```
import numpy as np
from numpy.random import *
import matplotlib.pyplot as plt
import math


nx = 128  # Number of computational grids along the x direction
ny = nx  # Number of computational grids along the y direction
number_of_grain = 10  # Total number of grains (N)
dx, dy = 0.5e-6, 0.5e-6  # Spacing of computational grids [m]
dt = 0.05  # Time increment [s]
nsteps = 500  # Total number of time steps
pi = np.pi
delta = 6.0 * dx  # Thickness of diffuse interface
eee = 3.0e+6  # Magnitude of driving force

# Grain boundary energy matrix
sigma_matrix = np.array([
    [0, 1, 1, 0.5, 1, 1, 1, 1, 0.67, 1],  # Energy between grains 1-2 and 1-3
    [1, 0, 1, 1, 1, 1, 1, 1, 1, 1],  # Energy between grains 2-1 and 2-3
    [1, 1, 0, 1, 1, 1, 1, 1, 1, 1],   # Energy between grains 3-1 and 3-2
    [0.5, 1, 1, 0, 1, 1, 1, 1, 1, 1],  # Energy between grains 1-2 and 1-3
    [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  # Energy between grains 2-1 and 2-3
    [1, 1, 1, 1, 1, 0, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 0, 1, 1, 1],  # Energy between grains 1-2 and 1-3
    [1, 1, 1, 1, 1, 1, 1, 0, 1, 1],  # Energy between grains 2-1 and 2-3
    [0.67, 1, 1, 1, 1, 1, 1, 1, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
])



# Gradient energy coefficient for each grain boundary pair
aaa_matrix = 2.0 / pi * np.sqrt(2.0 * delta * sigma_matrix)

# Height of double-obstacle potential for each grain boundary pair
www_matrix = 4.0 * sigma_matrix / delta

# Mobility of phase-field remains unchanged
pmobi = pi * pi / (8.0 * delta) * 3.0e-13

# Display results for verification
print("Gradient Energy Coefficient Matrix (aaa):")
print(aaa_matrix)
print("\nHeight of Double-Obstacle Potential Matrix (www):")
print(www_matrix)
print("\nPhase-Field Mobility (pmobi):", pmobi)
```

