FAU-Friedrich Alexander Universität
Institute of Materials Simulation (WW8), Prof. Zaiser
Anastasia Papadaki
vi45supa

Phase-field method: basics and application in materials science

Answer to Task 1

Phase field is a simulation approach applied for solid- liquid phase transition and dislocation evaluation. It provides primary experience on phase field method. In the application of the simulation in solid- liquid phase transition, partial derivatives are applied to obtain the desired parameters such as models, structure, energy possessed and other required parameters of the phase field.
In the finite volume python package (Fipy) which it is a finite volume solver for PDEs written in python which is expressed in standard finite volume the program. This approach enables the determination of the volume through the program alongside its accrued parameters.
The general partial quantifier differential equation takes the form,

$$\frac{\partial \rho\phi}{\partial t} = -\nabla u(\phi) + \nabla_\Gamma \nabla\phi + \alpha\phi^2 + b\phi + C$$

Under single component solidification the following terms are defined to easen the administration of the non- conserved order parameter. Where:

- $\phi = 0$, for liquid
- $\phi = 1$, for solid

This implies that,
The Functional Energy, F will be given by:

- $F = \int (f_0\phi^2(1-\phi^2) + \frac{K_\phi}{2}\left|\nabla\phi^2\right|)\ \partial r$ (*)

From the general format of the analytical expression with:

- $\frac{\partial\phi}{\partial t} = -L\frac{\partial F}{\partial\phi}$

Where,

- $f_0 = 1$
- $K_\phi = 1$
- L=1
- $\partial_x = 1$
- $\partial_t = 1$

By keeping, $\partial x = 1$, $\partial t = 1$ and changing f_0, K_ϕ and L one at a time, and using the theory of the functional derivative, the analytical expression of $\frac{\partial F}{\partial\phi}$ will be given by:
Let ϕ_0, ϕ and $t \in \mathbb{R}$:

$$\frac{\partial\phi}{\partial t} = -L\frac{\partial F}{\partial\phi} = -L[-K_\phi\Delta\phi + f_0(4\phi^3 - 6\phi^2 + 2\phi)] \tag{1}$$

So, by using the theory of the above analytical equation can be substituted for any parameter being calculated since the Hamilton term represents any quantity being calculated.
Now, as it concerns the terms f_0, K_ϕ, and L. Reusing the codes in Fipy tutorial and section 4, we can determine the following statements:

- By changing the f_0, we can define the difference values of the local free energy density in our system. Also, changes on the value of f_0 mean different different Δf values. By changing f_0, e change the Δf, and so the bulk energy density.
- By changing the K_ϕ, we can determine differences in interface thickness and numeric stable, and at the same time insurance that the numeric interface energy roughly agree with experimental interface energy.
- By changing the L, we can define the interface mobility coefficient. Different values of L can affect the kinetic energy of the system but also how fast our system will reach the equilibrium.

Also, by keeping $dx = 1$ and $dt = 1$, and changing f_0, K_ϕ, and L one by one, we took the following plots:

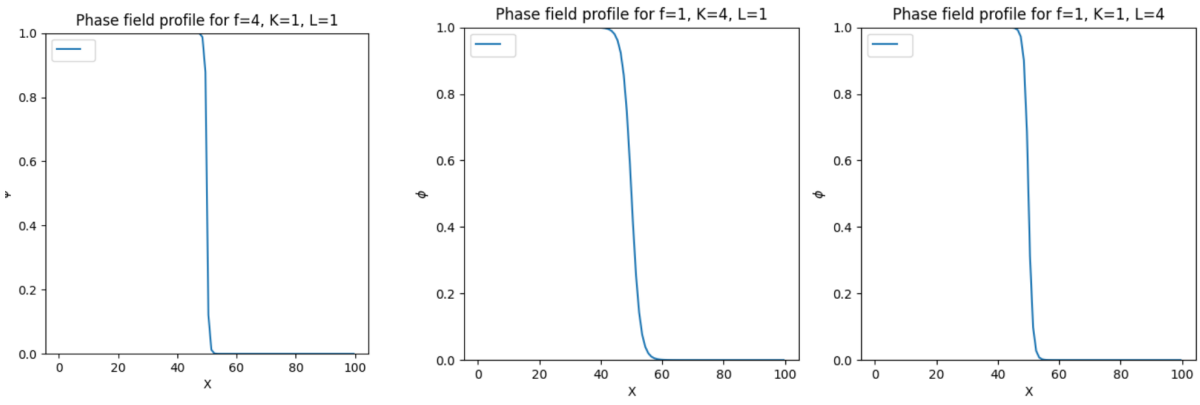

Figure 1: The different plots that we take if we keep $dx = 1$ and $dt = 1$, change f_0, K_ϕ, and L one by one

