Problem 1: Support Vector Machines

Instructions:

1. Please use this q1.ipynb file to complete hw5-q1 about SVMs
2. You may create new cells for discussions or visualizations

In [1]:
```
# Install cvxopt with pip
#!pip install cvxopt
```

In [2]:
```
# Import modules
import numpy as np
import matplotlib.pyplot as plt
from cvxopt import matrix, solvers
```

a): Linearly Separable Dataset

In [3]:
```
data = np.loadtxt('clean_lin.txt', delimiter='\t')
x = data[:, 0:2]
y = data[:, -1]
```

In [4]:
```
(x.T * y).T
```

```
Out[4]: array([[ 0.0478,  0.9555],
               [ 1.4237, -0.396 ],
               [ 0.2514,  0.8968],
               [ 0.2549,  0.7987],
               [ 0.3378,  0.7251],
               [ 0.5349,  0.5453],
               [ 0.7319,  0.5371],
               [ 0.7768,  0.7088],
               [ 0.6593,  0.8028],
               [ 0.9807,  0.766 ],
               [ 0.877 ,  0.439 ],
               [ 0.8321,  0.1733],
               [ 0.6524,  0.3082],
               [ 1.4231,  0.9213],
               [ 1.2814,  0.6515],
               [ 1.3021,  0.3777],
               [ 1.1915,  0.1896],
               [ 1.0913, -0.1374],
               [ 1.4438,  0.112 ],
               [-0.0959, -0.3368],
               [-0.0752, -0.1569],
               [-0.1789, -0.2101],
               [-0.2549, -0.3368],
               [-0.324 , -0.2509],
               [-0.4934, -0.1651],
               [-0.5971, -0.0343],
               [-0.6005,  0.1169],
               [-0.718 ,  0.2355],
               [-0.5452,  0.3009],
               [-0.2272,  0.3091],
               [-0.7802,  0.3418],
               [-0.9565,  0.3377],
               [-0.1028,  0.3091],
               [-0.0579,  0.0188],
               [-0.1927, -0.0547],
               [-0.3862,  0.3091]])
```

```
In [5]: Q = matrix(np.diag([1.,1.,0.]))
        p = matrix(np.zeros(3))
        G = matrix((-1 * np.hstack(((x.T * y).T, y[:,None]))))
        h = matrix((-1 * np.ones(x.shape[0])))


        solution = solvers.qp(Q,p,G,h)
```

```
     pcost       dcost       gap    pres   dres
 0:  1.5173e+00  3.7821e+01  1e+02  2e+00  3e+01
 1:  1.3803e+01  9.6371e+00  3e+01  5e-01  7e+00
 2:  2.1717e+01  1.7415e+01  3e+01  4e-01  6e+00
 3:  3.9769e+01  3.8469e+01  8e+00  7e-02  1e+00
 4:  4.3534e+01  4.3375e+01  5e-01  4e-03  6e-02
 5:  4.3726e+01  4.3699e+01  3e-02  8e-06  1e-04
 6:  4.3723e+01  4.3721e+01  1e-03  4e-07  5e-06
 7:  4.3723e+01  4.3723e+01  1e-05  4e-09  5e-08
Optimal solution found.
```

```
In [6]: solution
```

```
Out[6]: {'x': <3x1 matrix, tc='d'>,
         'y': <0x1 matrix, tc='d'>,
         's': <36x1 matrix, tc='d'>,
         'z': <36x1 matrix, tc='d'>,
         'status': 'optimal',
         'gap': 1.2351050732129998e-05,
         'relative gap': 2.824869509847119e-07,
         'primal objective': 43.722565192948956,
         'dual objective': 43.72255316245894,
         'primal infeasibility': 3.6658505120358153e-09,
         'dual infeasibility': 4.8823395494201985e-08,
         'primal slack': 8.018556690959323e-09,
         'dual slack': 3.172431909201818e-08,
         'iterations': 7}
```

```
In [7]: z = np.ravel(solution['x'])

        w = z[:2]
        b = z[-1]
```

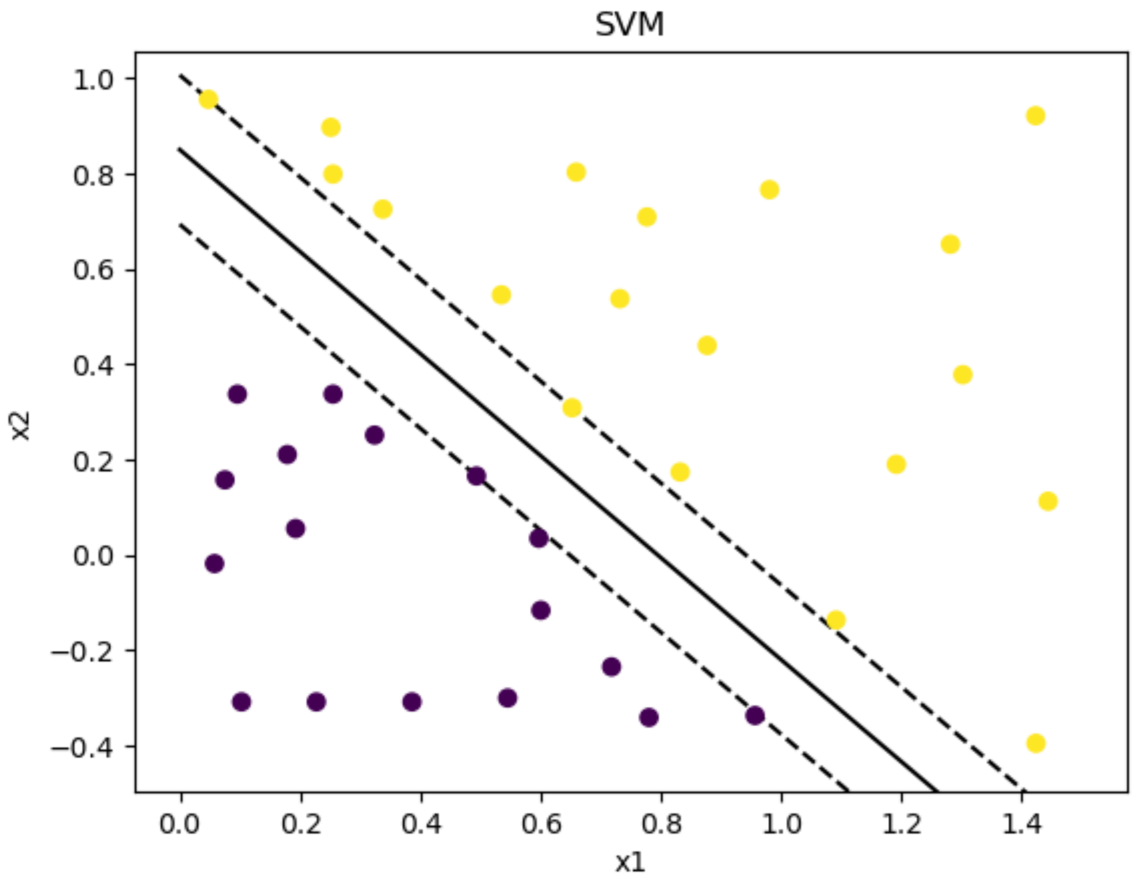
```
In [8]: print("Weights: ", w.T)
        print("Bias: ", b)
```

```
Weights:  [6.83386258 6.38305982]
Bias:  -5.425670975889619
```

```
In [9]: x[:,0].max()
```

```
Out[9]: 1.4438
```

```
In [10]: # plot the data
         margin = 1/np.linalg.norm(w[1])

         plt.scatter(x[:, 0], x[:, 1], c=y, zorder=10)

         # plot the separating hyperplaneusing w and b
         x_hyperplane = np.linspace(0,1.5)
         y_hyperplane = - (w[0] * x_hyperplane + b) / w[1]
         upper_margin = y_hyperplane + margin
         lower_margin = y_hyperplane - margin
         plt.plot(x_hyperplane, y_hyperplane, 'k-')
         plt.plot(x_hyperplane, upper_margin, 'k--')
         plt.plot(x_hyperplane, lower_margin, 'k--')
         plt.ylim([x[:,1].min()-0.1, x[:,1].max()+0.1])
         plt.title('SVM')
         plt.xlabel('x1')
         plt.ylabel('x2')
         plt.show()
```



b) and c) : Linearly Non-separable Dataset

In [11]:
```
# Load the data set that is not linearly separable
data = np.loadtxt('dirty_nonlin.txt', delimiter='\t')
x = data[:, 0:2]
y = data[:, 2]
```

