+\frac{1}{2},m,\alpha}} - \psi_{ij}^{n,\alpha} \end{aligned} \tag{25}$$

where

- $\Sigma_G \mu_{ij} = G_{i+\frac{1}{2}j}\mu_{i+1j}^{n+\frac{1}{2},m} + G_{i-\frac{1}{2}j}\mu_{i-1j}^{n+\frac{1}{2},m} + G_{ij+\frac{1}{2}}\mu_{ij+1}^{n+\frac{1}{2},m} + G_{ij-\frac{1}{2}}\mu_{ij-1}^{n+\frac{1}{2},m}$,

We then solve directly for the smoothed variables, $\overline{\mu_{ij}^{n+1,m,\alpha}}$ and $\overline{\phi_{ij}^{n+1,m,\alpha}}$. This was not done in the original paper [1] because the required system of linear equations in the paper [1] was solved numerically.

$$\varepsilon^2\alpha(\phi_{ij}^{n+1,m,\alpha}) + 2\phi_{ij}^{n+1,m,\alpha} = \varepsilon^2\alpha c^\alpha - \frac{h^2}{\Sigma_G}\left(\frac{\phi_{ij}^{n+1,m,\alpha}}{\Delta t} - \zeta_{ij}^n - \frac{1}{h^2}\Sigma_G\mu_{ij}\right) - \psi_{ij}$$

$$\Longrightarrow$$

$$\varepsilon^2\alpha(\phi_{ij}^{n+1,m,\alpha}) + 2\phi_{ij}^{n+1,m,\alpha} + \frac{h^2}{\Sigma_{Gij}}\frac{\phi_{ij}^{n+1,m,\alpha}}{\Delta t} = \varepsilon^2\alpha c^\alpha - \frac{h^2}{\Sigma_G}\left(-\zeta_{ij}^n - \frac{1}{h^2}\Sigma_G\mu_{ij}\right) - \psi_{ij}$$

$$\Longrightarrow$$

$$\left(\varepsilon^2\alpha + 2 + \frac{h^2}{\Sigma_G\Delta t}\right)\phi_{ij}^{n+1,m,\alpha} = \varepsilon^2\alpha c^\alpha - \frac{h^2}{\Sigma_G}\left(-\zeta_{ij}^n - \frac{\Sigma_G\mu_{ij}}{h^2}\right) - \psi_{ij}$$

$$\Longrightarrow$$

$$\phi_{ij}^{n+1,m,\alpha} = \left(\varepsilon^2\alpha c^\alpha - \frac{h^2}{\Sigma_G}\left(-\zeta_{ij}^n - \frac{\Sigma_G\mu_{ij}}{h^2}\right) - \psi_{ij}\right)\left(\varepsilon^2\alpha + 2 + \frac{h^2}{\Sigma_G\Delta t}\right)^{-1}$$

```
function SMOOTH!(
    solver::T,
    iterations,
    adaptive
) where T <: Union{relaxed_multi_solver , adapted_relaxed_multi_solver}
    for k = 1:iterations
        old_phase = copy(solver.phase)
        for I in CartesianIndices(solver.phase)[2:end-1, 2:end-1]
            i, j = I.I
```

```
            bordernumber = neighbours_in_domain(i, j, G, solver.len,
            ↪  solver.width)


            solver.phase[I] = (solver.epsilon^2 * solver.alpha *
            ↪  solver.c[I] - solver.h^2 / bordernumber * ( -solver.xi[I]
            ↪  - discrete_G_weigted_neigbour_sum(i,j,solver.potential , G
            ↪  , solver.len , solver.width) / solver.h^2 ) -
            ↪  solver.psi[I]) / (solver.epsilon^2 * solver.alpha  + 2 +
            ↪  solver.h^2 / (bordernumber*solver.dt))

            #since the solver still needs the potetential we calculate it
            ↪  as well
            solver.potential[I] = (solver.phase[I]/solver.dt -
            ↪  solver.xi[I] - discrete_G_weigted_neigbour_sum(i,j,
            ↪  solver.potential , G , solver.len ,
            ↪  solver.width)/solver.h^2) * (-solver.h^2/bordernumber)
        end

        if adaptive && LinearAlgebra.norm(old_phase - solver.phase) <
        ↪  1e-10
            #println("SMOOTH terminated at $(k) succesfully")
            break
        end
    end
end
```