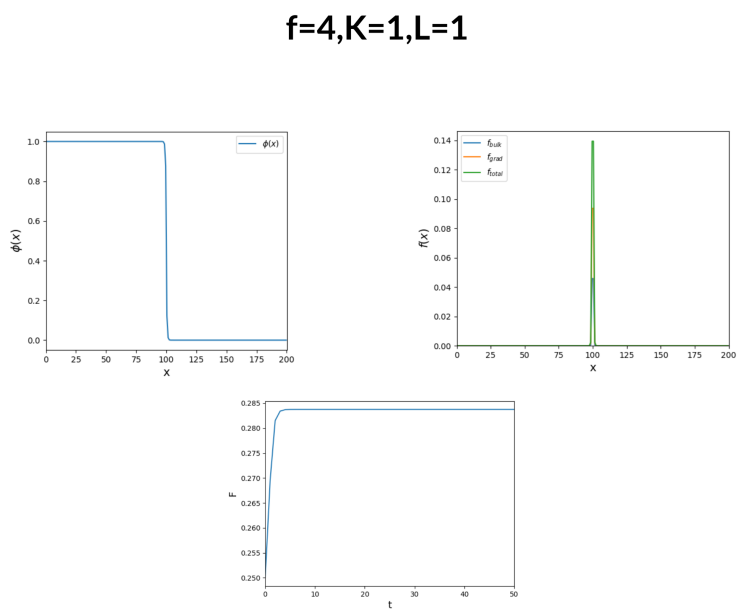

Figure 2: The different plots that we take if we keep $dx = 1$ and $dt = 1$, change f_0=4 and keep $K_\phi = L = 1$.

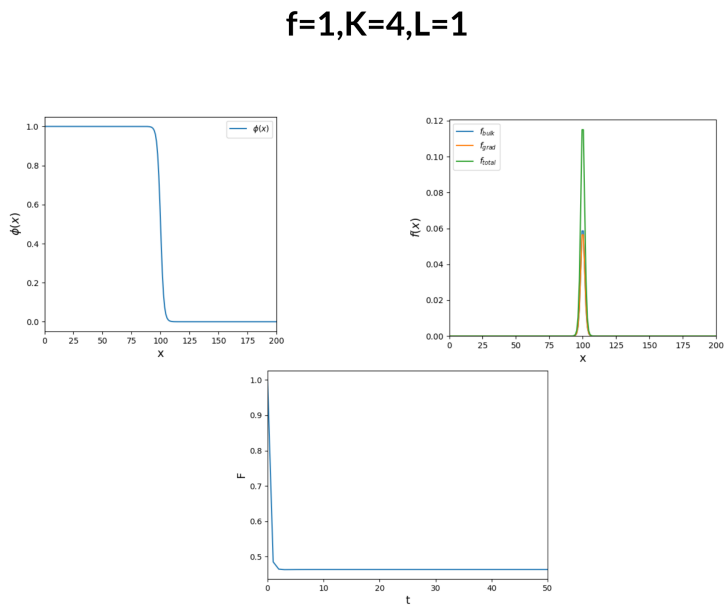

Figure 3: The different plots that we take if we keep $dx = 1$ and $dt = 1$, change K_ϕ=4 and keep $f_0 = L = 1$.

