Bachelor Thesis

Jonathan Ulmer

March 10, 2024

Contents

1 Utility functions

```
include(pwd() * "/" * "utils.jl")
```

```
################################################################################
#                  Common Utility Functions For Multi Solvers                  #
################################################################################

"""
restricts an array on the small grid to an array in the large grid asserts size arr=2^n + 2 and returns ret=2^(n-1) + 2

Returns
---------------------------
large grid array + padding
"""
function restrict(arr, G)
    shape = (size(arr) .- 2) .÷ 2
    ret = zeros(shape .+ 2)
    for I in CartesianIndices(ret)[2:end-1, 2:end-1]
        i, j = I.I
        g = [
            G(2 * i - 1, 2 * j - 1, (size(arr) .- 2)...),
            G(2 * i - 1, 2 * j, (size(arr) .- 2)...),
            G(2 * i, 2 * j - 1, (size(arr) .- 2)...),
            G(2 * i, 2 * j, (size(arr) .- 2)...)
        ]
        if sum(g) == 0
            ret[I] = 0
        else
            ret[I] = (
                1 / sum(g)
                *
                dot(g,
                    [
                        arr[2*i-1, 2*j-1],
                        arr[2*i-1, 2*j],
                        arr[2*i, 2*j-1],
                        arr[2*i, 2*j]
                    ]
                )
            )
        end
```

```
    end
    return ret
end

"""
    prolong(arr , G)

interpolates int a smaller grid by a factor of 2

"""
function prolong(arr, G)
    inner_shape = (size(arr) .- 2) .* 2
    ret = zeros(inner_shape .+ 2)
    ONE = oneunit(CartesianIndices(arr)[1])
    for I in CartesianIndices(arr)[2:end-1, 2:end-1]
        Ind = 2 * (I - ONE) + ONE
        for J in (Ind-ONE):Ind
            ret[J] = G(J.I..., inner_shape...) * arr[I]
        end
    end
    return ret
end

struct multi_solver
    phase::Matrix{Float64}
    potential::Matrix{Float64}
    xi::Matrix{Float64}
    psi::Matrix{Float64}
    epsilon::Float64
    h::Float64
    dt::Float64
    W_prime::Function
    len::Int
    width::Int

end
struct relaxed_multi_solver
    phase::Matrix{Float64}
    potential::Matrix{Float64}
    xi::Matrix{Float64}
    psi::Matrix{Float64}
    c::Matrix{Float64}
    epsilon::Float64
    h::Float64
    dt::Float64
    W_prime::Function
    len::Int
    width::Int
```

```
    alpha::Float64

end

function testgrid(M, len)
    grid = Array{multi_solver}(undef, len)
    phase = zeros(size(M) .+ 2)
    phase[2:end-1, 2:end-1] = M
    W_prime(x) = -x * (1 - x^2)
    h0 = 3e-3


    for i = 1:len
        grid[i] = multi_solver(zeros(size(M) .÷ i .+ 2),
            zeros(size(M) .÷ i .+ 2),
            zeros(size(M) .÷ i .+ 2),
            zeros(size(M) .÷ i .+ 2),
            8e-3, h0 * 2^i, 1e-3,
            W_prime,
            size(M, 1) ÷ i, size(M, 2) ÷ i)

    end
    copyto!(grid[1].phase, phase)
    return grid

end

function relaxed_testgrid(M, len)
    grid = Array{relaxed_multi_solver}(undef, len)
    phase = zeros(size(M) .+ 2)
    phase[2:end-1, 2:end-1] = M
    W_prime(x) = -x * (1 - x^2)
    h0 = 3e-3

    for i = 1:len
        grid[i] = relaxed_multi_solver(zeros(size(M) .÷ i .+ 2),
            zeros(size(M) .÷ i .+ 2),
            zeros(size(M) .÷ i .+ 2),
            zeros(size(M) .÷ i .+ 2),
            zeros(size(M) .÷ i .+ 2),
            8e-3, h0 * 2^i, 1e-3,
            W_prime,
            size(M, 1) ÷ i, size(M, 2) ÷ i,
            1000001)

    end
    copyto!(grid[1].phase, phase)
    return grid
```

```
end

"""
    restrict!(smallgrid_solver::multi_solver ,
    ↪  largegrid_solver::multi_solver)::multi_solver

------------
Requires
----------
smallgrid solver and largegid solvers to be multiple of 2 from each other
↪  bar padding eg. (66x66)->(34x34)

------------
Returns
------------
    nothing. mutatest largegid in place to represent the smallgrid

"""
function restrict_solver!(smallgrid_solver::T, largegrid_solver::T) where
↪  {T<:Union{multi_solver,relaxed_multi_solver}}
    copy!(largegrid_solver.phase, restrict(smallgrid_solver.phase, G))
    copy!(largegrid_solver.potential, restrict(smallgrid_solver.potential,
    ↪  G))
    return nothing
end
```