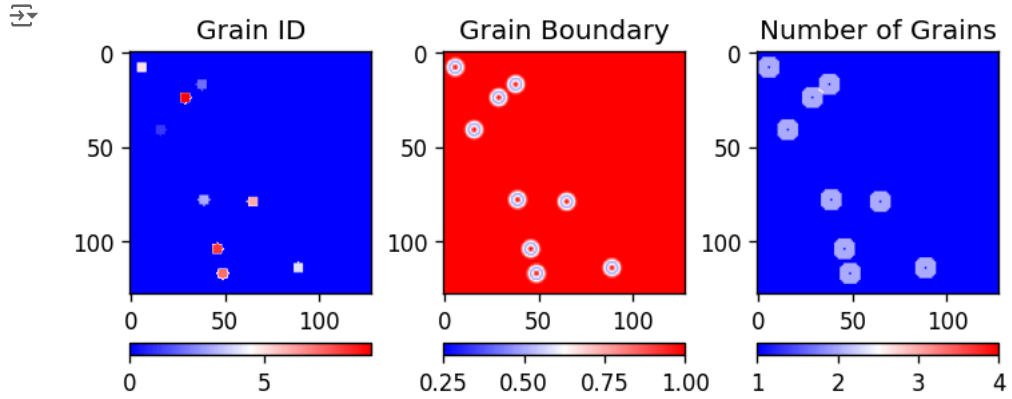
```
Gradient Energy Coefficient Matrix (aaa):
[[0.         0.00155939 0.00155939 0.00110266 0.00155939 0.00155939
  0.00155939 0.00155939 0.00127642 0.00155939]
 [0.00155939 0.         0.00155939 0.00155939 0.00155939 0.00155939
  0.00155939 0.00155939 0.00155939 0.00155939]
 [0.00155939 0.00155939 0.         0.00155939 0.00155939 0.00155939
  0.00155939 0.00155939 0.00155939 0.00155939]
 [0.00110266 0.00155939 0.00155939 0.         0.00155939 0.00155939
  0.00155939 0.00155939 0.00155939 0.00155939]
 [0.00155939 0.00155939 0.00155939 0.00155939 0.         0.00155939
  0.00155939 0.00155939 0.00155939 0.00155939]
 [0.00155939 0.00155939 0.00155939 0.00155939 0.00155939 0.
  0.00155939 0.00155939 0.00155939 0.00155939]
 [0.00155939 0.00155939 0.00155939 0.00155939 0.00155939 0.00155939
  0.         0.00155939 0.00155939 0.00155939]
 [0.00155939 0.00155939 0.00155939 0.00155939 0.00155939 0.00155939
  0.00155939 0.         0.00155939 0.00155939]
 [0.00127642 0.00155939 0.00155939 0.00155939 0.00155939 0.00155939
  0.00155939 0.00155939 0.         0.00155939]
 [0.00155939 0.00155939 0.00155939 0.00155939 0.00155939 0.00155939
  0.00155939 0.00155939 0.00155939 0.        ]]

Height of Double-Obstacle Potential Matrix (www):
[[      0.         1333333.33333333 1333333.33333333  666666.66666667
  1333333.33333333 1333333.33333333 1333333.33333333 1333333.33333333
   893333.33333333 1333333.33333333]
 [1333333.33333333       0.         1333333.33333333 1333333.33333333
  1333333.33333333 1333333.33333333 1333333.33333333 1333333.33333333
```

```
 1333333.33333333 1333333.33333333]
[1333333.33333333 1333333.33333333       0.         1333333.33333333
 1333333.33333333 1333333.33333333 1333333.33333333 1333333.33333333
 1333333.33333333 1333333.33333333]
[ 666666.66666667 1333333.33333333 1333333.33333333       0.
 1333333.33333333 1333333.33333333 1333333.33333333 1333333.33333333
 1333333.33333333 1333333.33333333]
[1333333.33333333 1333333.33333333 1333333.33333333 1333333.33333333
       0.         1333333.33333333 1333333.33333333 1333333.33333333
 1333333.33333333 1333333.33333333]
[1333333.33333333 1333333.33333333 1333333.33333333 1333333.33333333
 1333333.33333333       0.         1333333.33333333 1333333.33333333
 1333333.33333333 1333333.33333333]
[1333333.33333333 1333333.33333333 1333333.33333333 1333333.33333333
 1333333.33333333 1333333.33333333       0.         1333333.33333333