Furthermore, experimentation shows that alpha alone is insufficient to get a relaxed method consistent with the original solver, since alpha had an effect similar to epsilon, where it changed the boundary thickness in the phase-field ϕ. Therefore epsilon and alpha cannot be chosen independently. Hence we use a simple MCMC optimizer for α, ε in order to give the relaxed solver the best chance we can. Monte Carlo Optimizer For ε, α.

```
using Distributions
using DataFrames
using JLD2
include(pwd() * "/src/solvers.jl")
include(pwd() * "/src/adapted_solvers.jl")
include(pwd() * "/src/utils.jl")
include(pwd() * "/src/multisolver.jl")
include(pwd() * "/src/multi_relaxed.jl")
include(pwd() * "/src/testgrids.jl")
include(pwd() * "/src/elypssolver.jl")
using Plots
using LaTeXStrings
using LinearAlgebra
```

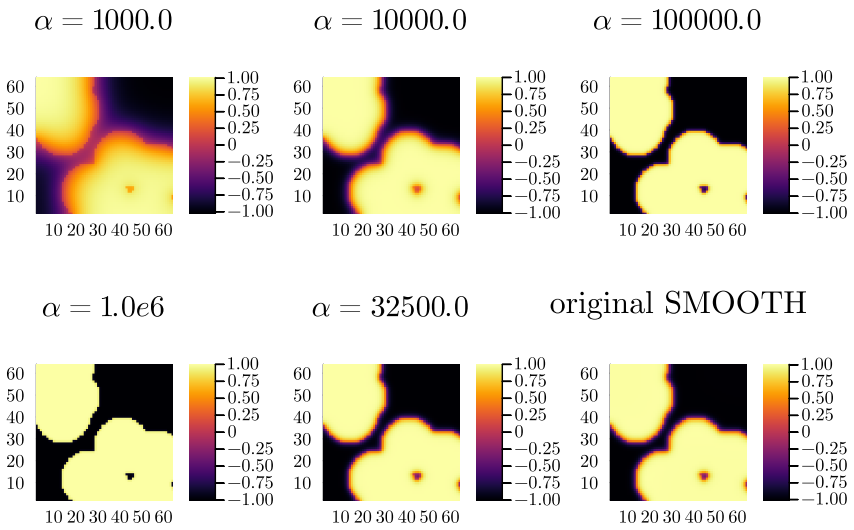


Figure 7: effect of the relaxed SMOOTH operator, and additional solving of the elliptical problem, for different values of alpha

```
using Printf
using ProgressBars
default(fontfamily="computer modern" , titlefontsize=23)
SIZE = 64
M = testdata(SIZE, SIZE ÷ 5, SIZE /5 , 2)

function test_values(alpha_distribution::Distribution ,
↪  epsilon_distribution::Distribution , M)
    alpha = rand(alpha_distribution)
    eps = max(rand(epsilon_distribution)  ,1e-10)
    relaxed_solver = testgrid(relaxed_multi_solver, M, 2; alpha=alpha,
    ↪  epsilon=eps)
    set_xi_and_psi!(relaxed_solver[1])
    elyps_solver!(relaxed_solver[1], 2000)
    #SMOOTH!(relaxed_solver[1], 100, false)
    for j=1:64
    v_cycle!(relaxed_solver , 1)
    end
    error = norm(relaxed_solver[1].phase .- original_solver[1].phase) /
    ↪  *(size(relaxed_solver[1].phase)...)
    return (;alpha=alpha , epsilon=eps , error=error)
end
```

```
original_solver = testgrid(multi_solver, M, 2)
set_xi_and_psi!(original_solver[1])
for j=1:64
v_cycle!(original_solver , 1)
end
#SMOOTH!(original_solver[1], 100, false);
eps = 3e-3
#M = testdata(64, div(64,3), 64/5 , 2)
alpha0 = 10000
epsilon0 = 1e-2
best_alpha = alpha0 / 10
best_epsilon = epsilon0 / 10
best_error  = Inf
results = DataFrame()
for n=1:1000
    searchradius = 1
    alpha_distribution = Normal(best_alpha , searchradius * alpha0)
    epsilon_distribution = Normal(best_epsilon , searchradius * epsilon0)
    result = test_values(alpha_distribution , epsilon_distribution , M)
    if result.error < best_error
        global best_error = result.error
        global best_alpha = result.alpha
        global best_epsilon = result.epsilon
        println(result)
    end
push!(results , result)
end
jldsave("experiments/alpha-epsilon.jld2"; result=results)
println("Best alpha: $best_alpha , Best epsilon: $best_epsilon")
```