2 Cahn Hillard Equation Overview

Partial Differential Equation (PDE) solving the state of a 2 Phase Fluid[2]. The form of the Cahn Hillard Equation used for the remainder of this thesis is: where ϕ is the so-called phase field. Demarking the different states of the fluids through an Interval $I = [-1, 1]$ and where $\partial I = \{-1, 1\}$ represents full state of one fluid. $\varepsilon > 0$ is a positive constant

$$\phi_t(x,t) = \Delta\mu \tag{1}$$

$$\mu = -\varepsilon^2\Delta\phi + W'(\phi) \tag{2}$$

, and μ is the chemical potential[2]. While the Cahn Hillard exist in a more general form taking the fluid's mobility $M(\Phi)$ into account, we will assume $M(\Phi) = 1$, simplifying the CH-Equations used in [2] [1] to what is stated above.

The Advantages of the Cahn Hillard Approach as compared to traditional fluid dynamics solvers 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

2.1 TODO Derivation from paper

2.1.1 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 will be approximated in further calculations as $W(\phi) = \frac{(1-\phi^2)^2}{4}$ as used in[1]

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

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

2.1.2 Derivation by mass balance

The Cahn Hillard equation then can be motivated as follows: consider

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

where **J** is mass flux. 3 then states that the change in mass balances the change of the phasefield. Using the no-flux boundry 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].

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:

$$-\nabla\mu = 0$$

$$\partial_n\phi = 0$$

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

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

hence the Free Energy is decreasing in time.

3 Baseline Multigrid solver:

As baseline for further experiments a two grid method based on finite differences by[1]. Is used.

3.1 Discretization:

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

$$D_x f_{i+\frac{1}{2}j} = \frac{f_{i+1j} - f_{ij}}{h} \qquad D_y f_{ij+\frac{1}{2}} = \frac{f_{ij+1} - f_{ij}}{h} \tag{7}$$

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

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

the authors further adapt the discretized phasefield 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 the following abbreviations are used:

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`

```
"""
    neighbours_in_domain(i, j, G, len, width)

TBW
counts neigbours in domain
"""
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

"""
    discrete_G_weigted_neigbour_sum(i, j, arr, G, len, width)

TBW
--------
sums all neighbours depending on wheter tey are in the domain determined
↪  by G
"""
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
```

The for the solver necessary modified Laplacians $\nabla_d(G\nabla_d f_{ij})$then are written 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 boundry conditions and arbitrary shaped domains. The authors [1] then define the discrete CH Equation adapted for Domain, as:

$$\frac{\phi_{ij}^{n+1} - \phi_{ij}^{n}}{\Delta t} = \nabla_d \cdot (G_{ij}\nabla_d \mu_{ij}^{n+\frac{1}{2}}) \tag{8}$$

$$\mu_{ij}^{n+\frac{1}{2}} = 2\phi_{ij}^{n+1} - \varepsilon^2 \nabla_d \cdot (G_{ij}\nabla_d \phi_{ij}^{n+1}) + W'(\phi_{ij}^{n}) - 2\phi_{ij}^{n} \tag{9}$$

and derive from these implicit equations a numerical scheme.