 1333333.33333333 1333333.33333333]
[1333333.33333333 1333333.33333333 1333333.33333333 1333333.33333333
 1333333.33333333 1333333.33333333 1333333.33333333       0.
 1333333.33333333 1333333.33333333]
[ 893333.33333333 1333333.33333333 1333333.33333333 1333333.33333333
 1333333.33333333 1333333.33333333 1333333.33333333 1333333.33333333
       0.         1333333.33333333]
[1333333.33333333 1333333.33333333 1333333.33333333 1333333.33333333
 1333333.33333333 1333333.33333333 1333333.33333333 1333333.33333333
 1333333.33333333       0.        ]]

Phase-Field Mobility (pmobi): 1.2337005501361697e-07
```

```
wij = np.zeros((number_of_grain,number_of_grain)) # array for the height of double-obstacle potential
aij = np.zeros((number_of_grain,number_of_grain)) # array for the gradient energy coefficient
mij = np.zeros((number_of_grain,number_of_grain)) # array for the mobility of phase-field
eij = np.zeros((number_of_grain,number_of_grain)) # arrays for saving the magnitude of driving foce of grain bou


phi = np.zeros((number_of_grain,nx,ny))
phi_new = np.zeros((number_of_grain,nx,ny))
mf = np.zeros((15,nx,ny),dtype = int)
nf = np.zeros((nx,ny),dtype = int)


gb = np.zeros((nx,ny))
gi = np.zeros((nx,ny),dtype = int)


for i in range(0, number_of_grain):
    for j in range(0, number_of_grain):
        if i != j:  # Only compute for distinct grain boundary pairs
            wij[i, j] = 4.0 * sigma_matrix[i, j] / delta  # Height of double-obstacle potential
            aij[i, j] = 2.0 / pi * np.sqrt(2.0 * delta * sigma_matrix[i, j])  # Gradient energy coefficient
            mij[i, j] = pi * pi / (8.0 * delta) * 3.0e-13  # Mobility of phase-field (constant)
            eij[i, j] = eee if (i == 0 or j == 0) else 0.0  # Driving force based on conditions
            if i < j:
                eij[i, j] = -eij[i, j]  # Apply sign change for i < j
        else:  # Diagonal elements for same grain
            wij[i, j] = 0.0
            aij[i, j] = 0.0
            mij[i, j] = 0.0
            eij[i, j] = 0.0



def update_nfmf(phi,mf,nf):
    for m in range(ny):
        for l in range(nx):
            l_p = l + 1
            l_m = l - 1
            m_p = m + 1
            m_m = m - 1
            if l_p > nx-1:
                l_p = l_p - nx
            if l_m < 0:
                l_m = l_m + nx
            if m_p > ny-1:
                m_p = m_p - ny
            if m_m < 0:
                m_m = m_m + ny
            n = 0
            for i in range(number_of_grain):
```