In [12]:
```
c = 0.05
def soft_svm(x, y, c=0.05):
    Q = matrix(np.diag(np.hstack((np.ones(x.shape[1]),np.zeros(x.shape[0]+1)
    p = matrix(np.hstack((np.zeros(x.shape[1]+1),np.full(x.shape[0], c))))

    m1 = (-1 * np.hstack(((x.T * y).T, y[:,None])))
    m2 = np.diag(-1 * np.ones(x.shape[0]))
    m3 = np.zeros(m1.shape)
    m4 = np.diag(-1 * np.ones(x.shape[0]))
    m12 = np.hstack((m1, m2))
    m34 = np.hstack((m3, m4))
    G = matrix(np.vstack((m12, m34)))
    h = matrix(np.hstack(((-1 * np.ones(x.shape[0])), np.zeros(x.shape[0])))

    solution = solvers.qp(Q,p,G,h)
    z = np.ravel(solution['x'])

    w = z[:2]
    b = z[2]
    zeta = z[3:]
    return w, b, zeta
```

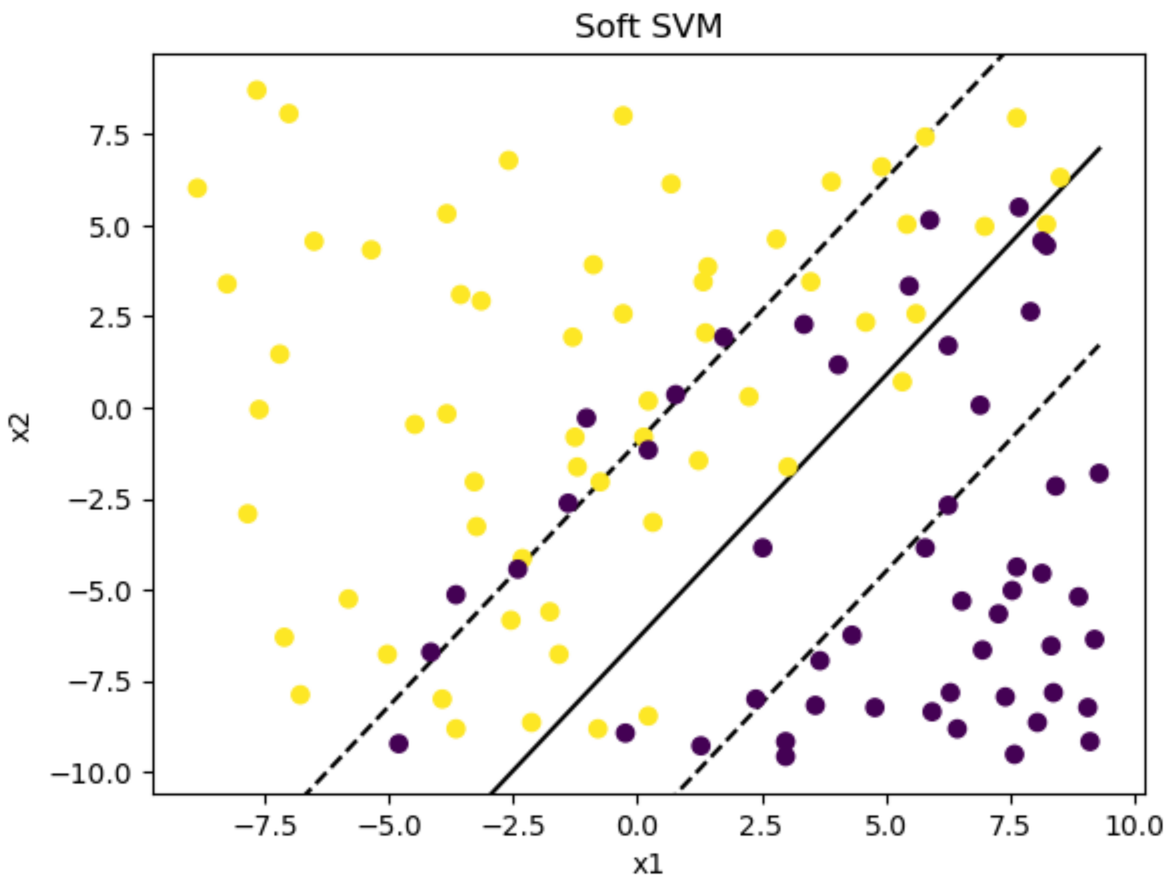
In [13]:
```
w, b, zeta = soft_svm(x, y)
```

```
     pcost       dcost       gap    pres   dres
 0:  1.2418e+00  3.9717e+01  6e+02  2e+00  7e+02
 1:  1.0376e+01 -1.5710e+01  3e+01  7e-02  2e+01
 2:  6.1726e+00  1.9674e-01  6e+00  1e-02  3e+00
 3:  2.6757e+00  1.9485e+00  7e-01  1e-03  3e-01
 4:  2.4029e+00  2.1622e+00  2e-01  3e-04  8e-02
 5:  2.3265e+00  2.2274e+00  1e-01  9e-05  3e-02
 6:  2.2888e+00  2.2613e+00  3e-02  2e-05  6e-03
 7:  2.2755e+00  2.2726e+00  3e-03  3e-16  4e-14
 8:  2.2740e+00  2.2739e+00  7e-05  4e-16  1e-14
 9:  2.2739e+00  2.2739e+00  7e-07  4e-16  5e-14
Optimal solution found.
```

In [14]:
```
print("Weights: ", w.T)
print("Bias: ", b)
print("Soft constraints: \n", zeta)
```