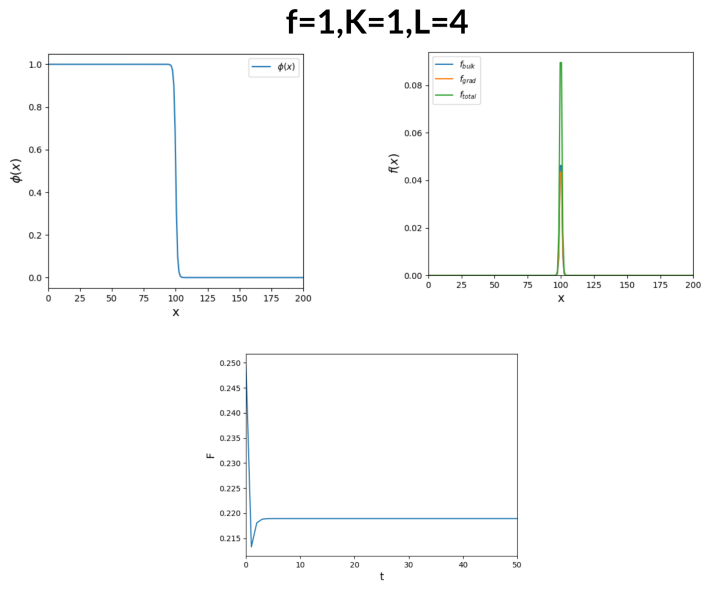

Figure 4: The different plots that we take if we keep $dx = 1$ and $dt = 1$, change L=4 and keep $f_0 = K_\phi$= 1.

Answer to Task 2

Determination of bulk energy density coefficient, f_0.

This will be a rough value of f_0.

Considering the binary system of $Ni - Al$ with bi-phases Γ and Γ'.

Using the equation
We know that the bulk energy density is constructed as:

$$\boxed{\mathrm{f}_{bulk} = f_0 = (c^e_{\gamma'} - c)^2(c - c^e_{\gamma})^2} \quad (23)$$

We can understand from the figure that:

- C= 0.195 $\simeq$ 150 J/mol is the concentration of the Ni- Al electrolyte.
- T= 1300 K
- Δf = 149.9 J/mol
- $c^e_{\gamma'} = 0.16$
- $c^e_{\gamma} = 0.23$

Then, we will replace these values in the Eq. (23). So we will have:

$$f_0 = \frac{f_{bulk}}{(c^e_{\gamma'} - c)^2(c - c^e_{\gamma})^2} \Rightarrow$$

$$f_0 = \frac{149.9}{(0.16 - 0.195)^2(0.195 - 0.23)^2} \Rightarrow$$

$$f_0 = \frac{149.9}{0.000001500625} \Rightarrow \boxed{f_0 = 9.989 \times 10^7 J/mol}$$

Hence approximately bulk energy density coefficient becomes $9.989 \times 10^7 J/mol$.

Answer to Task 3

Determination of gradient energy density coefficient, K_c

Gradient energy density is given by:

$f_{Grad} = \frac{K_c}{2}|\nabla_c|^2$

Depicted from $\frac{\partial C}{\partial t} = M\nabla^2 \frac{\partial F}{\partial C}$

Given that:

- $V_m = 1.0x10^{-5} m^3 mol^{-1}$,
- $M = 1.0x10^{-17} mol^2 J^{-1} m^{-1} s^{-1}$
- $\partial t = 1s$
- $\partial x = 1.0x10^{-8} m$

- $F_{infinite} = 5\ 50x10^{-3} Jm^{-2}$

From Task 2 solution and the gradient function above with incorporation of the differential function above, the gradient function becomes:

$$\frac{\partial C}{\partial t} = M\nabla^2\Big[\frac{\partial f_{bulk}}{\partial C} - \frac{\nabla \partial f_{gradient}}{\partial(\nabla C)}\Big] \tag{2}$$

However, as it is mentioned in the theory in phase-field the interface is diffusive and the thickness is much wider than a few atom layers, which seems to be intuitionally unphysical. So, It turns out that on one hand the diffusive interface is a numeric advantage. Because if it is a sharp interface (numerically a line), we have to always trace the line in order to know where the interface is.
On the other hand we can still make it a physical system as long as we construct the interface energy correctly. In the interface region $0 < c$ (or ϕ) < 1, we will have some bulk energy according to Fig.2. However, if comparing this volume energy with the experimental interface energy, we will find that there is still some of the missing energy.
This is due to the fact that the interface region is inhomogeneous, resulting in a increase in energy. Therefore, in order to compensate the energy of the interface, a gradient energy is introduced into the phase field.
So, we have to use the Eq.(23) of the task 2 and put it into the Eq.(2) above, so in the end we will take:

$$\frac{\partial c}{\partial t} = M\nabla^2[-2f_0(c - c^e_\gamma)(c - c^e_{\gamma'})(c^e_{\gamma'} - 2c + c^e_\gamma) - \nabla(K_c\nabla_c)] \tag{3}$$

The coefficients f_0, c^e_γ and $c^e_{\gamma'}$ are known from the previous task, M was given as $M = 1x10^{-17} mol^2 \cdot J^{-1} \cdot m^{-1} \cdot s^{-1}$.
Furthermore, a molar volume $V_m = 1x10^{-5} m^3 \cdot mol^{-1}$ was given which was needed to equalize the units in Eq.(3) to make the parameters consistent. .