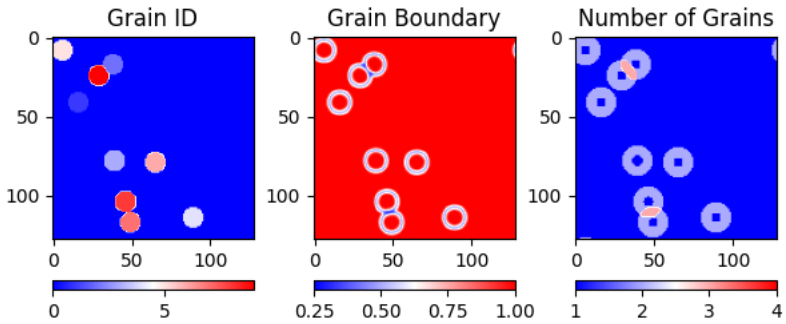
```
                if phi[i,l,m] > 0.0 or (phi[i,l,m] == 0.0 and phi[i,l_p,m] > 0.0 or phi[i,l_m,m] > 0.0 or phi[i,
                    n += 1
                    mf[n-1,l,m] = i
            nf[l,m] = n


def update_phasefield(phi, phi_new, mf, nf, eij, wij, aij, mij):
    for m in range(ny):
        for l in range(nx):
            l_p = l + 1
            l_m = l - 1
            m_p = m + 1
            m_m = m - 1
            if l_p > nx - 1:
                l_p -= nx
            if l_m < 0:
                l_m += nx
            if m_p > ny - 1:
                m_p -= ny
            if m_m < 0:
                m_m += ny
            for n1 in range(nf[l, m]):
                i = mf[n1, l, m]
                dpi = 0.0
                for n2 in range(nf[l, m]):
                    j = mf[n2, l, m]
                    ppp = 0.0
                    for n3 in range(nf[l, m]):
                        k = mf[n3, l, m]
                        ppp += (wij[i, k] - wij[j, k]) * phi[k, l, m] + \
                               0.5 * (aij[i, k] ** 2 - aij[j, k] ** 2) * \
                               (phi[k, l_p, m] + phi[k, l_m, m] + phi[k, l, m_p] + phi[k, l, m_m] - 4.0 * phi[k,
                    phii_phij = phi[i, l, m] * phi[j, l, m]
                    dpi -= 2.0 * mij[i, j] / float(nf[l, m]) * (ppp - 8. / pi * np.sqrt(phii_phij) * eij[i, j])
                phi_new[i, l, m] = phi[i, l, m] + dpi * dt

    # Apply constraints to ensure phase fields remain in valid range
    phi_new = np.clip(phi_new, 0.0, 1.0)

    # Normalize phase fields at each grid point
    for m in range(ny):
        for l in range(nx):
            a = np.sum(phi_new[:, l, m])
            phi[:, l, m] = phi_new[:, l, m] / a



phi = np.zeros((number_of_grain, nx, ny))
phi_new = np.zeros((number_of_grain, nx, ny))
mf = np.zeros((15, nx, ny), dtype=int)
nf = np.zeros((nx, ny), dtype=int)

phi[0, :, :] = 1.0  # Initial phase field for background
nf[:, :] = 1  # Number of phases at each grid point starts at 1
r_nuclei = 3.0 * dx  # Radius of the initial grains

# Initialize nuclei for grains
for i in range(1, number_of_grain):
    x_nuclei = int(rand() * nx)
    y_nuclei = int(rand() * ny)
    for m in range(ny):
        for l in range(nx):
            # Apply periodic boundary conditions
            if l > nx - 1:
                l -= nx
            if l < 0:
                l += nx
            if m > ny - 1:
                m -= ny
            if m < 0:
                m += ny
```

```
                # Calculate distance from nucleus center
                r = np.sqrt((l * dx - x_nuclei * dx) ** 2 + (m * dy - y_nuclei * dy) ** 2) - r_nuclei
                tmp = np.sqrt(2.0 * wij[0, 1]) / aij[0, 1] * r  # Adjusted to use `wij` and `aij`
                phi_tmp = 0.5 * (1.0 - np.sin(tmp))

                # Boundary conditions for phase field
                if tmp >= pi / 2.0:
                    phi_tmp = 0.0
                if tmp <= -pi / 2.0:
                    phi_tmp = 1.0

                # Update phase field and grain tracking arrays
                if 0.0 < phi_tmp < 1.0:
                    nf_tmp = nf[l, m] + 1
                    nf[l, m] = nf_tmp
                    mf[nf_tmp - 1, l, m] = i
                    phi[i, l, m] = phi_tmp
                    phi[0, l, m] -= phi[i, l, m]
                    if phi[0, l, m] < 0.0:
                        phi[0, l, m] = 0.0
                if phi_tmp >= 1.0:
                    nf_tmp = 1
                    nf[l, m] = nf_tmp
                    mf[0, l, m] = i
                    phi[i, l, m] = phi_tmp
                    phi[0, l, m] = 0.0

    # Normalize phase field and compute grain properties
    for m in range(ny):
        for l in range(nx):
            # Normalize phase field
            a = np.sum(phi[:, l, m])
            phi[:, l, m] = phi[:, l, m] / a

            # Compute grain boundary energy
            gb[l, m] = np.sum(phi[:, l, m] * phi[:, l, m])

            # Identify the dominant grain
            phi_max = 0.0
            for n in range(nf[l, m]):
                i = mf[n, l, m]
                if phi[i, l, m] > phi_max:
                    gi[l, m] = i
                    phi_max = phi[i, l, m]

    # Visualization
    fig = plt.figure(figsize=(7, 4))
    fig.set_dpi(120)
    plt.subplots_adjust(wspace=0.3)

    plt.subplot(1, 3, 1)
    plt.imshow(gi, cmap='bwr', vmin=0, vmax=number_of_grain - 1)
    plt.title('Grain ID')
    plt.colorbar(aspect=20, pad=0.1, orientation='horizontal')

    plt.subplot(1, 3, 2)
    plt.imshow(gb, cmap="bwr", vmin=0.25, vmax=1.0)
    plt.title('Grain Boundary')
    plt.colorbar(aspect=20, pad=0.1, orientation='horizontal')

    plt.subplot(1, 3, 3)
    plt.imshow(nf, cmap='bwr', vmin=1, vmax=4)
    plt.title('Number of Grains')
    plt.colorbar(aspect=20, pad=0.1, orientation='horizontal')

    plt.show()
```

