2D FEM SOLVER : ALLEN-CAHN EQUATION

The code provides a framework to solve the Allen-Cahn equation for different parameters and initial conditions and visualize the results.

In [1]:
```
import numpy as np
import sympy as sym
import scipy
from scipy.interpolate import PPoly, splrep
from scipy.special import roots_legendre
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

Double Well Potential and Derivative:

The code defines a symbolic function f representing a double well potential, which is a typical example of a double well potential in the Allen-Cahn equation. The potential function is given by f = et**2 * (et - 1)**2.

In [2]:
```
#Double Well potential and derivative
et = sym.var('et')
f = et**2 * (et - 1)**2
F = f.diff(et, 1)
F = sym.lambdify(et,F, 'numpy')
```

one_dim_M_K Function:

This function calculates the one-dimensional Mass matrix (M) and Stiffness matrix (K) using the given basis functions and their derivatives. The function returns the calculated M, K matrices, and the list of basis functions.

In [3]:
```
def one_dim_M_K(x, ndofs,deg):

    #Basis Function and derivative
    basisfun = [PPoly.from_spline(splrep(x, np.eye(ndofs)[i], k=deg)) for i in range(ndofs)]
    dbasisfun = [basisfun[i].derivative(1) for i in range(len(basisfun))]

    m = len(basisfun)
    bp = np.unique(basisfun[0].x) #Breakpoints
    bpd = np.diff(bp)

    #Quadrature points and weights
    q, w = roots_legendre(deg)
    q = (q + 1.0) / 2.0
    w = w / 2.0

    Q = np.array([bp[i] + bpd[i] * q for i in range(len(bpd))]).reshape((-1,))
    W = np.array([w * bpd[i] for i in range(len(bpd))]).reshape((-1,))

    Basisquad = np.array([basisfun[i](Q) for i in range(m)]).T
    dBasisquad = np.array([dbasisfun[i](Q) for i in range(m)]).T

    #One Dimensional Mass matrix and Stiffness Matrix
    M = np.array([[np.dot(Basisquad[:, i] * W, Basisquad[:, j]) for j in range(m)] for i in range(m)])
    K = np.array([[np.dot(dBasisquad[:, i] * W, dBasisquad[:, j]) for j in range(m)] for i in range(m)])

    return M,K,basisfun
```

two_dim_tensor_M_K Function:

This function takes the one-dimensional Mass matrix M and Stiffness matrix K and constructs the two-dimensional Mass matrix (M_2d) and Stiffness matrix (K_2d) using a tensor product structure. It reshapes the matrices to match the number of degrees of freedom (ndofs). The function returns the two-dimensional M_2d and K_2d matrices.

In [4]:
```
def two_dim_tensor_M_K(M,K,ndofs):

    assert M.shape[0] == K.shape[0], f"Shape Mismatch between M anb K"

    #Two Dimensional Mass matrix and Stiffness Matrix based on tensor product structure
    M_2d = np.einsum('ik,jl->ijkl', M, M).reshape(ndofs**2, ndofs**2)
    K_2d = np.einsum('ik,jl->ijkl', K, M).reshape((ndofs**2, ndofs**2)) + np.einsum('ik,jl->ijkl', M, K).reshape((ndofs**2, ndofs**2))

    return M_2d, K_2d
#
```

solver Function:

This function solves the PDE numerically using the FEM approach. It takes the initial condition func0 (a function of x and y representing the initial state), eps (a parameter representing diffusion coefficient), dt (time step), ndofs (number of degrees of freedom), and deg (degree of basis functions). It calculates the one-dimensional M and K matrices using the one_dim_M_K function and constructs the two-dimensional M_2d and K_2d matrices using the two_dim_tensor_M*K function. It then performs a time-stepping loop to compute the solution \eta*{k+1} at each time step. Finally, it returns the computed solution eta_2d and the list of basis functions.

In [5]:
```
def solver(func0, eps, dt, ndofs, deg):
    x = np.linspace(0, 1, ndofs)
    y = np.linspace(0, 1, ndofs)
    tstps = int(1 / dt) + 1

    M, K,basisfun = one_dim_M_K(x, ndofs, deg)
    M_2d, K_2d = two_dim_tensor_M_K(M, K, ndofs)

    eta_2d = np.zeros((tstps, ndofs**2))
    for i in range(tstps):
        if i == 0:
            eta_2d[i] = func0(x[:, None], y[None, :]).reshape(-1)
        else:
            A_2d = M_2d + dt * eps ** 2 * K_2d
            b_2d = M_2d.dot(eta_2d[i - 1] - dt * F(eta_2d[i - 1].reshape((ndofs, ndofs))).flatten())
            eta_2d[i] = scipy.linalg.solve(A_2d, b_2d)

    return eta_2d,basisfun
```

plot_2d Function:

This function plots the computed solution in both 3D surface and contour plots. It takes the solution eta_2d, dt, and ndofs as input. It reshapes the solution to match the dimensions of the grid, creates a meshgrid (X, Y), and plots the 3D surface using ax.plot_surface. It also plots the contour plot using ax.contourf.

In [6]:
```
def plot_2d(eta_2d, dt, ndofs):
    x = np.linspace(0, 1, ndofs)
    y = np.linspace(0, 1, ndofs)
    tstps = int(1 / dt) + 1

    X, Y = np.meshgrid(x, y)

    # 3D surface plot
    eta_2d_grid = eta_2d.reshape((tstps, ndofs, ndofs))
    fig = plt.figure(figsize=(13, 5))
    ax = fig.add_subplot(121, projection='3d')
    ax.plot_surface(X, Y, eta_2d_grid[-1], cmap='viridis', linewidth=0.5, antialiased=True)
    ax.set(xlabel='X', ylabel='Y', zlabel='eta', title='3D Surface Plot')
```

```python
    # Contour plot
    eta_2d_contour = eta_2d[-1].reshape((ndofs, ndofs))
    ax = fig.add_subplot(122)
    contour = ax.contourf(X, Y, eta_2d_contour, levels=100, cmap='viridis')
    fig.colorbar(contour, ax=ax, label='eta')
    ax.set(xlabel='x', ylabel='y', title='Contour Plot')


    plt.tight_layout()
    plt.show()
```

Main Execution

In [7]:
```python
if __name__ == '__main__':

    eps = 0.01
    dt = 0.1
    ndofs = 50
    deg = 2

    #Initial eta
    funcs = [
             lambda x, y: 1 / (1 + 100 * ((x - 0.5) ** 2 + (y - 0.5) ** 2)),  #Runge Function 2D
             lambda x, y: np.sin(2 * np.pi * x) * np.cos(2 * np.pi * y)       #sincos Function 2D
            ]

    for func0 in funcs:
        eta_2d,basisfun = solver(func0, eps, dt, ndofs, deg)
        plot_2d(eta_2d, dt, ndofs)
```