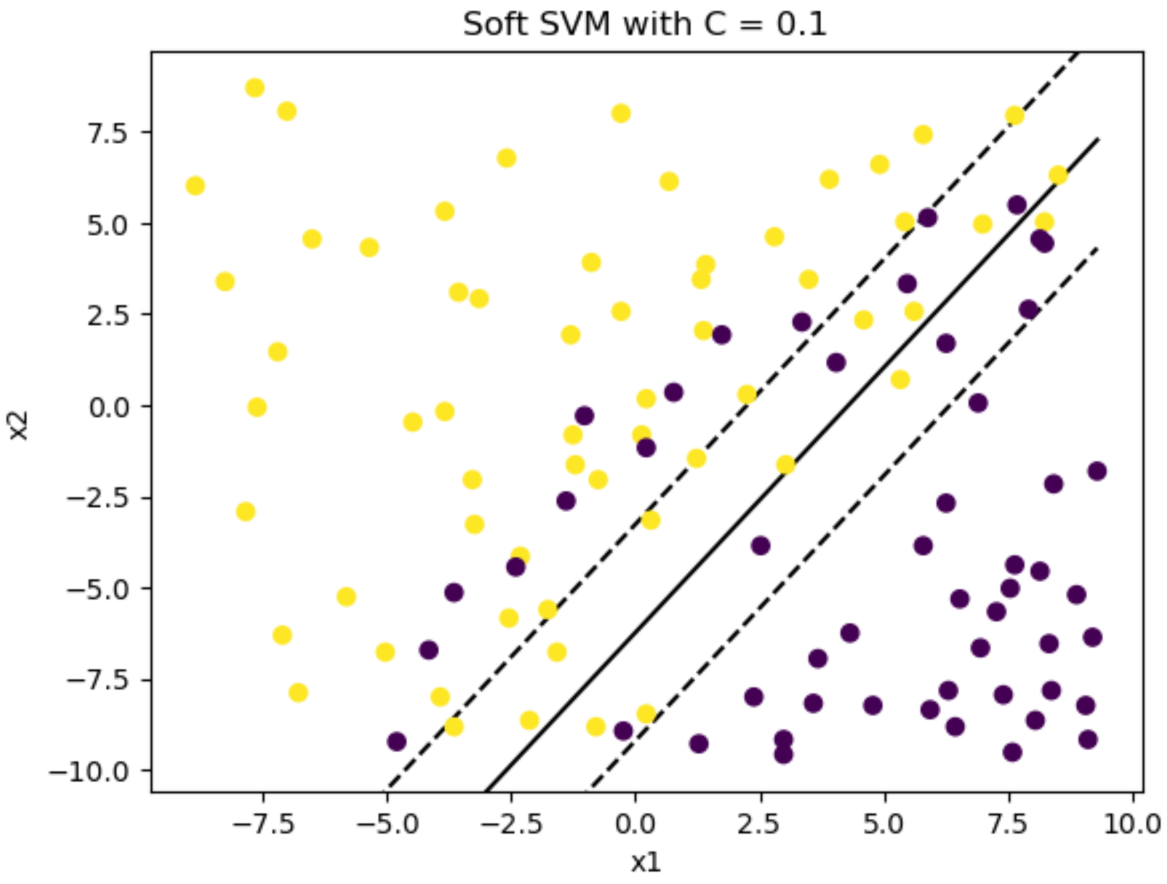
```
Weights:  [-0.26849615  0.18572682]
Bias:  1.1785387677637391
Soft constraints:
 [9.36079074e-01 1.10044314e+00 7.77226668e-01 8.50697893e-01
 6.21863705e-01 1.18843052e-01 6.54551518e-08 6.51798572e-08
 6.50563233e-08 6.51073782e-08 6.52230218e-08 6.55113189e-08
 6.64327901e-08 6.62833056e-08 6.50873472e-08 6.50041788e-08
 6.50322889e-08 6.51342946e-08 6.52353835e-08 6.55316801e-08
 2.22376447e-01 6.53636505e-08 6.59531045e-08 4.90642839e-01
 3.70405675e-01 1.12350785e+00 6.60204374e-08 6.50426553e-08
 6.50121381e-08 6.49769741e-08 6.49733014e-08 6.49746997e-08
 6.49829218e-08 6.49980744e-08 6.49979642e-08 6.49883097e-08
 6.50079854e-08 6.51249903e-08 6.54023014e-08 6.59163062e-08
 6.82862538e-08 9.73434511e-07 3.97320489e-01 3.45700449e-01
 7.31142714e-03 4.23579094e-01 9.38667996e-01 6.50913610e-08
 6.35764622e-08 3.89330546e-01 6.55751592e-01 8.54322358e-01
 4.78532748e-01 1.24567996e+00 1.45282125e+00 2.52461886e-01
 6.51728693e-08 6.50410302e-08 6.51336301e-08 6.52381968e-08
 1.75484983e+00 2.04800301e+00 2.20471922e+00 2.00076824e+00
 2.06478810e+00 1.90285467e+00 2.03636533e+00 2.06935851e+00
 1.70137974e+00 1.32741221e+00 7.91111104e-01 8.13390396e-01
 1.30970444e+00 6.50820303e-08 6.50516017e-08 6.50894761e-08
 6.50503271e-08 6.50327750e-08 6.50344423e-08 6.50168594e-08
 6.50620933e-08 6.50983173e-08 5.24146330e-02 6.15956309e-08
 1.20074554e-07 3.36520175e-01 5.41506698e-01 8.37513919e-01
 1.13480694e+00 1.55226955e+00 2.39959571e+00 7.85331866e-01
 6.35503893e-08 6.50828381e-08 6.50921362e-08 6.50820273e-08
 6.50670689e-08 6.50563905e-08 6.50275292e-08 6.50228805e-08
 1.10768141e-01 5.84148908e-01 6.49704587e-08 6.51150138e-08
 6.50837461e-08 6.50292042e-08 6.50081931e-08 6.50806268e-08
 6.16032416e-08 6.50976005e-08 6.50883957e-08]
```

In [15]:
```python
# plot the data
margin = 1/np.linalg.norm(w[1])

plt.scatter(x[:, 0], x[:, 1], c=y, zorder=10)

# plot the separating hyperplaneusing w and b
x_hyperplane = np.linspace(-8,x[:,0].max())
y_hyperplane = - (w[0] * x_hyperplane + b) / w[1]
upper_margin = y_hyperplane + margin
lower_margin = y_hyperplane - margin
plt.plot(x_hyperplane, y_hyperplane, 'k-')
plt.plot(x_hyperplane, upper_margin, 'k--')
plt.plot(x_hyperplane, lower_margin, 'k--')
plt.ylim([x[:,1].min()-1, x[:,1].max()+1])
plt.title('Soft SVM')
plt.xlabel('x1')
plt.ylabel('x2')
plt.show()
```