So we have in the end:

$$\frac{\partial c}{\partial t} = MV_m^2\nabla^2\Big[\frac{-2f_0}{V_m}(c - c^e_\gamma)(c - c^e_{\gamma'})(c^e_{\gamma'} - 2c + c^e_\gamma) - \nabla(K_c\nabla^2 c)\Big] \tag{4}$$

Now as we can notice from the physical units - the parameters are consistent.
As it mentioned in the theory, for finding a proper value for K_c we have to use the trial and error method.
The correct K_c should not only let numeric interface energy roughly agree with experimental interface energy, but also keep a proper interface thickness and numerics stable.

Using the Fipy tutorials and set up a 1D simulation we can end up in a proper value of K_c= 5.e-8. Also, we ended up in the following values of F:

- $F_{bulk} = 0.026668608982321033$
- $F_{grad} = 0.02650330511695169$
- $F_{inte} = 0.05317191409927272$
- interface with is (in nm) 6.9999999999999964

Also, by extending the simulation to 2D and set c = 0.195 plus a gaussian noise, we ended up in the following result:

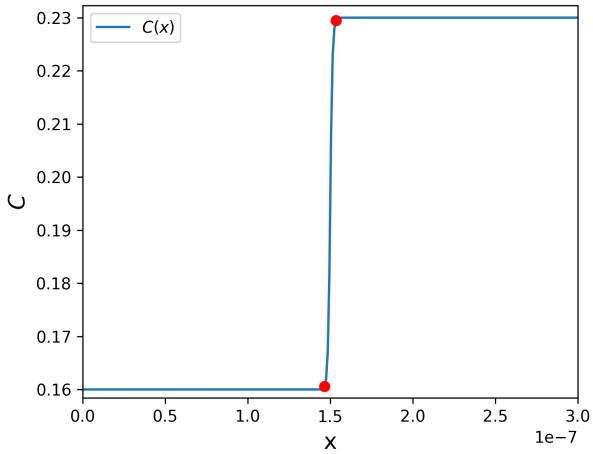

Figure 5: The plot of the 1D simulation

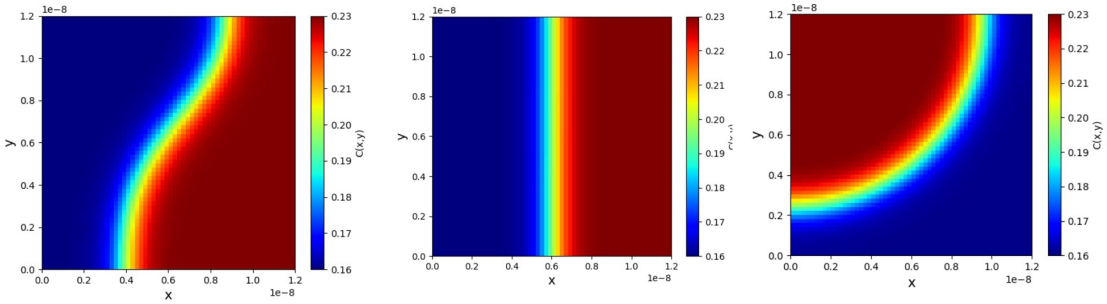

Figure 6: The plot of the 2D simulation

1 Appendix

Table A.1: Implementation of the present code in FiPy for Task 1

```
from fipy import *
from scipy.integrate import simpson
import matplotlib.pyplot as plt


def Energies(phi,dx):
    # Energy densities
    fbulk = f0*( phi*(1-phi) )**2
    fgrad = 0.5*K*numerix.gradient(phi,dx,edge_order=2)**2

    # Integrate to obtain Energies
    Fb=simpson(fbulk,dx=dx)
    Fg=simpson(fgrad,dx=dx)
    Fint = Fb + Fg

    return fbulk,fgrad,Fb,Fg,Fint


#mesh
dx = 1.
Lx = 200.
nx = int(Lx/dx)
mesh = Grid1D(dx=dx, nx=nx)
x = mesh.cellCenters[0]

#time discretization:
dt = 1
steps = 50

#field varialbe
phi = CellVariable(name=r'$\phi(x)$',mesh=mesh, hasOld=1)

#initial condition
# phi0 = GaussianNoiseVariable(mesh=mesh, mean=0.5, variance=0.01, hasOld=0)
# phi.setValue(value = phi0.value)
phi.setValue(1.)
phi.setValue(0., where=x > Lx/2)

#boundary condtions