3.2 adaptations to the simplified problem

even tough this work uses rectangular domains, the adaptation of the algorithm is simplified 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 phasefield and potential field ϕ, μ as 2D arrays of shape $(N_x + 2, N_y + 2)$ in order to accommodate padding. Where N_x and N_y are the number of steps in x-/y-Direction respectively. Hence, we define the discrete domain function as:

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

```
"""
Boundry indicator function

Returns
---------------
1 if index i,j is in bounds(without padding) and 0 else
"""
function G(i, j, len, width)
    if 2 <= i <= len + 1 && 2 <= j <= width + 1
        return 1.0
    else
        return 0.0
    end
end
```

3.3 PDE as Operator

and derive the iteration operator $L(\phi^{n+1}, \mu^{n+\frac{1}{2}}) = (\zeta^n, \psi^n)$

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

```
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 an d $(\zeta_{ij}^n, \psi_{ij}^n)^T$ respectivly.

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

Due to being explicit, everything needed to calculate $(\zeta_{ij}^n, \psi_{ij}^n)^T$ is known. Those values are commuted once for every time step, and stored.

```
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 in respect to the current grid point $(\phi_{ij}^{n+1}, \mu_{ij}^{n+\frac{1}{2}})^T$:

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

```
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
```

3.4 v-cycle

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

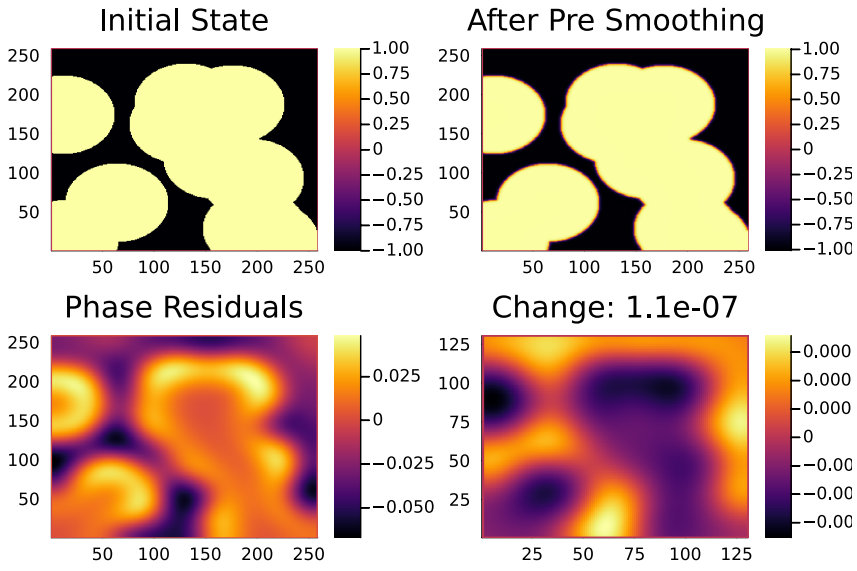
1. A Gaus-Seidel Relaxation for Smoothing
2. restriction and prolongation methods between grids $h \leftrightarrow H$
3. a Newton Iteration to solve $L(x,y)_H = L(\bar{x},\bar{y}) + (d_h, r_h)$

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

```
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 have a closer look at the internals, namely the phasefield 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 corsest level.



and a few iterations of the V-cycle exhibit the following behaviour:

images/iteration.gif

3.5 SMOOTH Operator

Gaus-Seidel Smoothing is derived by

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

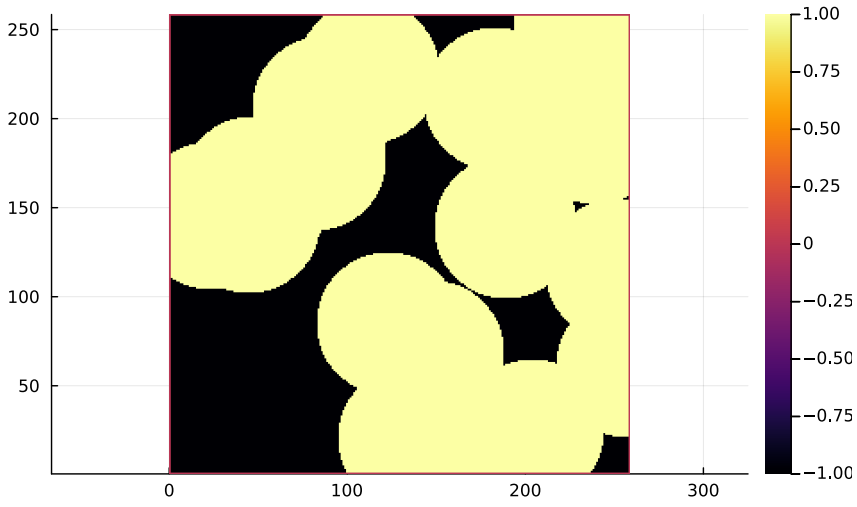
solved for ϕ, μ. SMOOTH consists of point-wise Gauß Seidel Relaxation, by solving L for $\overline{\phi}, \overline{\mu}$ with the initial guess for ζ^n, ψ^n.

$$SMOOTH \tag{10}$$

and is implemented as:

```
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
```



3.6 Test Data:

For testing and later training, a multitude o different phasefields where used. 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
```

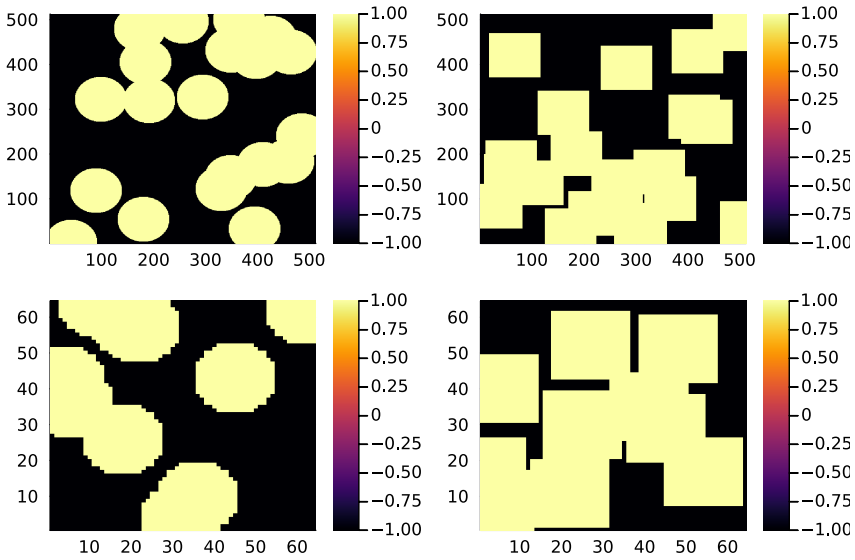


Figure 1: Examples of different phasefields used as initial condition later on

4 Numerical Evaluation

The analytical Chan-Hilliard equation exhibits mass conservation 3 and a decrease in Energy E_{bulk} **??**. Therefore dicretisations of those concepts are used as necessary conditions for a good solution. Furthermore, since E_{bulk} is closely correlated with potential Energy μ, we evauate this diffrence as quality if convergence.

4.1 Energy Evaluations

As discrete energy measure we use

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

```
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
```

4.2 Massbalance

massbalance is calculated by

4.3 Tests

5 Relaxed Problem

In effort to decrease the order of complexity, the following relaxation to the classical Cahn Hillard Equation is proposed:

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

that in turn requires solving an additional PDE each time-step to calculate c. c is the solution of the following elliptical PDE:

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

5.1 TODO relaxed operators:

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

5.1.1 L Relaxed

for the reformulation of the iteration in terms of Operator L then follows:

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

```
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 relaxed derivaitve

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