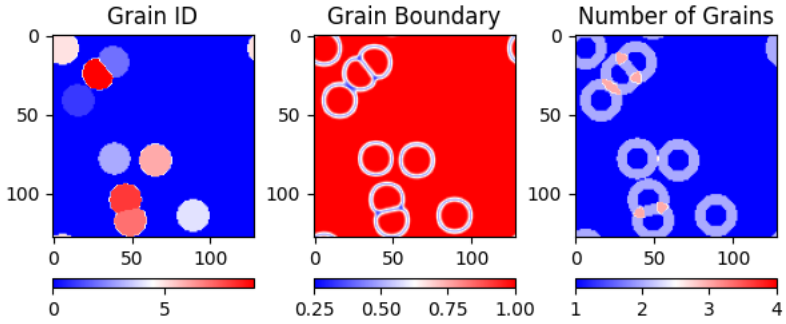



```
for nstep in range(1, nsteps + 1):
    # Update nf and mf arrays
    update_nfmf(phi, mf, nf)

    # Update phase fields with energy coupling
    update_phasefield(phi, phi_new, mf, nf, eij, wij, aij, mij)  # Added `wij`, `aij`, `mij` parameters

    # Output and visualization every 50 steps
    if nstep % 50 == 0:
        print('nstep =', nstep)

        # Recompute grain boundary and dominant grain ID
        for m in range(ny):
            for l in range(nx):
                # Compute grain boundary energy
                gb[l, m] = np.sum(phi[:, l, m] ** 2)

                # Find the dominant grain
                phi_max = 0.0
                for n in range(nf[l, m]):
                    i = mf[n, l, m]
                    if phi[i, l, m] > phi_max:
                        gi[l, m] = i
                        phi_max = phi[i, l, m]

        # Plot results
        fig = plt.figure(figsize=(7, 4))
        fig.set_dpi(100)
        plt.subplots_adjust(wspace=0.3)

        # Grain ID plot
        plt.subplot(1, 3, 1)
        plt.imshow(gi, cmap='bwr', vmin=0, vmax=number_of_grain - 1)
        plt.title('Grain ID')
        plt.colorbar(aspect=20, pad=0.1, orientation='horizontal')

        # Grain boundary plot
        plt.subplot(1, 3, 2)
        plt.imshow(gb, cmap='bwr', vmin=0.25, vmax=1.0)
        plt.title('Grain Boundary')
        plt.colorbar(aspect=20, pad=0.1, orientation='horizontal')

        # Number of grains per grid point plot
        plt.subplot(1, 3, 3)
        plt.imshow(nf, cmap='bwr', vmin=1, vmax=4)
        plt.title('Number of Grains')
        plt.colorbar(aspect=20, pad=0.1, orientation='horizontal')

        plt.show()
```