sadly the MCMC didn't yield results consistent with the original solver after a few Iterations

4.4 The relaxed V-cycle approach

As the difference between both methods is abstracted away in the operators, the relaxed V-cycle the replaces the original operators with their relaxed counterparts. Due to julias multiple dispatch features this changes nothing in the implementation Therefore we reuse the original V-cycle in the 2.6. In the executions for each time step, we add the elliptic solver before the subiteration.

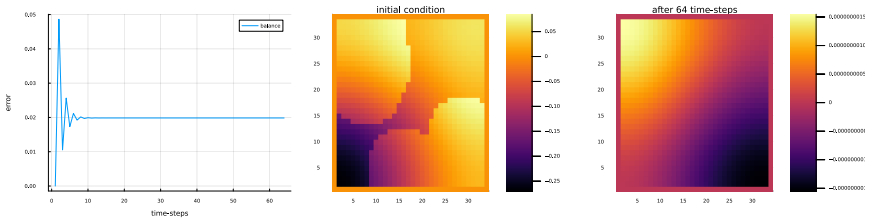
```
<<setup-relaxed-grid>>

pbar = ProgressBar(total = 1000)
```

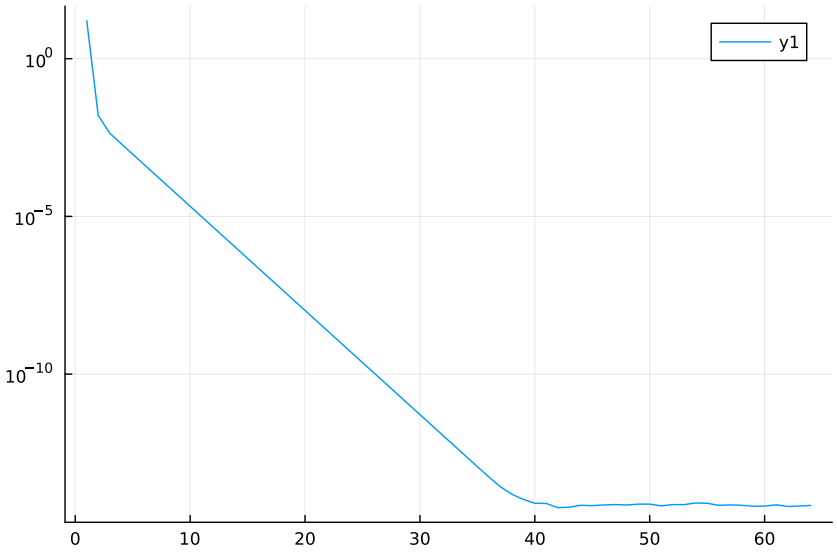
```
anim = @animate for t in 1:100
    set_xi_and_psi!(testgrd[1])
    elyps_solver!(solver , 1000)
    for j in 1:10
        v_cycle!(testgrd, 1)
        update(pbar)
        end
    heatmap(testgrd[1].phase , clim =(-1,1) , framestyle=:none )
end
gif(anim , "images/iteration_relaxed2.gif" , fps = 10)
```