```
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
```

5.1.2 SMOOTH

and correspondingly the SMOOTH operation expands to:

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

$$-\frac{\Sigma_G}{h^2}\overline{\mu}_{ji}^{n+\frac{1}{2},m} = \frac{(\phi_{ij}^{n+1,m})^\alpha}{\Delta t} - \zeta_{ij}^n - \frac{\Sigma_G \mu_{ij}}{h^2} \tag{11}$$

$$\varepsilon^2\alpha(\overline{\phi}_{ij}^{n+1,m})^\alpha + 2\phi_{ij}^{n+1,m} = \varepsilon^2\alpha c^\alpha - \mu_{ji}^{n+\frac{1}{2},m} - \psi_{ij} \tag{12}$$

1. Proposal1 Since the resulting system no longer is linear, (albeit simpler in Dimension), we propose a newton method to solve second equation (in conjunction with the first one) hopefully solving this converges faster than the original multiple SMOOTH Iterations. The iteration solves for $(\phi_{ij}^{n+1,m})^\alpha = x$ as free variable. Therefore, it follows for $F(x)$

$$\begin{aligned} F(x) &= \varepsilon^2 x^\alpha + 2x - \varepsilon^2 c^\alpha + y + \psi_{ij} \\ y &= \frac{x}{\Delta t} - \zeta_{ij}^n \\ &\quad - \frac{1}{h^2}\left(G_{i+\frac{1}{2}j}\mu_{i+1j}^{n+\frac{1}{2},m} + G_{i-1j}\mu_{i-1j}^{n+\frac{1}{2},m} + G_{ij+1}\mu_{ij+1}^{n+\frac{1}{2},m} + G_{ij-1}\mu_{ij-1}^{n+\frac{1}{2},m}\right) \\ &\quad \cdot (G_{i+1j} + G_{i-1j} + G_{ij+1} + G_{ij-1})^{-1} \end{aligned}$$

   And the derivative for the iteration is

$$\begin{aligned} \frac{d}{dx}F(x) &= \alpha\varepsilon^2 x^{\alpha-1} + 2 + \frac{d}{dx}y \\ \frac{d}{dx}y &= \frac{1}{\Delta t} \end{aligned}$$

2. Proposal2 solve for $\overline{\mu_{ij}^{n+1,m}}$ and $(\overline{\phi_{ij}^{n+1,m}})^\alpha$. This was not done in the original paper as the there required System of linear equations was

solved numerically. The relaxation simplifies the it to one dimension, and enables explicit solutions:

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

$$\Longrightarrow$$

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

$$\Longrightarrow$$

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

solved for $\phi_{ij}^{n+1\alpha}$ and the Smoothing Operator then follows:

```
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))

            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() *"/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))

            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)
using ProgressBars

"""
    elyps_solver(c,
    phase,
    len,
    width,
    alpha,
    h,
    n
)

TBW
"""
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")
```

5.1.3 Relaxed V-cycle

As The diffrence between both methods is abstracted away in the operators, the relaxed V-cycle Is Identical to its original counterpart. And therefore reused. testing: `images/iteration_relaxed2.gif`

5.2 Elliptical PDE:

on order to solve the relaxed CH Equation the following PDE as to be solved in Each additional time step: or in terms of the characteristic function:

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

Similarly to the first solver this PDE is solved with a finite difference scheme using the same discretisations as before:

5.2.1 Discretization

the Discretization of the PDE expands the differential opperators in the same way and proposes an equivalent scheme for solving.

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

$\Longrightarrow$

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

$\Longrightarrow$

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

As before we abbreviate $\Sigma_G c_{ij}^{\alpha} = G_{i+\frac{1}{2}j} c_{i+1j}^{\alpha} + G_{i-\frac{1}{2}j} c_{i-1j}^{\alpha} + G_{ij+\frac{1}{2}} c_{ij+1}^{\alpha} + G_{ij-\frac{1}{2}} c_{ij-1}^{\alpha}$ 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 elyptical PDE can be stated as:

$$-\frac{\Sigma_G c_{ij}^{\alpha}}{h^2} + \frac{\Sigma_G}{h^2} c_{ij}^{\alpha} + \alpha c_{ij}^{\alpha} = \alpha \phi_{ij}^{\alpha} \tag{13}$$

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

   $$F(x) = -\frac{\Sigma_G c_{ij}^{\alpha}}{h^2} + \frac{\Sigma_G}{h^2} x + \alpha x - \alpha \phi_{ij}^{\alpha}$$

   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

```
using ProgressBars

"""
    elyps_solver(c,
    phase,
    len,
    width,
    alpha,
    h,
    n
)

TBW
"""
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
```

3. Proposal 4 as the solver still exhibits unexpected behaviour, ie. it doesn't seem to converge wit higher iterations, we propose a relaxation by interpolating the new value of c_{ij}^{α} with the old one

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.