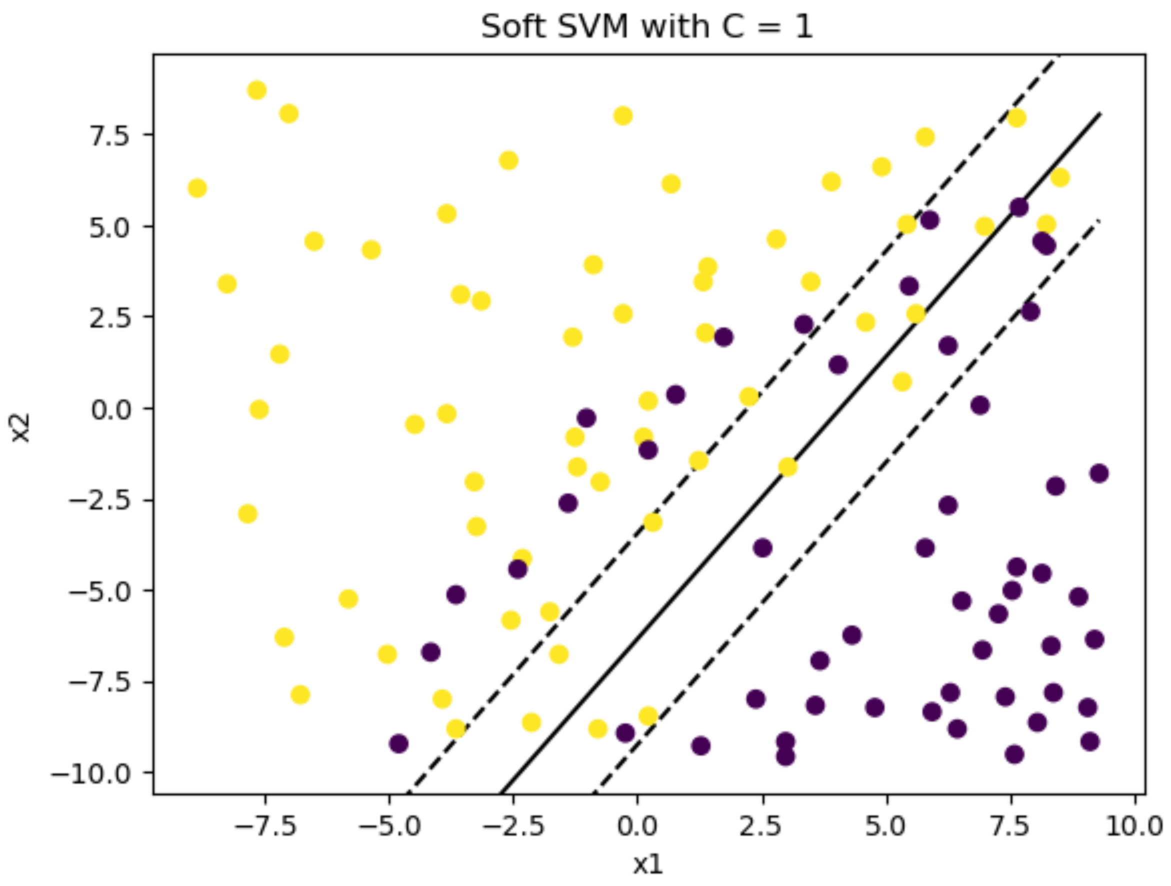


In [16]:
```
C = [0.1, 1, 100, 1000000]

for c in C:
    w, b, zeta = soft_svm(x, y, c)
    # plot the data
    margin = 1/np.linalg.norm(w)

    plt.scatter(x[:, 0], x[:, 1], c=y, zorder=10)

    # plot the separating hyperplaneusing w and b
    x_hyperplane = np.linspace(-8,x[:,0].max())
    y_hyperplane = - (w[0] * x_hyperplane + b) / w[1]
    upper_margin = y_hyperplane + margin
    lower_margin = y_hyperplane - margin
    plt.plot(x_hyperplane, y_hyperplane, 'k-')
    plt.plot(x_hyperplane, upper_margin, 'k--')
    plt.plot(x_hyperplane, lower_margin, 'k--')
    plt.ylim([x[:,1].min()-1, x[:,1].max()+1])
    plt.title(f'Soft SVM with C = {c}')
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.show()
```

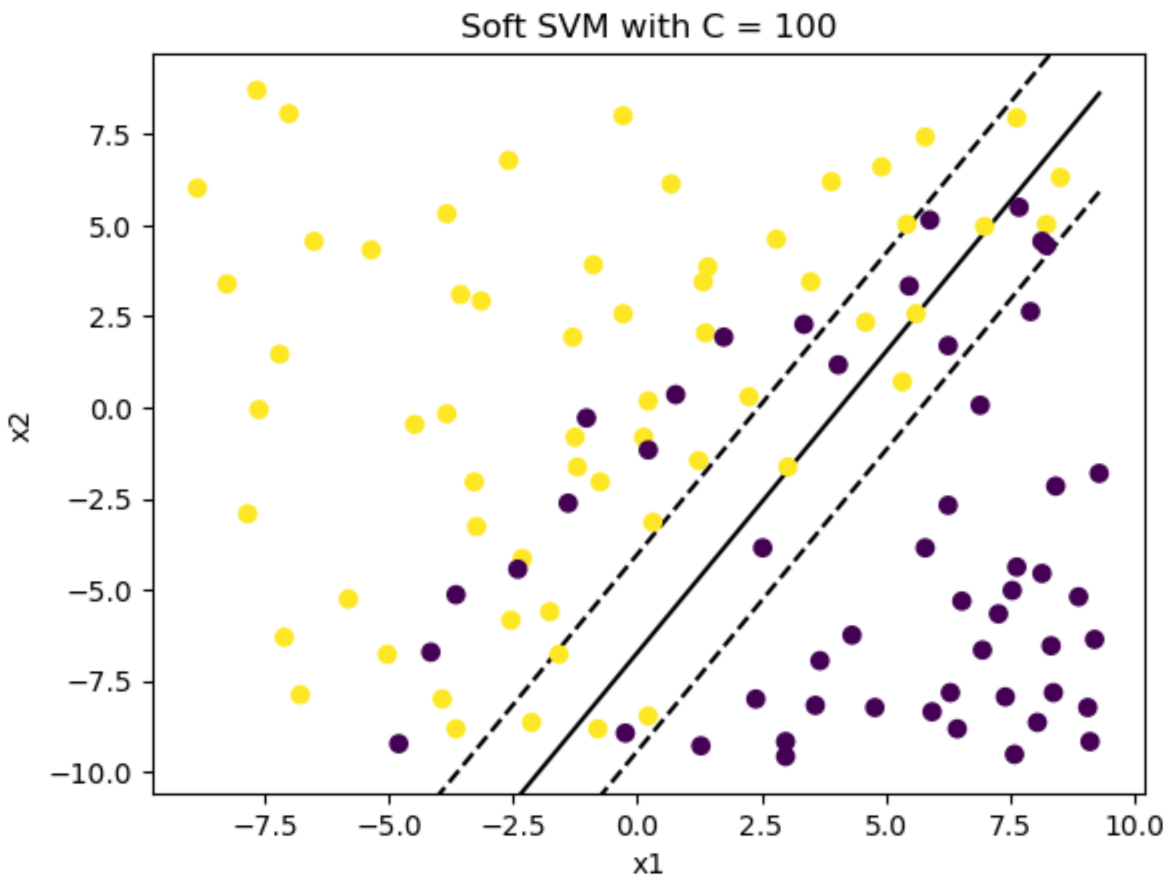
```
     pcost       dcost       gap    pres   dres
 0:  2.0644e+00  5.0524e+01  7e+02  3e+00  6e+02
 1:  1.9690e+01 -2.3059e+01  5e+01  1e-01  3e+01
 2:  1.1116e+01  1.1465e+00  1e+01  2e-02  4e+00
 3:  5.1661e+00  3.9158e+00  1e+00  2e-03  5e-01
 4:  4.6987e+00  4.3010e+00  4e-01  5e-04  1e-01
 5:  4.5752e+00  4.4251e+00  2e-01  1e-04  3e-02
 6:  4.5227e+00  4.4685e+00  5e-02  3e-05  7e-03
 7:  4.5033e+00  4.4844e+00  2e-02  9e-06  2e-03
 8:  4.4945e+00  4.4920e+00  3e-03  7e-07  2e-04
 9:  4.4932e+00  4.4931e+00  3e-05  8e-09  2e-06
10:  4.4931e+00  4.4931e+00  3e-07  8e-11  2e-08
Optimal solution found.
```