4.5 rate of stability

4.5.1 massbal



4.5.2 convergence of a sub iteration v-cycle



4.5.3 convergence unter refinment in time

we test the bahaviour unter refinement in time by succesivly subdeviding the original time interval $[0, T]$ in finer parts

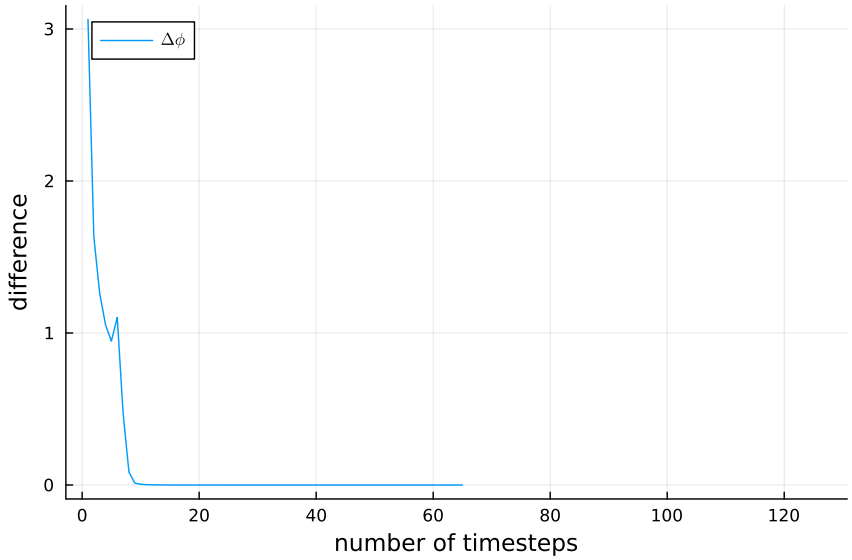
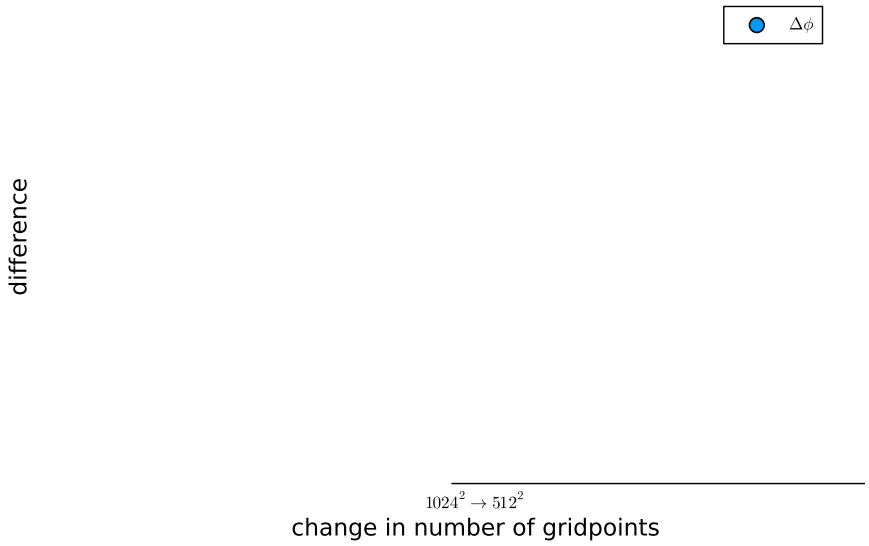


Figure 8: behavior of the baseline solver while solving the time interval $T = \left[0, 10^{-2}\right]$ with increasing number of timesteps

4.5.4 convergence under refinement in space

we test convergence in space by succesivly subdividing our grid into finer meshes



