nty/global//global/global

Bachelor Thesis

Jonathan Ulmer

February 4, 2024

Contents

1 Utility functions

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:

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

hence the Free Energy is decreasing in time.

3 Baseline Multigrid solver:

As baseline for further experiments a multi 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: $\Sigma_G f_{ij} = G_{i+\frac{1}{2}j} f_{i+1j}^{n+\frac{1}{2},m} + G_{i-\frac{1}{2}j} f_{i-1j}^{n+\frac{1}{2},m} + G_{ij+\frac{1}{2}} f_{ij+1}^{n+\frac{1}{2},m} + G_{ij-\frac{1}{2}} f_{ij-1}^{n+\frac{1}{2},m}$ Code: `discrete_weigted_neigbour_sum(i,j,...)` and Math: $\Sigma_G = G_{i+\frac{1}{2}j} + G_{i-\frac{1}{2}j} + G_{ij+\frac{1}{2}} + G_{ij-\frac{1}{2}}$ Code: `neighbours_in_domain(i,j,G)` the expansion of $\nabla_d \cdot G_{ij} \nabla_d f_{ij} = \Sigma_G f_{ij} - \Sigma_G \cdot f_{ij}$. To account for boundry conditions and arbitrary shaped domains. The authors [1] then define the discrete CH Equation adapted for Domain, as:

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

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 $\mathrm{N_x}$ and $\mathrm{N_y}$ are the number of steps in x-/y-Direction respectively. Hence, we define the discrete domain function as:

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

3.3 PDE as Operator

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

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

implented as

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

.

Furthermore, as it enabled a Newtorn iteration we state its derivative in respect to the current gridpoint $(i,j)^T$ in as:

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

implemented:

```
function dL(solver::multi_solver , i , j)
    return [ 1/solver.dt neighbours_in_domain(i,j,G);
             (neighbours_in_domain(i,j,G) - 2) 1]
    end
```

initialized as

$$(\zeta^n, \psi^n) = \begin{pmatrix} \frac{\phi_{ij}^{n+1}}{\Delta t} \\ W'(\phi_{ij}^n) - 2\phi_{ij}^n \end{pmatrix}$$

```
function set_xi_and_psi(phase , len , width , dt , W_prime)
    xi = zeros(len + 2 , width + 2)
    psi = zeros(len + 2 , width + 2)
    xi_init(x) = x / dt
    psi_init(x) = W_prime(x) - 2 * x
    xi[2:end-1, 2:end-1] = xi_init.(phase[2:end-1,2:end-1])
    psi[2:end-1, 2:end-1] = psi_init.(phase[2:end-1,2:end-1])

    (xi , psi)
end
```

the algorithm is then defined as:

Wherein 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.

3.4 SMOOTH Operator

$$SMOOTH \tag{8}$$

and is implemented as:

```
function SMOOTH(
    xi,
    psi,
    phase,
    mu,
    epsilon,
    h,
    dt,
    len,
    width,
    iterations,
    adaptive
)
    for k =  ProgressBar(1:iterations)
        old_phase = copy(phase)
        for i = 2:(len + 1)
            for j = 2:(width + 1)
                bordernumber = neighbours_in_domain(i, j, len, width)
                coefmatrix =
                    [
                        (1 / dt)  (bordernumber / h^2) ;
                        (-1 * (2 + (epsilon^2 / h^2) * bordernumber))  1
                    ]


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

                res = coefmatrix \ b
                phase[i, j] = res[1]
                mu[i, j] = res[2]

            end
        end

        if adaptive && LinearAlgebra.norm(old_phase - phase) < 1e-8
            print("SMOOTH terminated at $(k) succesfully")
            break
        end
    end
     (phase, mu)
    end
```

3.5 tests$_{\mathbf{data}}$:

For testing and later training, a multitude o different phasefields where used. Notably an assortment of randomly placed circles, squares, and arbitrary generated values

```
function testdata(gridsize , blobs , radius ,norm)
rngpoints = rand(1:gridsize, 2, 10)
M = zeros(gridsize,gridsize) .- 1
```