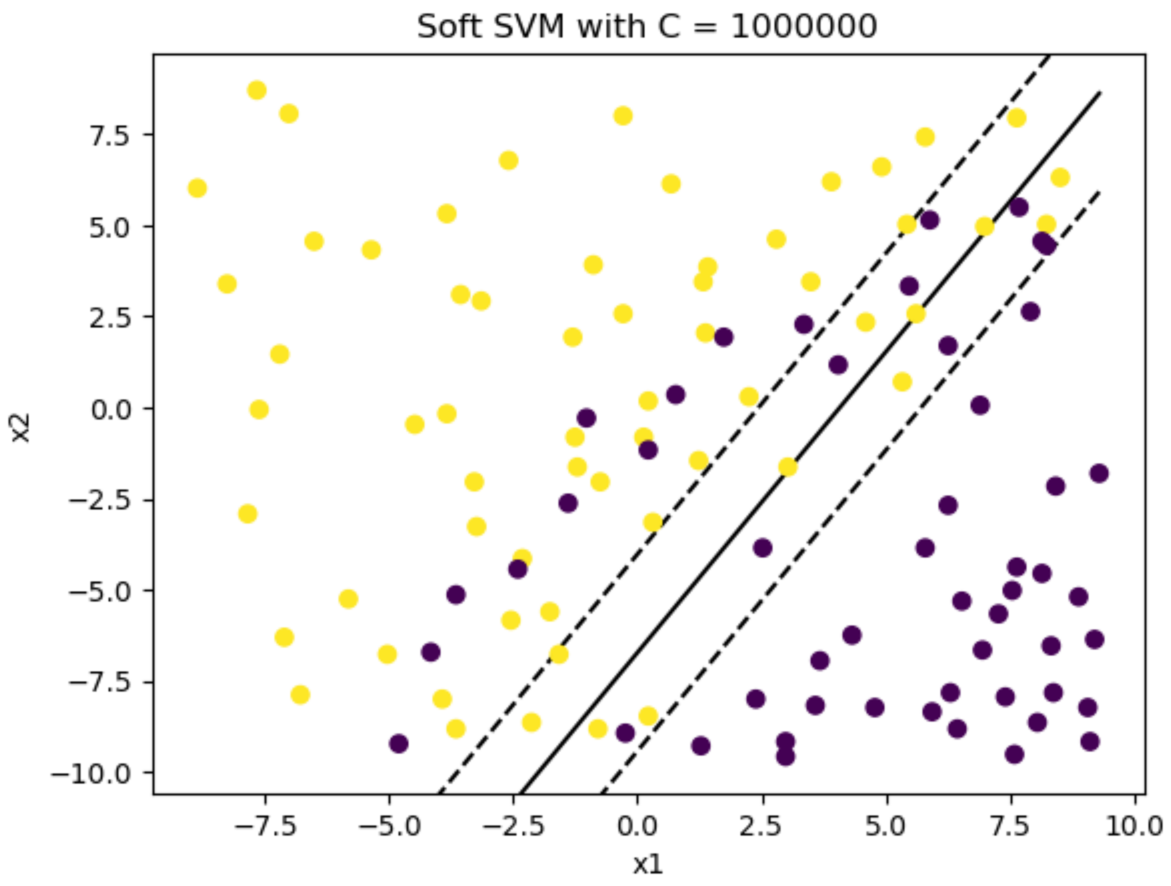
```
     pcost       dcost       gap    pres   dres
 0: -5.3430e+01  2.5866e+02  1e+03  4e+00  7e+01
 1:  1.3083e+02 -3.0700e+01  2e+02  5e-01  8e+00
 2:  5.6266e+01  3.4536e+01  2e+01  3e-02  5e-01
 3:  4.9781e+01  4.0113e+01  1e+01  1e-02  2e-01
 4:  4.6228e+01  4.2703e+01  4e+00  4e-03  6e-02
 5:  4.5200e+01  4.3715e+01  2e+00  1e-03  2e-02
 6:  4.4871e+01  4.3992e+01  9e-01  4e-04  6e-03
 7:  4.4528e+01  4.4306e+01  2e-01  8e-05  1e-03
 8:  4.4480e+01  4.4339e+01  1e-01  4e-05  7e-04
 9:  4.4429e+01  4.4388e+01  4e-02  4e-16  2e-14
10:  4.4408e+01  4.4407e+01  8e-04  3e-16  4e-13
11:  4.4407e+01  4.4407e+01  8e-06  3e-16  2e-13
Optimal solution found.
```



```
     pcost       dcost       gap    pres   dres
 0: -8.1935e+05  4.6922e+05  2e+06  2e+02  2e+01
 1:  1.4977e+05 -5.9743e+03  2e+05  4e+00  3e-01
 2:  8.0014e+03  2.6761e+03  6e+03  1e-01  8e-03
 3:  5.3250e+03  3.5779e+03  2e+03  3e-02  2e-03
 4:  5.1725e+03  3.8285e+03  1e+03  2e-02  1e-03
 5:  4.7825e+03  4.0969e+03  7e+02  8e-03  6e-04
 6:  4.6499e+03  4.2525e+03  4e+02  4e-03  3e-04
 7:  4.4915e+03  4.3960e+03  1e+02  1e-04  1e-05
 8:  4.4683e+03  4.4074e+03  6e+01  5e-05  4e-06
 9:  4.4356e+03  4.4342e+03  1e+00  8e-07  6e-08
10:  4.4348e+03  4.4348e+03  6e-02  3e-08  3e-09
11:  4.4348e+03  4.4348e+03  1e-03  6e-10  5e-11
Optimal solution found.
```



```
     pcost       dcost       gap    pres   dres
 0: -8.2223e+13  4.5071e+13  2e+14  2e+06  2e+01
 1:  1.4294e+13 -6.7821e+11  2e+13  3e+04  2e-01
 2:  2.5270e+11 -1.8156e+08  3e+11  4e+02  3e-03
 3:  2.5980e+09  2.6019e+07  3e+09  4e+00  3e-05
 4:  9.5287e+07  2.6381e+07  8e+07  1e-01  8e-07
 5:  5.5354e+07  3.4041e+07  2e+07  3e-02  2e-07
 6:  5.2265e+07  3.7746e+07  2e+07  2e-02  1e-07
 7:  4.8992e+07  4.0548e+07  9e+06  6e-03  5e-08
 8:  4.6404e+07  4.2701e+07  4e+06  2e-03  2e-08
 9:  4.4997e+07  4.3915e+07  1e+06  9e-05  7e-10
10:  4.4776e+07  4.4016e+07  8e+05  4e-05  3e-10
11:  4.4418e+07  4.4290e+07  1e+05  6e-06  5e-11
12:  4.4349e+07  4.4346e+07  3e+03  1e-07  1e-12
13:  4.4347e+07  4.4347e+07  2e+02  8e-09  7e-12
14:  4.4347e+07  4.4347e+07  2e+00  8e-11  2e-12
Optimal solution found.
```



Explain your observations here:

The value of C affects the softness of the margins, allowing points to get between the gutters. Small value of C let more points in, with a softer margin and large values of C make the margins hard. It shows in how wide the "street" is. Smaller values for C make the street wider and larger values make it smaller. We can see that by comparing the first plot with C = 0.1 and the final one with C = 1000000, the first plot has a noticeably wide street in comparison with more points inside.

Assignment 5, Question 2: Physics Informed Neural Networks

Importing the necessary libraries

In [1]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, c
all drive.mount("/content/gdrive", force_remount=True).
```

In [2]:
```
cd /content/gdrive/MyDrive/
```

```
/content/gdrive/MyDrive
```

In [3]:
```
pwd
```

Out[3]:
```
'/content/gdrive/MyDrive'
```

In [4]:
```
cd /content/gdrive/MyDrive/MLAI-HW5
```

```
/content/gdrive/MyDrive/MLAI-HW5
```

In [5]:
```
!pip install pyDOE
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/cola
b-wheels/public/simple/
Requirement already satisfied: pyDOE in /usr/local/lib/python3.8/dist-packa
ges (0.3.8)
Requirement already satisfied: scipy in /usr/local/lib/python3.8/dist-packa
ges (from pyDOE) (1.7.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.8/dist-packa
ges (from pyDOE) (1.21.6)
```

In [6]:
```
import sys
import torch
import torch.nn as nn
from collections import OrderedDict
from pyDOE import lhs
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
from scipy.interpolate import griddata
from mpl_toolkits.axes_grid1 import make_axes_locatable
import matplotlib.gridspec as gridspec
import time
import matplotlib.pyplot as plt


np.random.seed(1234)
```

In [7]:
```
# Uncomment this to install the PyDOE package.
# !pip install pyDOE
```

In [8]:
```
# Enable use of the GPU
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')
```

In [9]:
```
steps=10000
lr=1e-1
layers = np.array([2,50,50,50,50,50,1]) #8 hidden layers
```

Physics-informed Neural Networks

We will use a Fully Connected Neural Network to solve this PDE. The network will take in two features as input, x, the spatial co-ordinate, and t, the time co-ordinate. From these two inputs, the network should output the solution to the PDE at that point in space and time. For instance, the solution to a PDE given by $u_t = -uu_x + \nu u_{xx}$ is going to be the function $u(y, t)$. Our goal would be to pass in a given y-coordinate and time value to this network, and output the corresponding value of f.

The first thing we'll do on the path towards implementing this is to define a *fully-connected* neural network with two nodes as input, one node as output, and several hidden layers in between. A good starting point would be to have 5 hidden layers with 50 neurons each, bias applied at each layer, and tanh() activation in between each linear layer.

In [10]:
```
# TO-DO: Construct a Fully Connected Neural Network to take in two features,
#Since all the layers are being initialized with the same number of neurons,
class FCNN(torch.nn.Module):

    def __init__(self,layers):
        super(FCNN, self).__init__()

        self.activation = nn.Tanh()

        'loss function'
        self.loss_function = nn.MSELoss(reduction ='mean')