phi.faceValue.constrain(value=1., where=mesh.facesLeft)
phi.faceValue.constrain(value=0., where=mesh.facesRight)
# phi.faceGrad.constrain(value=0., where=mesh.facesLeft)
# phi.faceGrad.constrain(value=0., where=mesh.facesRight)
```

```
#boundary condtions
phi.faceValue.constrain(value=1., where=mesh.facesLeft)
phi.faceValue.constrain(value=0., where=mesh.facesRight)
# phi.faceGrad.constrain(value=0., where=mesh.facesLeft)
# phi.faceGrad.constrain(value=0., where=mesh.facesRight)

#equation coefficients
f0 = 1. ; L = 1. ; K=4. ;

#equation
s1 = -( 2*(1-2*phi)**2 - 4*phi*(1-phi) )*f0
s0 = -( 2*phi*(1-phi)*(1-2*phi) )*f0 - s1*phi

eq = TransientTerm(coeff=1/L) == DiffusionTerm(coeff=K) \
        + ImplicitSourceTerm(coeff=s1)  + s0

#viewers
viewer = Viewer(vars=phi,xmin=0.,xmax=Lx,datamax=1.05,datamin=-0.05,\
                legend='upper right')
viewer.axes.set_xlabel("x",fontsize=14)
viewer.axes.set_ylabel("$\phi(x)$",fontsize=14)

# Energies
fbulk = CellVariable(name=r'$f_{bulk}$', mesh=mesh, hasOld=0)
fgrad = CellVariable(name=r'$f_{grad}$',mesh=mesh, hasOld=0)

Fint=numerix.zeros(steps+1) # array to store Fint(t)
# calculate initial energies:
fbulk.value,fgrad.value,Fb,Fg,Fint[0] = Energies(phi.value,dx)

#solver
for step in range(steps):
    print(step)
    phi.updateOld()
    res = 1.e5
    while res > 1.e-4:
        res = eq.sweep(var=phi, dt=dt)
        viewer.plot()

    fbulk.value,fgrad.value,Fb,Fg,Fint[step+1] = Energies(phi.value,dx)
```

```
#----------------------------------------
# Other plots

f = CellVariable(name=r'$f_{total}$',mesh=mesh, hasOld=0)
f.value = fbulk.value + fgrad.value

viewer2 = Viewer(vars=(fbulk,fgrad,f),xmin=0.,xmax=Lx,datamin=0,\
                 datamax=1.05*max(f))
viewer2.axes.set_xlabel("x",fontsize=14)
viewer2.axes.set_ylabel("$f(x)$",fontsize=14)
viewer2.plot()

#------------------------------------------------
plt.figure(3)
t = numerix.arange(0,(steps+1)*dt,dt)
plt.plot(t,Fint)
plt.xlabel("t",fontsize=14)
plt.ylabel("F",fontsize=14)
plt.xlim(0,steps*dt)
plt.show()
```