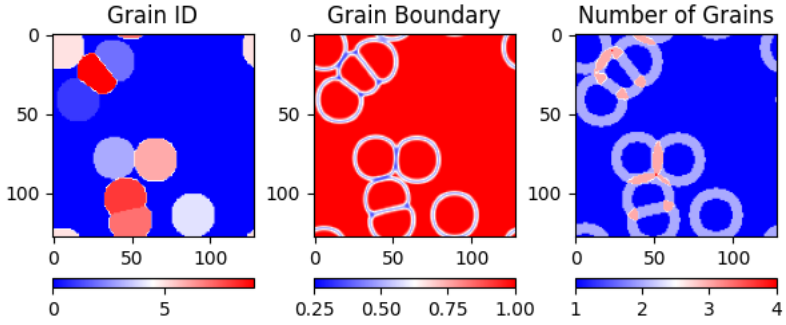
nstep = 50



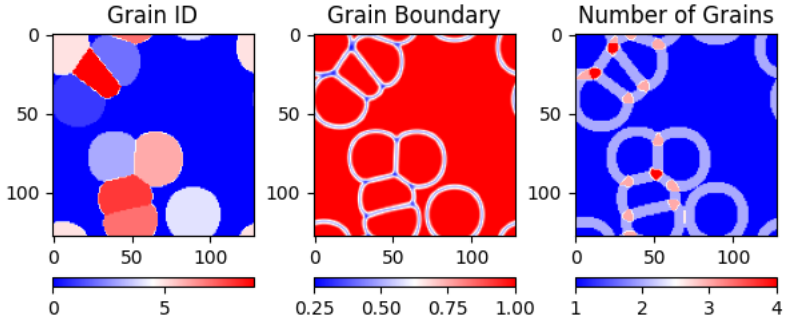
nstep = 100



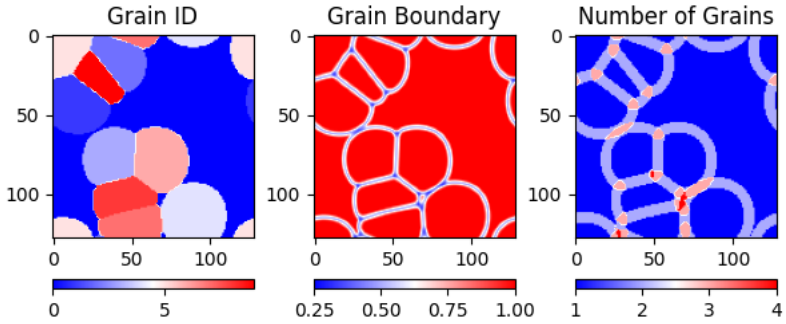
nstep = 150



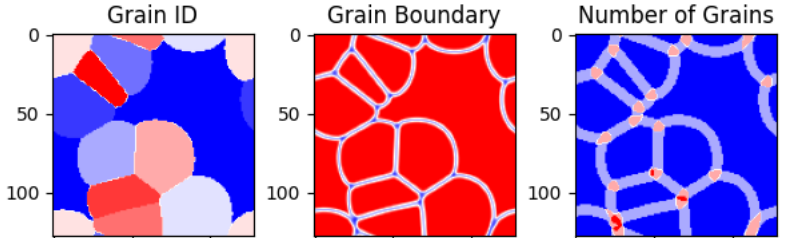
nstep = 200

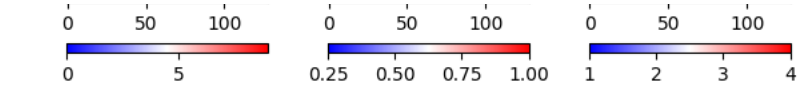


nstep = 250

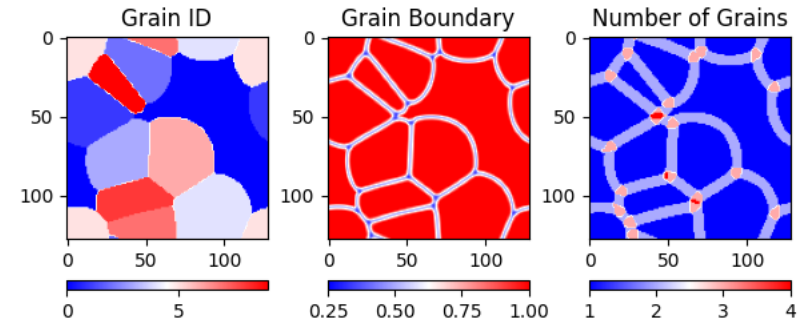


nstep = 300

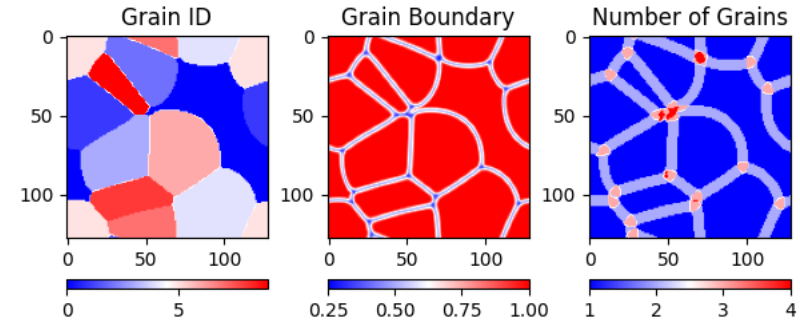




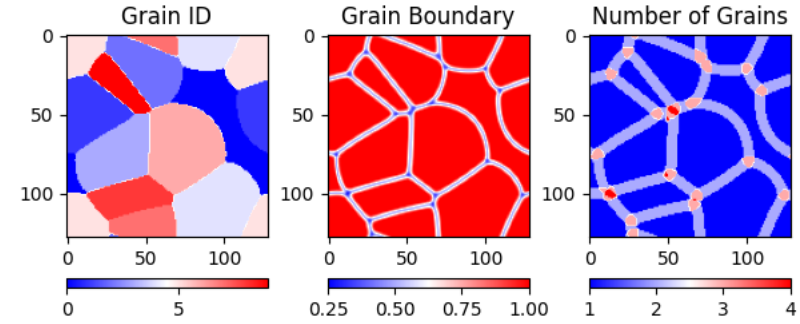
nstep = 350



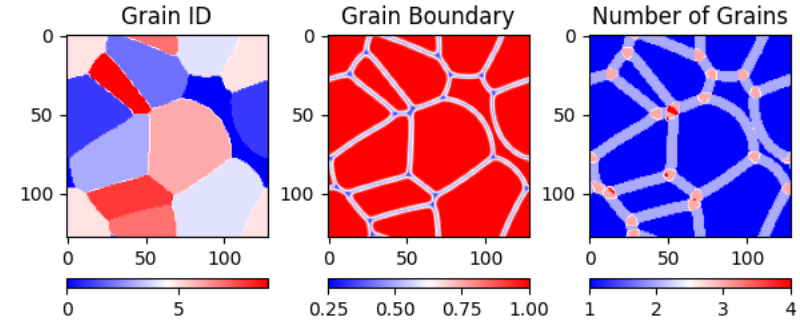
nstep = 400



nstep = 450



nstep = 500



```
import matplotlib.pyplot as plt
import numpy as np

# Initialize the time array and grain ID evolution list
time_steps = []
grain_id_evolution = []

# Time loop (existing code)
for nstep in range(1, nsteps + 1):
    update_nfmf(phi, mf, nf)
    update_phasefield(phi, phi_new, mf, nf, eij, wij, aij, mij)  # Updated to pass additional parameters

    # Track the evolution of grain IDs
    if nstep % 50 == 0:
        print('nstep = ', nstep)

        # Capture the current grain IDs based on `phi`
        current_grain_ids = np.zeros((nx, ny), dtype=int)
        for m in range(0, ny):
            for l in range(0, nx):
                # For each grid point, find the grain ID with the maximum phi value
                phi_max = 0.0
                grain_id = -1
                for n in range(nf[l, m]):
                    i = mf[n, l, m]
                    if phi[i, l, m] > phi_max:
                        grain_id = i
                        phi_max = phi[i, l, m]
                current_grain_ids[l, m] = grain_id