4.6 Comparison

```
<<setup-comparison>>
n = 100
m = 100
pbar = ProgressBar(total = n*m)

anim = @animate for i in 1:n
    set_xi_and_psi!(original_grid[1])
    set_xi_and_psi!(relaxed_grid1[1])
    set_xi_and_psi!(relaxed_grid2[1])
    set_xi_and_psi!(relaxed_grid3[1])
    elyps_solver!(relaxed_grid1[1] , 1000)
    elyps_solver!(relaxed_grid2[1] , 1000)
    elyps_solver!(relaxed_grid3[1] , 1000)
    for j in 1:m
        v_cycle!(original_grid, 1)
        v_cycle!(relaxed_grid1, 1)
        v_cycle!(relaxed_grid2, 1)
        v_cycle!(relaxed_grid3, 1)
        update(pbar)
        end
    p0 = heatmap(original_grid[1].phase , clim =(-1,1) , framestyle=:none
    ↪  , title="Original")
    p1 = heatmap(relaxed_grid1[1].phase , clim =(-1,1) , framestyle=:none,
    ↪  title="alpha=1e3" )
```

```
    p2 = heatmap(relaxed_grid2[1].phase , clim =(-1,1) , framestyle=:none,
    ↪  title="alpha=1e4" )
    p3 = heatmap(relaxed_grid3[1].phase , clim =(-1,1) , framestyle=:none,
    ↪  title="alpha=1e5" )
    plot(p0,p1,p2,p3)
end
gif(anim , "images/comparison.gif" , fps = 10)
```

`images/comparison.gif`

Furthermore we expect the approximation for ϕ_{ij}^{n+1} to converge.

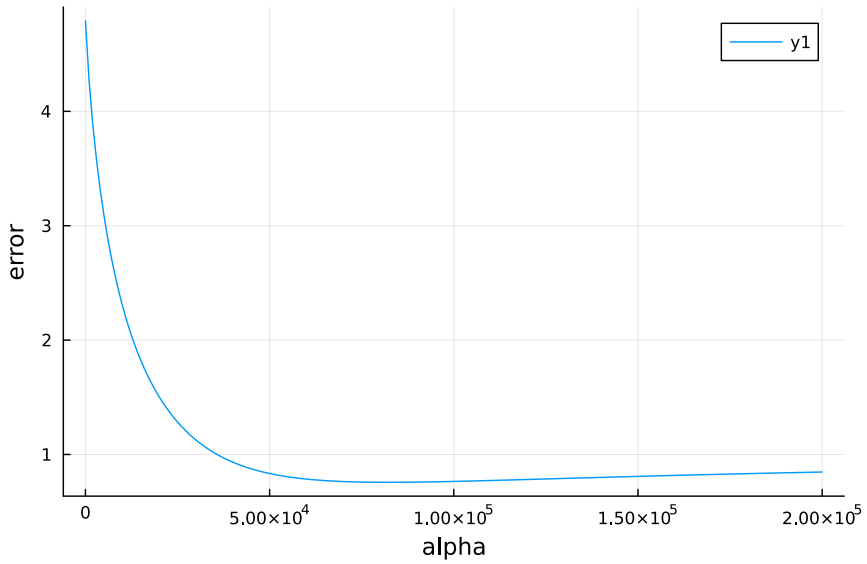
$$||\phi_{ij}^{n+1} - \phi_{ij}^{n+1,\alpha}|| \to 0 \tag{26}$$

In practice we observe the following behaviour:

```
<<init>>
using JLD2
using Distributed
JULIA_NUM_THREADS = 24
M = jldopen("data/test-phasefield.jld2")["M"]

original_grid = testgrid(multi_solver, M, 2)
alphas = 0:1e3:2e5

function alpha_error(alpha::Number , solution::Array )
    test_solver  = testgrid(relaxed_multi_solver, M, 2, alpha=)
    set_xi_and_psi!(test_solver[1])
    elyps_solver!(test_solver[1], 1000)
    for j in 1:100
        v_cycle!(test_solver , 1)
    end
return norm(test_solver[1].phase - solution)
end
set_xi_and_psi!(original_grid[1])
for j in 1:100
    v_cycle!(original_grid, 1)
end
print("finished original v_cycle")
tasks = []
for alpha in alphas
    t = Threads.@spawn alpha_error(alpha , original_grid[1].phase)
    push!(tasks , (alpha=alpha , task = t))
end
results  = @show [(alpha=t.alpha, error=fetch(t.task)) for t in tasks]
p=plot(results)
savefig(p, "images/alpha-error.svg")
```