        'Initialise neural network as a list using nn.Modulelist'
        self.linears = nn.ModuleList([nn.Linear(layers[i], layers[i+1]) for

        self.iter = 0

        'Xavier Normal Initialization'
        for i in range(len(layers)-1):

            nn.init.xavier_normal_(self.linears[i].weight.data, gain=1.0)

            # set biases to zero
            nn.init.zeros_(self.linears[i].bias.data)

    def forward(self, x, t):
        if torch.is_tensor(x) != True:
            x = torch.from_numpy(x)


        xt = torch.hstack((x[:,None], t[:,None]))

        #preprocessing input
        xt = (xt - lb_torch)/(ub_torch - lb_torch) #feature scaling
        #convert to float
        a = xt.float()
        for i in range(len(layers)-2):

            z = self.linears[i](a)

            a = self.activation(z)

        a = self.linears[-1](a)

        return a


        #return output
```

Dataset Configuration

After you've defined the network architecture above, the next step is to create a dataset. Since our goal is to train a model that can predict the PDE solution at some arbitrary space co-ordinate and time co-ordinate, we need to randomly sample some space and time co-ordinates to act as the training input for the model.

To do so, we'll first:

1. Define a grid spanning all possible x, t combinations. If we have N possible values of x and M possible values of t, we'll have $N_{samples} = N \times M$ combinations of x and t.
2. Reshape this grid into a feature matrix, of shape ($N_{samples}$, $N_{features}$). We'll use $N_{samples}$ from (1.), and we have two $N_{features}$ = 2 (x and t).
3. Extract the vectors from the grid that define the intitial condition, and the left and right boundaries of the domain. We'll also reshape these vectors into a feature matrix.
4. Randomly sample these points to create a training set.

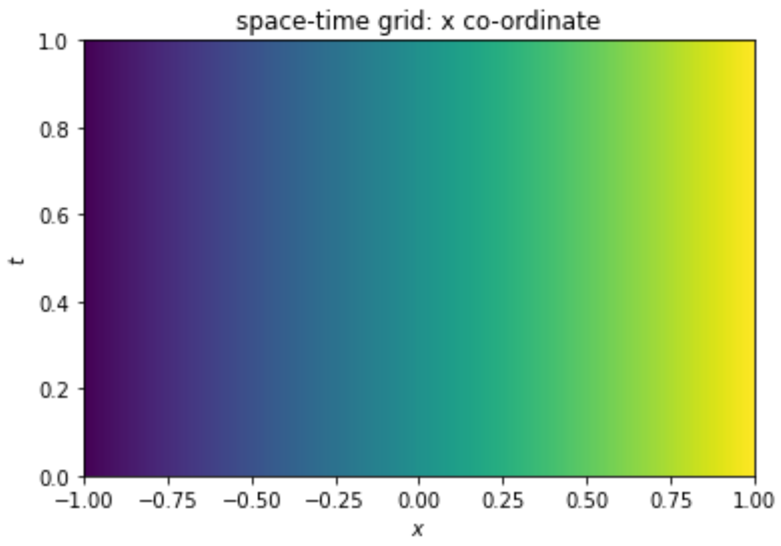
We'll use a direct MSE loss to make sure that the predictions of the network at the boundary and initial conditions are the correct values, and a loss based on the PDE residual for points in (x, t) space that do not lie on the boundary or initial conditions.

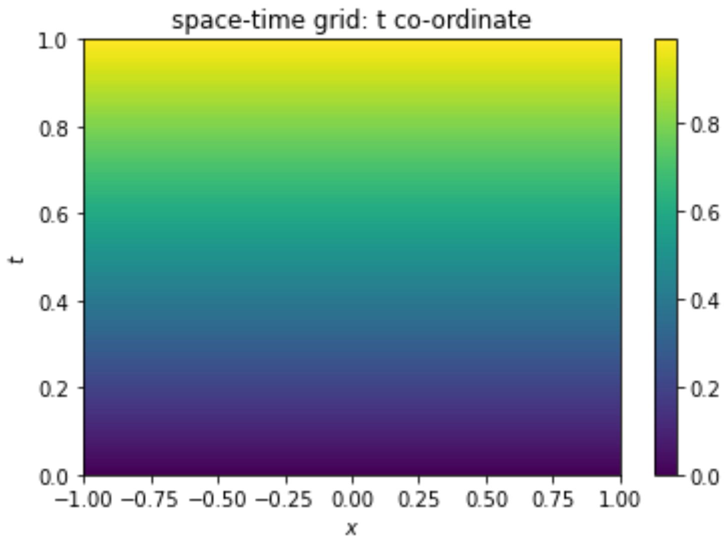
Some of the code to do this has been provided below, please fill in the blanks where indicated.

In [11]:
```python
# TO-DO: a) Defining the dataset.

# Nu defines the value for viscosity
nu = 0.01/np.pi


# Number of points to sample on u(x,t)
N_u = 100 # Number of samples for the MSE loss function
N_f = 10000 # Number of samples for the physics-based loss function

# Load in data
data = np.load('q2_data.npy')
gt_solution = data.T
print(gt_solution.shape)
print(gt_solution)

# 100 elements in the time dimension (Note, shape should be (100,1). You may
t_vector = np.linspace(0, 1, 100).reshape(100,1)
print(t_vector.shape)

# 256 elements in the space dimension (Note, shape should be (256,1). You ma
x_vector = np.linspace(-1, 1, 256).reshape(256,1)
print(x_vector.shape)

# Create a grid of x, t values using np.meshgrid(), and store them in arrays
# This is similar to what was done in HW4 to create the mesh for drawing the
xx, tt = np.meshgrid(x_vector,t_vector)
print(xx.shape)
print(tt.shape)
```

```
(100, 256)
[[0.         0.         0.         ... 1.         1.         1.        ]
 [0.56603457 0.12558765 0.0095487  ... 1.00374694 0.99385909 0.94468851]
 [0.79905436 0.40741454 0.10518197 ... 0.99923071 0.9977133  0.96299993]
 ...
 [0.95431941 0.95853579 0.96252125 ... 0.94052313 0.94530495 0.94991993]
 [0.94880498 0.95319305 0.95745857 ... 0.93438677 0.93939223 0.94415286]
 [0.9430292  0.94767609 0.95209251 ... 0.92815458 0.93326088 0.93825096]]
(100, 1)
(256, 1)
(100, 256)
(100, 256)
```

In [12]:
```python
plt.pcolormesh(xx,tt, xx)
plt.title('space-time grid: x co-ordinate')
plt.xlabel(r'$x$')
plt.ylabel(r'$t$')
plt.colorbar
plt.show()
plt.pcolormesh(xx, tt, tt)
plt.title('space-time grid: t co-ordinate')
plt.xlabel(r'$x$')
plt.ylabel(r'$t$')
plt.colorbar()
plt.show()
```

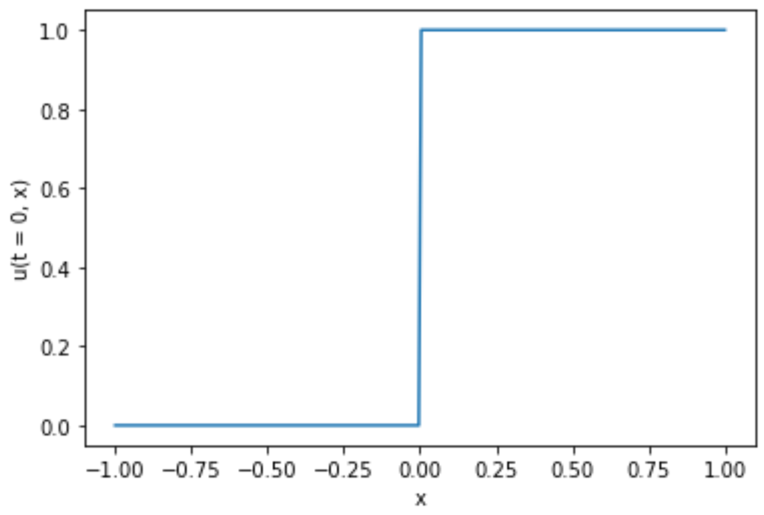

space-time grid: x co-ordinate
1.0
0.8
0.6
0.4
0.2
0.0
t
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x


space-time grid: t co-ordinate
1.0
0.8
0.6
0.4
0.2
0.0
t
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x
0.8
0.6
0.4
0.2
0.0

In [13]:
```
# TO-DO: Next, reshape the xx and tt arrays to be of shape (25600, 1), and s
xt_combined_flat = np.hstack((xx.flatten()[:,None], tt.flatten()[:,None]))
print(xt_combined_flat.shape)