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

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

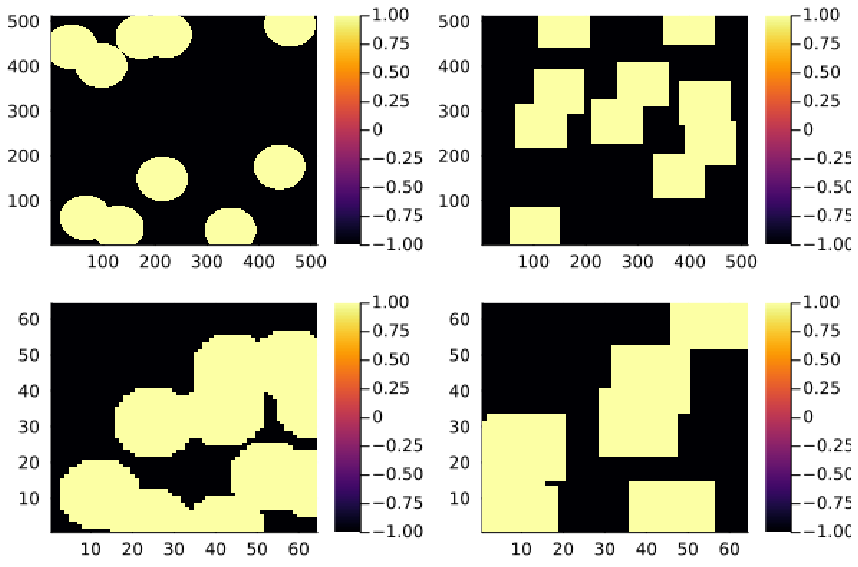


Figure 1: Examples of different phasefields used as initial condition later on

4 Relaxed Problem

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

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

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

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

4.1 TODO relaxed operators:

the multi-grid solver proposed earlier is then adapted to the relaxed Problem by replacing the differential operators by their discrete counterparts as defined in **??** and expanding them

4.1.1 L Relaxed

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

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

4.1.2 SMOOTH

and correspondingly the SMOOTH operation expands to:

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

$$\frac{1}{h^2}\left(G_{i+\frac{1}{2}j}+G_{i-\frac{1}{2}j}+G_{ij+\frac{1}{2}}+G_{ij-\frac{1}{2}}\right)\overline{\mu}_{ji}^{n+\frac{1}{2},m} = \frac{(\phi_{ij}^{n+1,m})^\alpha}{\Delta t} - \zeta_{ij}^n$$
$$- \frac{1}{h^2}($$
$$G_{i+\frac{1}{2}j}\mu_{i+1j}^{n+\frac{1}{2},m}$$
$$+ G_{i-\frac{1}{2}j}\mu_{i-1j}^{n+\frac{1}{2},m}$$
$$+ G_{ij+\frac{1}{2}}\mu_{ij+1}^{n+\frac{1}{2},m}$$
$$+ G_{ij-\frac{1}{2}}\mu_{ij-1}^{n+\frac{1}{2},m}$$
$$)$$

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

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 $F(x)$

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

   And the derivative for the iteration is

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

2. Proposal2 solve analytically 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 analytical solutions:

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

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

$\Longrightarrow$

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

$\Longrightarrow$

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

4.2 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:

4.2.1 Discretization

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

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

$\Longrightarrow$

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

$\Longrightarrow$

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

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

$$
-\frac{\Sigma_G c^\alpha_{ij}}{h^2} + \frac{\Sigma_G}{h^2}c^\alpha_{ij} + \alpha c^\alpha_{ij} = \alpha\phi^\alpha_{ij} \tag{9}
$$

1. Proposal1 Newton Solver And then we propose a simple newton Iteration to solve 9 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 $dF(x)$

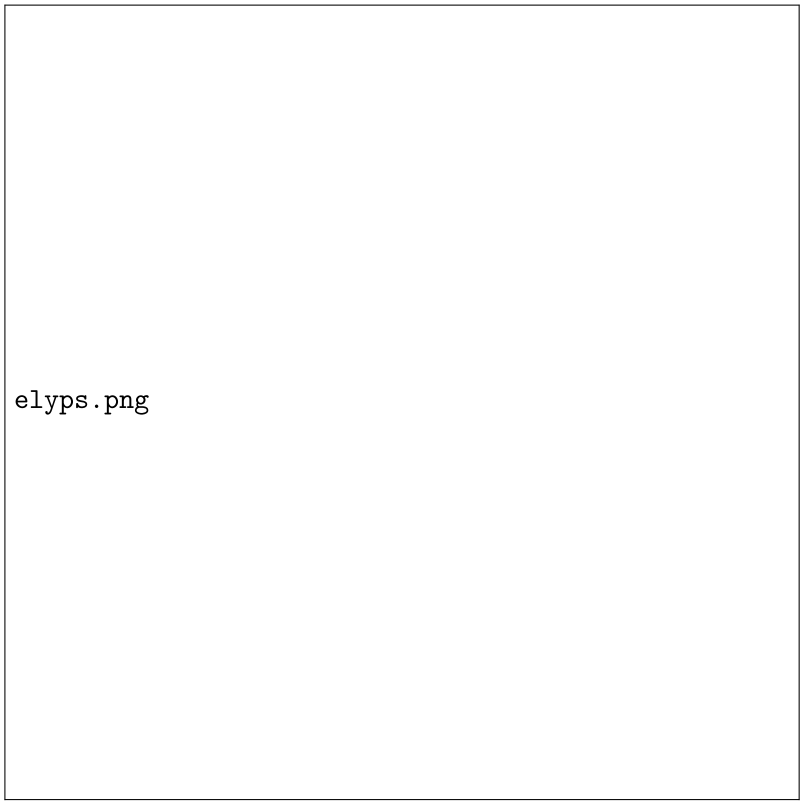
$$
dF(x) = -\frac{\Sigma_G}{h^2} + \alpha
$$