in all cases the difference to the original solver is apparent. Furthermore we observe a optimal value of α at approximately $7.5 * 10^5$ we explain this with our observations done for the Smoothing operator, where for small and large values of α the relaxed approach ironically results in restricted behaviour. Empirical this is to be expected as. for large values of alpha the elliptical equation approaches ϕ and for small values the elliptical solver does not converge.

5 AI

We propose a data motivated alternative to the elliptical PDE in the relaxed CH equation 23. We propose there to be a better solution then the discrete result of 21. We define "better" as minimizing:

$$||\phi^{n+1} + \frac{1}{\alpha}\nabla \cdot (G\nabla\phi^{n+1}) - c||_{Fr} \tag{27}$$

in the Frobenious norm $||\cdot||_{Fr}$ and implement our loss function accordingly

```
loss(model , x , y) = Flux.Losses.mse(model(x) , y)
```

We calculate the expected value $\phi^{n+1} + \frac{1}{\alpha}\nabla \cdot (G\nabla\phi^{n+1})$ before training and use it as $\hat{y}$

```
function ggrad(x::AbstractArray, solver::T) where T <: solver
Indices = CartesianIndices(x)
Ifirst , Ilast = first(Indices) , last(Indices)
padding = oneunit(Ifirst)
res = zeros(size(x))

for I in (Ifirst + padding):(Ilast - padding)
    i,j = I.I
   res[I] = x[i] +
   ↪  (discrete_G_weigted_neigbour_sum(i,j,x,G,solver.len,solver.width)
            - neighbours_in_domain(i,j,G, solver.len , solver.width) *
            ↪  x[I])/ solver.h^2
    end
return res
end
```

```
Executing... 64f93dab
```

```
<<init>>
using JLD2
using DataFrames
df = jldopen("experiments/subiteration.jld2")["result"]
prep = (x) -> ggrad(x.phase , x) .+ x.phase
gd = groupby(df , :subiteration)
res = prep.(gd[end].cycle)
for sd in gd
       sd[!,:target] = res
end
df[!, :input] = [ s.phase for s in df.cycle]
data = select(df , [:input , :target])
reshape_data(d) = reshape(stack(d) , (size(d[1])... , 1 , :))
train_set = DataLoader((data=reshape_data(data.input) ,
↪  label=reshape_data(data.target)) , batchsize=32)
```

5.1 proposal CNN

```
using Flux

model = Chain(
    Conv((5,5) , 1=>5; stride=1 , pad=SamePad()),
    Conv((5,5) , 5=>1; stride=1 , pad=SamePad()),
)
```

5.2 training

```
using Flux.Optimize
opt_state = Flux.setup(Adam() , model)

@withprogress for epoch in 1:10
   @logprogress train!(loss, model , train_data, opt_state)
end
```

6 References

References

[1] Jaemin Shin, Darae Jeong, and Junseok Kim. "A conservative numerical method for the CahnHilliard equation in complex domains". In: *Journal of Computational Physics* 230.19 (2011), pp. 7441–7455. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2011.06.009. URL: https://www.sciencedirect.com/science/article/pii/S0021999111003585.

[2] Hao Wu. "A review on the CahnHilliard equation: classical results and recent advances in dynamic boundary conditions". In: *Electronic Research Archive* 30.8 (2022), pp. 2788–2832. DOI: 10.3934/era.2022143. URL: https://doi.org/10.3934%2Fera.2022143.