Bachelor Thesis

Jonathan Ulmer

March 21, 2024

Contents

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 extraxt the code to be used independantly. Further details on execution and reading of the original source provided in org-mode format is provided in **??**

1 Cahn-Hilliard equation

is a partial differential equation (PDE) solving the state of a two-phase fluid[2]. The form of the Cahn-Hilliard (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 $I = [-1, 1]$

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

$\varepsilon > 0$ is a constant correlated with boundary thickness and μ is the chemical potential[2]

In this thesis we assume $M(\phi) = 1$, 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 Free energy

The Cahn Hillard Equations can be motivated Using a **Ginzburg Landau** type free energy equation:

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

where $W(\phi)$ denotes the (Helmholtz) free energy density of mixing."" [2] and we approximate it in further calculations as $W(\phi) = \frac{(1-\phi^2)^2}{4}$ like in [1]

The chemical potential then follows as derivative of Energy in respect to time.

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

1.1.2 TODO Derivation by mass balance

We motivate the Cahn Hillard equation as follows: consider

$$\partial_t\phi + \nabla \cdot J = 0 \tag{2}$$

where **J** is mass flux. 3 then states that the change in mass balances the change of the phase field. Using the no-flux boundary conditions:

$$J \cdot n = 0 \qquad \partial\Omega \times (0, T) \tag{3}$$

$$\partial_n\phi = 0 \qquad \partial\Omega \times (0, T) \tag{4}$$

conservation of mass follows see[2].

$$\begin{aligned}
\frac{d}{dt}\int_\Omega \phi &= \int_\Omega \frac{\partial \phi}{\partial t} dV \\
&= -\int_\Omega \nabla \cdot J \; dV \\
&= -\int_{\partial\Omega} J \cdot n \; dA \\
&= 0
\end{aligned} \tag{5}$$

Using:

$$J = -\nabla \mu \tag{6}$$

which conceptionally sets mass flux to equalize the potential energy gradient, leads to the formulation of the CH equations as stated above. Additionally, the boundary conditions evaluate to:

$$\begin{aligned}
-\nabla \mu &= 0 \\
\partial_n \phi &= 0
\end{aligned} \tag{7}$$

i.e. no flow leaves and potential on the border doesn't change. Then for ϕ then follows:

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

hence the Free Energy is decreasing in time.

2 Baseline multi-grid solver

As baseline for further experiments we use a two-grid method based on finite differences as defined in[1].

2.1 Discretization:

it discretizes the phase-field ,ϕ, and chemical potential ,μ, into grid-wise functions ϕ_{ij}, μ_{ij} and defines the partial derivatives $D_x f_{ij},\ D_y f_{ij}$ using differential quotients:

$$D_x f_{i+\frac{1}{2}j} = \frac{f_{i+1j} - f_{ij}}{h} \qquad D_y f_{ij+\frac{1}{2}} = \frac{f_{ij+1} - f_{ij}}{h} \tag{8}$$

for $\nabla f, \Delta f$ then follows:

$$\nabla_d f_{ij} = (D_x f_{i+1j},\ D_y f_{ij+1})$$
$$\Delta_d f_{ij} = \frac{D_x f_{i+\frac{1}{2}j} - D_x f_{i-\frac{1}{2}j} + D_y f_{ij+\frac{1}{2}} - D_y f_{ij-\frac{1}{2}}}{h} = \nabla_d \cdot \nabla_d f_{ij}$$

the authors[1] further adapt the discretized phase-field by the characteristic function of the domain Ω:

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

To simplify notation we use the following abbreviations:

Math	Code
$\Sigma_G f_{ij} = G_{i+\frac{1}{2}j} f_{i+1j}^{n+\frac{1}{2},m} + G_{i-\frac{1}{2}j} f_{i-1j}^{n+\frac{1}{2},m} + G_{ij+\frac{1}{2}} f_{ij+1}^{n+\frac{1}{2},m} + G_{ij-\frac{1}{2}} f_{ij-1}^{n+\frac{1}{2},m}$	`discrete_weigted_neigbou`
$\Sigma_G = G_{i+\frac{1}{2}j} + G_{i-\frac{1}{2}j} + G_{ij+\frac{1}{2}} + G_{ij-\frac{1}{2}}$	`neighbours_in_domain(i,j`

```julia
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, often occurring, modified Laplacian $\nabla_d(G\nabla_d f_{ij})$ as

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

To account for no-flux boundary conditions and arbitrary shaped domains. The authors [1] then define the discrete CH equation adapted for the domain, as:

$$\begin{aligned} \frac{\phi_{ij}^{n+1} - \phi_{ij}^{n}}{\Delta t} &= \nabla_d \cdot (G_{ij}\nabla_d \mu_{ij}^{n+\frac{1}{2}}) \\ \mu_{ij}^{n+\frac{1}{2}} &= 2\phi_{ij}^{n+1} - \varepsilon^2 \nabla_d \cdot (G_{ij}\nabla_d \phi_{ij}^{n+1}) + W'(\phi_{ij}^{n}) - 2\phi_{ij}^{n} \end{aligned} \tag{9}$$

and derive a numerical scheme from these implicit equations.

2.2 Adaptations to the simplified problem

Even tough this work uses rectangular domains, we simplify the adaptation of the algorithm by the domain indicator function, as well as 0 padding, in order to correctly include the boundary conditions of the CH equation. Therefore, the internal representation of the adapted algorithm considers phase-field ,ϕ, and chemical potential field ,μ, as two-dimensional arrays with the shape $(N_x + 2, N_y + 2)$ in order to accommodate padding. Where $\mathrm{N_x}$ and $\mathrm{N_y}$ are the number of steps in x-/y-direction, respectively. Hence, we define the discrete domain function as:

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

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

2.3 PDE as operator L

We derive the iteration operator $L(\phi^{n+1}, \mu^{n+\frac{1}{2}}) = (\zeta^n, \psi^n)$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 8 by separating implicit and explicit terms L and $(\zeta_{ij}^n, \psi_{ij}^n)^T$, respectively.

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

Due to being explicit, we know everything needed to calculate $(\zeta_{ij}^n, \psi_{ij}^n)^T$. We compute those values once for every time step, 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 derivative with respect to the current grid point $(\phi_{ij}^{n+1}, \mu_{ij}^{n+\frac{1}{2}})^T$:

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

```julia
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
```

2.4 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.
2. restriction and prolongation methods between grids $h \leftrightarrow H$.
3. a Newton iteration to solve $L(x,y)_H = L(\bar{x},\bar{y}) + (d_h, r_h)$.

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

```julia
function v_cycle!(grid::Array{T}, level) where T <: Union{multi_solver ,
↪   relaxed_multi_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:

```
set_xi_and_psi!(solver)
```

```julia
pbar = ProgressBar(total = 1000)

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

images/iteration.gif

2.5 SMOOTH operator

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

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

solved for ϕ, μ. SMOOTH consists of point-wise Gauß-Seidel relaxation, by solving L for $\overline{\phi}, \overline{\mu}$ with the initial guess for ζ^n, ψ^n.

$$SMOOTH(\phi_{ij}^{n+1,m}, \mu_{ji}^{n+\frac{1}{2},m}, L_h, \zeta^n, \psi^n) \tag{10}$$

and we implement it as

```julia
function SMOOTH!(
    solver::multi_solver,
    iterations,
    adaptive
)
    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
```

2.6 Test data:

For testing and later training we use a multitude of different phase-fields, notably an assortment of randomly placed circles, squares, and arbitrary generated values.

Size	blobs	blobsize	norm
64	10	10	2
64	10	10	100
512	20	50	2

```
function testdata(gridsize , blobs , radius ,norm)
rngpoints = rand(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.

3 Numerical evaluation

The analytical CH equation conserves mass 3 and energy E_{bulk} decreases **??** in respect to 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:

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

```
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

$$\frac{1}{|\Omega|}\int_{\Omega} \phi \; dx \tag{11}$$

we calculate mass 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, $\phi \equiv -1$.

3.3 Tests

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:

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

Moreover 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 equation1:

$$\begin{aligned} \partial_t \phi^{\alpha} &= \Delta \mu \\ \mu &= \varepsilon^2 \alpha (c^{\alpha} - \phi^{\alpha}) + W'(\phi) \end{aligned} \tag{12}$$

It in turn requires solving the elliptical PDE each time-step to calculate c. We obtain a simpler approach in the numerical solver, with the drawback of having more variables. However those are independent. As ansatz for the numerical solver we propose:

$$\begin{aligned} \frac{\phi_{ij}^{n+1,\alpha} - \phi_{ij}^{n,\alpha}}{\Delta t} &= \nabla_d \cdot (G_{ij} \nabla_d \mu_{ij}^{n+\frac{1}{2},\alpha}) \\ \mu_{ij}^{n+\frac{1}{2},\alpha} &= 2\phi_{ij}^{n+1,\alpha} - \varepsilon^2 a (c_{ij}^{n+1,\alpha} - \phi_{ij}^{n+1,\alpha}) + W'(\phi_{ij}^{n,\alpha}) - 2\phi_{ij}^{n,\alpha} \end{aligned} \tag{13}$$

This approach is inspired by **??** adapted to the relaxed CH equation 8.

4.1 relaxed operators:

We then adapt the multi-grid solver proposed in **??** to the relaxed problem by replacing the differential operators by their discrete counterparts as defined in **??**, and expand them.

4.2 Relaxed PDE as operator L

We reformulate of the iteration in terms of Operator L as follows:

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

```
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 derivative:

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

```
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

Correspondingly the SMOOTH operation expands to:

$$SMOOTH(\phi_{ij}^{n+1,m,\alpha}, \mu_{ji}^{n+\frac{1}{2},m,\alpha}, L_h, \zeta^{n,\alpha}, \psi^{n,\alpha})$$

$$\begin{aligned} -\frac{\Sigma_G}{h^2}\overline{\mu_{ji}^{n+\frac{1}{2},m,\alpha}} &= \frac{\phi_{ij}^{n+1,m,\alpha}}{\Delta t} - \zeta_{ij}^{n,\alpha} - \frac{\Sigma_G \mu_{ij}}{h^2} \\ \varepsilon^2\alpha\overline{\phi_{ij}^{n+1,m,\alpha}} + 2\phi_{ij}^{n+1,m,\alpha} &= \varepsilon^2\alpha c_{ij}^{n,\alpha} - \overline{\mu_{ji}^{n+\frac{1}{2},m,\alpha}} - \psi_{ij}^{n,\alpha} \end{aligned} \tag{14}$$

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. We simplify the relaxed system in one-dimension, and solve explicitly:

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

$\implies$

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

$\implies$

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

$\implies$

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

```
function SMOOTH!(
    solver::relaxed_multi_solver,
    iterations,
    adaptive
)
    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
```

```
using Plots
using LaTeXStrings
using LinearAlgebra
include(pwd() *"/src/utils.jl")
function SMOOTH!(
    solver::relaxed_multi_solver,
    iterations,
    adaptive
)
    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
SIZE =64
M = testdata(SIZE, 5 , 8, 2);
phase = zeros(size(M) .+ 2);
phase[2:end-1,2:end-1] = M;
mu = copy(phase);
W_prime(x) = -x * (1-x^2)
function elyps_solver!(solver::relaxed_multi_solver, n)
    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
solver = relaxed_multi_solver(
    phase ,
    zeros(size(phase)) ,
    zeros(size(phase)) ,
    zeros(size(phase)) ,
    zeros(size(phase)) ,
    8e-3 ,1e-3 , 1e-3 ,
    W_prime ,
    size(M , 1) , size(M , 2),
    1000001
)
set_xi_and_psi!(solver)
elyps_solver!(solver , 2000)
SMOOTH!(solver, 1000, true);
p2 = heatmap(solver.phase, aspect_ratio=:equal, title="with solving c" ,
↪   xlim=(2,SIZE) , ylim=(2,SIZE));
savefig(p2,"images/smooth_relaxed.svg")
```

4.4 Relaxed V-cycle approach

As the difference between both methods is abstracted away in the operators, the relaxed V-cycle is identical to the original counterpart and therefore reused. The only additional step is solving the elliptical equation:

```

set_xi_and_psi!(solver)

pbar = ProgressBar(total = 1000)

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

images/iteration_relaxed2.gif

4.5 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.

4.5.1 Discretization

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

$$-\nabla_d \cdot (G_{ij} \nabla_d c^{\alpha}_{ij}) + \alpha c^{\alpha}_{ij} = \alpha \phi^{\alpha}_{ij}$$

$\implies$

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

$\implies$

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

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 $\Sigma_G = G_{i+\frac{1}{2}j} + G_{i-\frac{1}{2}j} + G_{ij+\frac{1}{2}} + G_{ij-\frac{1}{2}}$. Then the discrete elliptical PDE can be stated as:

$$-\frac{\Sigma_G c^{\alpha}_{ij}}{h^2} + \frac{\Sigma_G}{h^2} c^{\alpha}_{ij} + \alpha c^{\alpha}_{ij} = \alpha \phi^{\alpha}_{ij} \tag{15}$$

1. **DONE** Proposal1 Newton Solver And then we propose a simple newton Iteration to solve 13 for $x = c^{\alpha}_{ij}$: Let F, dF be:

   $$F(x) = -\frac{\Sigma_G c^{\alpha}_{ij}}{h^2} + \frac{\Sigma_G}{h^2} x + \alpha x - \alpha \phi^{\alpha}_{ij}$$

   and $dF(x)$

   $$dF(x) = -\frac{\Sigma_G}{h^2} + \alpha$$

   the implementation then is the following:

   as input, we use :

2. Proposal2 solver solving 13 for c_{ij}^{α} then results in.

$$\left(\frac{\Sigma_G}{h^2} + \alpha\right) c_{ij}^{\alpha} = \alpha\phi_{ij}^{\alpha} + \frac{\Sigma_G c_{ij}^{\alpha}}{h^2}$$

and can be translated to code as follows

```
function elyps_solver!(solver::relaxed_multi_solver, n)
    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
```

5 Technical details

We are writing this thesis in org-mode file format.

6 References

References

[1] Jaemin Shin, Darae Jeong, and Junseok Kim. "A conservative numerical method for the Cahn–Hilliard 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 Cahn–Hilliard 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.