        # Store the grain IDs at this time step
        grain_id_evolution.append(current_grain_ids)
        time_steps.append(nstep * dt)  # Track time evolution (assuming `dt` is the time step)

        # Visualization of grain boundary, grain ID, etc. (unchanged)
        for m in range(0, ny):
            for l in range(0, nx):
                gb[l, m] = np.sum(phi[:, l, m] * phi[:, l, m])
                phi_max = 0.0
                for n in range(nf[l, m]):
                    i = mf[n, l, m]
                    if phi[i, l, m] > phi_max:
                        gi[l, m] = i
                        phi_max = phi[i, l, m]

        # Visualization every 50 steps (this part is unchanged)
        fig = plt.figure(figsize=(7, 4))
        fig.set_dpi(100)
        plt.subplots_adjust(wspace=0.3)

        plt.subplot(1, 3, 1)
        plt.imshow(gi, cmap='bwr', vmin=0, vmax=number_of_grain-1)
        plt.title('grain ID')
        plt.colorbar(aspect=20, pad=0.1, orientation='horizontal')
        plt.subplot(1, 3, 2)
        plt.imshow(gb, cmap='bwr', vmin=0.25, vmax=1.)
        plt.title('grain boundary')
        plt.colorbar(aspect=20, pad=0.1, orientation='horizontal')
        plt.subplot(1, 3, 3)
        plt.imshow(nf, cmap='bwr', vmin=1, vmax=4)
        plt.title('number of grains')
        plt.colorbar(aspect=20, pad=0.1, orientation='horizontal')
        plt.show()

# After the simulation loop, plot the evolution of grain IDs over time

# Create a plot for each grain ID's area over time
grain_area_evolution = []

for i in range(number_of_grain):
```