the implementation then is the following:

```python
from numba import njit
from numpy.typing import NDArray
import numpy as np
from multi_solver import neighbours_in_domain ,
↪  discrete_G_weigted_neigbour_sum , __G_h

@njit
def elyptical_PDE_solver(
    c: NDArray[np.float64],
    phase: NDArray[np.float64],
    len: int,
    width: int,
    alpha: float,
    h: float,
    n: int,
) -> NDArray[np.float64]:
    """
    solves elyptical equation
    """
    maxiter = 10000
    tol = 1.48e-4
    for k in range(n):
        for i in range(1, len + 1):
            for j in range(1, width + 1):
                bordernumber = neighbours_in_domain(i, j, len,
                ↪   width)
                x = c[i, j]
                for iter in range(maxiter):
                    F = (
                        -1
                        * h**-2
                        * discrete_G_weigted_neigbour_sum(i, j, c,
                        ↪   __G_h, len, width)
                        + h**-2 * bordernumber * x
                        + alpha * x
                        - alpha * phase[i, j]
                    )

                    dF = alpha + h**-2 * bordernumber

                    if dF == 0:
                        continue

                    step = F / dF
                    x = x - step
                    if abs(step) < tol:
                        break
                c[i, j] = x
    return c
```

as input we use :

```

from multi_solver_relaxed import CH_2D_Multigrid_Solver_relaxed ,
↪   test_solver , plot

test_phase = tu.k_spheres_phase(15, 10, size=64)
t = test_solver(test_phase)
t.elyps_solver = elyptical_PDE_solver
t.solve_elyps(100)
sns.heatmap(t.c)
plt.plot()
```



2. Proposal2 Analytical solver solving 9 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

```
@njit
def elyps_solver(
    c: NDArray[np.float64],
    phase: NDArray[np.float64],
    len: int,
    width: int,
    alpha: float,
    h: float,
    n: int,
) -> NDArray[np.float64]:
    for k in range(n):
        for i in range(1, len + 1):
            for j in range(1, width + 1):
                bordernumber = neighbours_in_domain(i, j, len,
                ↪  width)

                c[i, j] = (
                    -1* alpha * phase[i, j]
                    + discrete_G_weigted_neigbour_sum(i, j, c,
                    ↪  __G_h, len, width)
                    / h**2
                ) / (bordernumber / h**2 + alpha)
    return c
```

and looks like

```

from multi_solver_relaxed import CH_2D_Multigrid_Solver_relaxed ,
↪  test_solver , plot

test_phase = tu.k_spheres_phase(15, 10, size=64)
t = test_solver(test_phase)
t.elyps_solver = elyps_solver
t.alpha = 1001
t.solve_elyps(100)
sns.heatmap(t.c)
plt.plot()
```

```

from multi_solver_relaxed import CH_2D_Multigrid_Solver_relaxed ,
↪  test_solver , plot

test_phase = tu.k_spheres_phase(15, 10, size=64)
t = test_solver(test_phase)
t.elyps_solver = elyps_solver
t.alpha = 1001
t.solve_elyps(40)
prev  = np.array(t.c)
t.solve_elyps(1)
sns.heatmap(t.c - prev)
plt.plot()
```

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

```
@njit
def elyps_solver(
    c: NDArray[np.float64],
    phase: NDArray[np.float64],
    len: int,
    width: int,
    alpha: float,
    h: float,
    n: int,
    delta = 0.9
) -> NDArray[np.float64]:
    for k in range(n):
        for i in range(1, len + 1):
            for j in range(1, width + 1):
                bordernumber = neighbours_in_domain(i, j, len,
                ↪   width)

                c_new = (
                    alpha * phase[i, j]
                    + discrete_G_weigted_neigbour_sum(i, j, c,
                    ↪   __G_h, len, width)
                    / h**2
                ) / (bordernumber / h**2 + alpha)
                c[i,j] = c[i,j] * delta + (1-delta) * c_new

    return c
```

```
from multi_solver_relaxed import CH_2D_Multigrid_Solver_relaxed ,
↪   test_solver , plot

test_phase = tu.k_spheres_phase(15, 10, size=64)
t = test_solver(test_phase)
t.elyps_solver = elyps_solver
t.alpha = 100000001
t.solve_elyps(1000)
sns.heatmap(t.c)
plt.plot()
```

5 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`.