# Reshape the gt_solution data array to be of shape (25600, 1) in an array c
u_flat = gt_solution.flatten('F')[:,None].reshape(25600,1) # u_flat is the g
print(u_flat.shape)
# Domain bounds
# lb should be the minimum x and t values, stored as a numpy array of shape
lb_numpy = xt_combined_flat[0].reshape(1,2)
print(lb_numpy.shape,np.min(lb_numpy),np.max(lb_numpy))
# ub should be the maximum x and t values, stored as a numpy array of shape
ub_numpy = xt_combined_flat[-1].reshape(1,2)
print(ub_numpy.shape,np.min(ub_numpy),np.max(ub_numpy))

# Define the initial conditions: t = 0.
# initial_conditions_xt is a 256 x 2 vector containing the x and t values at
# initial_condition_u is a 256 x 1 vector storing u at the initial condition

initial_condition_xt = np.hstack((xx[0,:][:,None], tt[0,:][:,None]))
initial_condition_u = gt_solution[0,:][:,None]

print(initial_condition_xt.shape, initial_condition_u.shape)

# Defining the left boundary: (x = -1).
# left_boundary_xt is a 100 x 2 vector containing the x and t values at the
# left_boundary_u is a 100 x 1 vector storing u at the left boundary.
# left_boundary_xt = xt_combined_flat[np.where(xt_combined_flat[:,0] == -1)]
# left_boundary_u = u_flat[np.where(xt_combined_flat[:,0] == -1)]
left_boundary_xt = np.hstack((xx[:,0][:,None], tt[:,0][:,None])) #L2
left_boundary_u = gt_solution[:,-1][:,None]

# Defining the right boundary: (x = 1)
# right_boundary_xt is a 100 x 2 vector containing the x and t values at the
# right_boundary_u is a 100 x 1 vector storing u at the right boundary.
# right_boundary_xt = xt_combined_flat[np.where(xt_combined_flat[:,0] == 1)]
# right_boundary_u = u_flat[np.where(xt_combined_flat[:,0] == 1)]
right_boundary_xt = np.hstack((xx[:,-1][:,None], tt[:,0][:,None])) #L3
right_boundary_u = gt_solution[:,0][:,None]

# Stack the initial condition, left boundary condition, and right boundary c
edge_samples_xt = np.vstack([initial_condition_xt, left_boundary_xt, right_b
# Sample randomly within the x,t space to create points for training
random_samples_xt = lb_numpy + (ub_numpy-lb_numpy)*lhs(2, N_f)
# Stack the [x,t] coordinates random samples with the boundary to create a t
all_samples_xt = np.vstack((random_samples_xt, edge_samples_xt))

# Stack the ground truth data at the boundary, initial conditions
edge_samples_u = np.vstack([initial_condition_u, left_boundary_u, right_boun
# Randomly sample a training set for the MSE loss from the data at the initi
idx = np.random.choice(edge_samples_xt.shape[0], N_u, replace=False)
train_samples_xt = edge_samples_xt[idx, :]
train_samples_u = edge_samples_u[idx,:]
```

```
(25600, 2)
(25600, 1)
(1, 2) -1.0 0.0
(1, 2) 1.0 1.0
(256, 2) (256, 1)
```

In [28]:
```
# Plot the initial condition as a function of x.
plt.plot(initial_condition_xt[:,0], initial_condition_u)
plt.xlabel('x')
plt.ylabel('u(t = 0, x)')
plt.show()
```



Next, we'll convert all of these data vectors into torch tensors. To convert a NumPy array to a Torch tensor, the baseline syntax is `torch.tensor(arr)`, where `arr` is the original numpy array. For each array, also convert it to a floating point array and put it on the GPU: `torch.tensor(arr).float().to(device)`.

You can specify manually if the gradient of a torch Tensor is also required with the argument `requires_grad = True` passed to the torch.tensor() function.

In [15]:
```
# TO-DO: Convert lower and upper bounds to torch tensors. No gradient is req
lb_torch = torch.tensor(lb_numpy).float().to(device)
ub_torch = torch.tensor(ub_numpy).float().to(device)

# Convert x and t data to torch Tensors. Specify that the gradient is requir
# train_samples_xt and all_samples_xt are 2-column matrices with x and t as
# be x, and t_u and t_f to just be t.

# assign x column of train_samples_xt to x_u, convert to torch tensor with g
x_boundary_train =  torch.tensor(train_samples_xt[:,0], requires_grad = True
# assign t column of train_samples_xt to t_u, convert to torch tensor with g
t_boundary_train = torch.tensor(train_samples_xt[:,1], requires_grad = True)
# assign x column of all_samples_xt to x_f, convert to torch tensor with gra
x_sampled_train = torch.tensor(all_samples_xt[:,0], requires_grad = True).fl
# assign t column of all_samples_xt to t_f, convert to torch tensor with gra
t_sampled_train = torch.tensor(all_samples_xt[:,1], requires_grad = True).fl

# Convert train_samples_u to torch tensor, no gradient is required.
u_boundary_train = torch.tensor(train_samples_u).float().to(device)

f_hat = torch.zeros(all_samples_xt.shape[0]).to(device)
```

Training

In [16]:
```
# b): Initialize the Deep Neural Network, and put it on the GPU.
fcnn = FCNN(layers = layers).to(device)

# Use the following optimizer to optimize your function.
optimizer = torch.optim.LBFGS(
    fcnn.parameters(),
    lr=1.0,
    max_iter=50000,
    max_eval=50000,
    history_size=50,
    tolerance_grad=1e-5,
    tolerance_change=1.0 * np.finfo(float).eps,
    line_search_fn="strong_wolfe"
)

iter = 0
# c) Calling the network and calculating loss
## The function net_u takes in a neural network (fcnn), x, and t, and return
def net_u(fcnn, x, t):
    # Call the fcnn network with the x and t co-ordinates, return the predic
    pred_u = fcnn.forward(x, t)

    return pred_u
    # return network output

## The function net_f, takes in the net_u function, x, and t, and computes t
def net_f(x, t):
    # Calculate the residual of the PDE, using the gradients computed via au

    # Step 1: Using the function net_u, calculate the predicted u variable.
    pred_u = net_u(fcnn, x, t).flatten()
    # Step 2: Compute the gradients used in the Burgers' equation PDE using
    # ut = torch.autograd.grad(pred_u, t)[0]
    u_t = torch.autograd.grad( pred_u, t, grad_outputs=torch.ones_like(pred_
    u_x = torch.autograd.grad( pred_u, x, grad_outputs=torch.ones_like(pred_
    u_xx = torch.autograd.grad( u_x, x, grad_outputs=torch.ones_like(u_x), r
    # Step 3: Calculate the residual.
    f = u_t + (pred_u * u_x) - (nu * u_xx)
    # return PDE residual
    return f

iteration = 0

## The loss function will calculate the loss as a combination of the MSE los
def loss_func():
    optimizer.zero_grad()
    # Predict the solution along the intitial and boundary conditions
    u_pred = net_u(fcnn, x_boundary_train, t_boundary_train)
    # Predict the solution at the sampled co-location points
    f_pred = net_f(x_sampled_train, t_sampled_train)
    # Compute MSE loss on (x,t) points that lie on initial and boundary cond
    mse_loss = fcnn.loss_function(u_pred, u_boundary_train)
    pde_loss = fcnn.loss_function(f_pred, f_hat)
    total_loss = mse_loss + pde_loss

    global iteration # iteration keeps track of the current iteration count
    # Uncomment line below to backpropagate loss
    total_loss.backward()