Table A.2: Implementation of the present code in FiPy for Task 3 (1D)

```
from fipy import *
from scipy.integrate import simpson

def Energies(C,dx):
    # Energy densities
    fbulk = a*( (C-C1)*(C2-C)  )**2
    fgrad = 0.5*Kc*numerix.gradient(C,dx,edge_order=2)**2

    # Integrate to obtain Energies
    Fb=simpson(fbulk,dx=dx)
    Fg=simpson(fgrad,dx=dx)
    Fint = Fb + Fg

    return Fb,Fg,Fint


# Model Constants
Vm = 1.e-5;  M = 1.e-17;  D=M*Vm**2
f0 = 9.989e7;  a=f0/Vm
Kc = 5.e-8
C1=0.16;  C2=0.23

#mesh
dx = 1e-9 #5.e-10
nx=300
Lx=nx*dx
mesh = Grid1D(dx=dx, nx=nx)
x = mesh.cellCenters[0]

#time discretization:
dt = 1
steps = 30

#field varialbe
C = CellVariable(name=r'$C(x)$',mesh=mesh, hasOld=1)
psi = CellVariable(mesh=mesh, hasOld=1)
```

```
#initial condition
# C0 = GaussianNoiseVariable(mesh=mesh, mean=0.195, variance=0.001, hasOld=0)
# C.setValue(value = C0.value)
C.setValue(C1)
C.setValue(C2, where=x > Lx/2)

#boundary conditions
C.faceValue.constrain(value=C1, where=mesh.facesLeft)
C.faceValue.constrain(value=C2, where=mesh.facesRight)
psi.faceGrad.constrain(value=0., where=mesh.facesLeft)
psi.faceGrad.constrain(value=0., where=mesh.facesRight)

#equations
dfdC = a* 2 * (C-C1) * (C2 - C) * (C1+C2 - 2*C)
d2fdC2 = a* ( 2*(C - C1)**2 - 8*(C - C1)*(C2 - C) + 2*(C2-C)**2 )

eq1 = (TransientTerm(var=C) == DiffusionTerm(coeff=D, var=psi))

eq2 = (ImplicitSourceTerm(coeff=1., var=psi) \
       == ImplicitSourceTerm(coeff=d2fdC2, var=C) - d2fdC2*C + dfdC \
           - DiffusionTerm(coeff=Kc, var=C))

eq = eq1 & eq2


#viewer
viewer = Viewer(vars=C,xmin=0,xmax=Lx,datamax=1.01*C2,datamin=0.99*C1,\
                legend='upper left')
viewer.axes.set_xlabel("x",fontsize=14)
viewer.axes.set_ylabel("$C$",fontsize=14)

#solver
for step in range(steps):
    print('step = ',step)
    C.updateOld()
    psi.updateOld()
    res = 1.e5
    while res > 1.e-4:
        res = eq.sweep(dt=dt)
        viewer.plot()

Fb,Fg,Fint = Energies(C.value,dx)
print('F_bulk =',Fb)
print('F_grad =',Fg)
print('F_inte =',Fint)


# Calculate interface thickness
tol = 0.0005
zone = numerix.logical_and(abs(C2-C.value)>tol, abs(C1-C.value)>tol)
Cmin = min(C.value[zone]); Cmax = max(C.value[zone])
Xmin = min(x.value[zone]); Xmax = max(x.value[zone])
print('interface with is (in nm)',(Xmax-Xmin)*1e9)

#viewer.axes.plot(x.value,C.value,'r*')
viewer.axes.plot([Xmin,Xmax],[Cmin,Cmax],'ro')
```

Table A.3: Implementation of the present code in FiPy for Task 3 (2D)

```
from fipy import *
from scipy.integrate import simpson

# Model Constants
Vm = 1.e-5;  M = 1.e-17;  D=M*Vm**2
f0 = 9.989e7;  a=f0/Vm
Kc = 2.e-8
C1=0.16;  C2=0.23

#mesh
nx=ny=60
dx = dy = 2.e-10
mesh = Grid2D(nx=nx, ny=ny, dx=dx, dy=dy)

#time discretization:
dt = 1
steps = 1000

#field varialbe
C = CellVariable(name='C(x,y)',mesh=mesh, hasOld=1)
psi = CellVariable(mesh=mesh, hasOld=1)

#initial condition
C0 = GaussianNoiseVariable(mesh=mesh, mean=0.195, variance=0.002, hasOld=0)
C.setValue(value = C0.value)

#equations
dfdC = a* 2 * (C-C1) * (C2 - C) * (C1+C2 - 2*C)
d2fdC2 = a* ( 2*(C - C1)**2 - 8*(C - C1)*(C2 - C) + 2*(C2-C)**2 )

eq1 = (TransientTerm(var=C) == DiffusionTerm(coeff=D, var=psi))

eq2 = (ImplicitSourceTerm(coeff=1., var=psi) \
       == ImplicitSourceTerm(coeff=d2fdC2, var=C) - d2fdC2*C + dfdC \
           - DiffusionTerm(coeff=Kc, var=C))

eq = eq1 & eq2


#viewer
viewer = Viewer(vars=(C,), datamin=C1, datamax=C2)
viewer.axes.set_xlabel("x",fontsize=14)
viewer.axes.set_ylabel("y",fontsize=14)

#solver
for step in range(steps):
    print('step = ',step)
    C.updateOld()
    psi.updateOld()
    res = 1.e5
    while res > 1.e-4:
        res = eq.sweep(dt=dt)
        viewer.plot()
```