    # Print out iteration progress by uncommenting the following line:
    iteration = iteration + 1
    if iteration % 100 == 0:
        print(
            'Iter %d, Loss: %.5e, Loss_u: %.5e, Loss_f: %.5e' % (iteration,
        )
    # pass
    return total_loss

# This initializes the gradients for training
fcnn.train()

# This carries out the entire optimization process with L-BFGS, by calling t
optimizer.step(loss_func)
# loss_func()
```

```
Iter 100, Loss: 5.38543e-02, Loss_u: 4.41125e-02, Loss_f: 9.74177e-03
Iter 200, Loss: 3.43765e-02, Loss_u: 2.71757e-02, Loss_f: 7.20083e-03
Iter 300, Loss: 1.61490e-02, Loss_u: 1.08801e-02, Loss_f: 5.26886e-03
Iter 400, Loss: 3.44884e-03, Loss_u: 1.84790e-03, Loss_f: 1.60095e-03
Iter 500, Loss: 2.29104e-03, Loss_u: 1.36941e-03, Loss_f: 9.21639e-04
Iter 600, Loss: 1.63524e-03, Loss_u: 8.87295e-04, Loss_f: 7.47944e-04
Iter 700, Loss: 1.13153e-03, Loss_u: 5.85373e-04, Loss_f: 5.46156e-04
Iter 800, Loss: 7.99728e-04, Loss_u: 3.70006e-04, Loss_f: 4.29723e-04
Iter 900, Loss: 6.14948e-04, Loss_u: 3.19333e-04, Loss_f: 2.95615e-04
Iter 1000, Loss: 4.82609e-04, Loss_u: 2.44938e-04, Loss_f: 2.37672e-04
Iter 1100, Loss: 3.76968e-04, Loss_u: 1.77063e-04, Loss_f: 1.99905e-04
Iter 1200, Loss: 2.92274e-04, Loss_u: 1.43637e-04, Loss_f: 1.48637e-04
Iter 1300, Loss: 2.24913e-04, Loss_u: 9.86304e-05, Loss_f: 1.26283e-04
Iter 1400, Loss: 1.84490e-04, Loss_u: 8.02174e-05, Loss_f: 1.04272e-04
Iter 1500, Loss: 1.40038e-04, Loss_u: 5.91863e-05, Loss_f: 8.08517e-05
Iter 1600, Loss: 1.18996e-04, Loss_u: 4.96556e-05, Loss_f: 6.93401e-05
Iter 1700, Loss: 1.04769e-04, Loss_u: 4.89529e-05, Loss_f: 5.58164e-05
Iter 1800, Loss: 9.06354e-05, Loss_u: 4.20453e-05, Loss_f: 4.85901e-05
Iter 1900, Loss: 7.76196e-05, Loss_u: 3.14872e-05, Loss_f: 4.61324e-05
Iter 2000, Loss: 6.55367e-05, Loss_u: 2.44066e-05, Loss_f: 4.11301e-05
Iter 2100, Loss: 5.87487e-05, Loss_u: 2.24035e-05, Loss_f: 3.63452e-05
Iter 2200, Loss: 5.07182e-05, Loss_u: 1.90201e-05, Loss_f: 3.16982e-05
Iter 2300, Loss: 4.23400e-05, Loss_u: 1.52496e-05, Loss_f: 2.70904e-05
Iter 2400, Loss: 3.73472e-05, Loss_u: 1.28043e-05, Loss_f: 2.45429e-05
Iter 2500, Loss: 3.20427e-05, Loss_u: 1.17894e-05, Loss_f: 2.02532e-05
Iter 2600, Loss: 2.75657e-05, Loss_u: 1.05173e-05, Loss_f: 1.70484e-05
Iter 2700, Loss: 2.38084e-05, Loss_u: 9.14622e-06, Loss_f: 1.46622e-05
Iter 2800, Loss: 2.17492e-05, Loss_u: 8.46563e-06, Loss_f: 1.32835e-05
Iter 2900, Loss: 1.96642e-05, Loss_u: 8.02023e-06, Loss_f: 1.16440e-05
Iter 3000, Loss: 1.75487e-05, Loss_u: 6.92818e-06, Loss_f: 1.06205e-05
Iter 3100, Loss: 1.60226e-05, Loss_u: 6.05920e-06, Loss_f: 9.96342e-06
Iter 3200, Loss: 1.48008e-05, Loss_u: 5.80416e-06, Loss_f: 8.99659e-06
Iter 3300, Loss: 1.36202e-05, Loss_u: 5.25973e-06, Loss_f: 8.36043e-06
Iter 3400, Loss: 1.29193e-05, Loss_u: 5.18140e-06, Loss_f: 7.73790e-06
Iter 3500, Loss: 1.24038e-05, Loss_u: 4.95774e-06, Loss_f: 7.44607e-06
Iter 3600, Loss: 1.17858e-05, Loss_u: 4.88437e-06, Loss_f: 6.90140e-06
Iter 3700, Loss: 1.11122e-05, Loss_u: 4.66042e-06, Loss_f: 6.45175e-06
Iter 3800, Loss: 1.05626e-05, Loss_u: 4.53027e-06, Loss_f: 6.03229e-06
Iter 3900, Loss: 9.92753e-06, Loss_u: 4.37424e-06, Loss_f: 5.55328e-06
Iter 4000, Loss: 9.41131e-06, Loss_u: 4.01287e-06, Loss_f: 5.39843e-06
Iter 4100, Loss: 8.93628e-06, Loss_u: 3.82382e-06, Loss_f: 5.11246e-06
Iter 4200, Loss: 8.29212e-06, Loss_u: 3.55098e-06, Loss_f: 4.74114e-06
Iter 4300, Loss: 7.87020e-06, Loss_u: 3.53277e-06, Loss_f: 4.33743e-06
Iter 4400, Loss: 7.65675e-06, Loss_u: 3.39220e-06, Loss_f: 4.26455e-06
Iter 4500, Loss: 7.48157e-06, Loss_u: 3.35707e-06, Loss_f: 4.12450e-06
```

Out[16]:
```
tensor(1.0512, grad_fn=<AddBackward0>)
```

In [17]:
```python
# d) Given a 2-D array of space and time variables, predict the correspondin
def predict(xt):
    # As before, separate xt into vectors and convert these vectors of time
    x_torch = torch.tensor(xt[:,0], requires_grad = True).float().to(device)
    t_torch = torch.tensor(xt[:,1], requires_grad = True).float().to(device)
    fcnn.eval()
    # TO-DO: get predicted u and PDE residual from networks.
    u_pred = net_u(fcnn, x_torch, t_torch)
    f_pred = net_f(x_torch, t_torch)
    u = u_pred.detach().cpu().numpy()
    f = f_pred.detach().cpu().numpy()
    return u, f
```

Compute the following error metrics

Normalized L_2 Error, E_{L_2}:

$$E_{L_2} = \frac{\sum_{x,t}\left(u(x,t)_{gt} - u(x,t)_{pred}\right)^2}{\sum_{x,t} u(x,t)_{gt}^2}$$

L_1 Error, E_{L_1}

$$E_{L_1} = \sum_{x,t} \left|u(x,t)_{gt} - u(x,t)_{pred}\right|$$

In [18]:
```python
# Predict the PDE solution for every combination of x and t

u_pred, f_pred = predict(xt_combined_flat)
u_gt = gt_solution.flatten()
# Compute the normalized L2 error of the solution
error_el2 = np.sum((u_gt - u_pred)**2)/np.sum(u_gt ** 2)
print('Normalized L2 Error: %e' % (error_el2))
# Compute the L1 error of the solution.
error_el1 = np.sum(np.absolute(u_gt - u_pred))
```

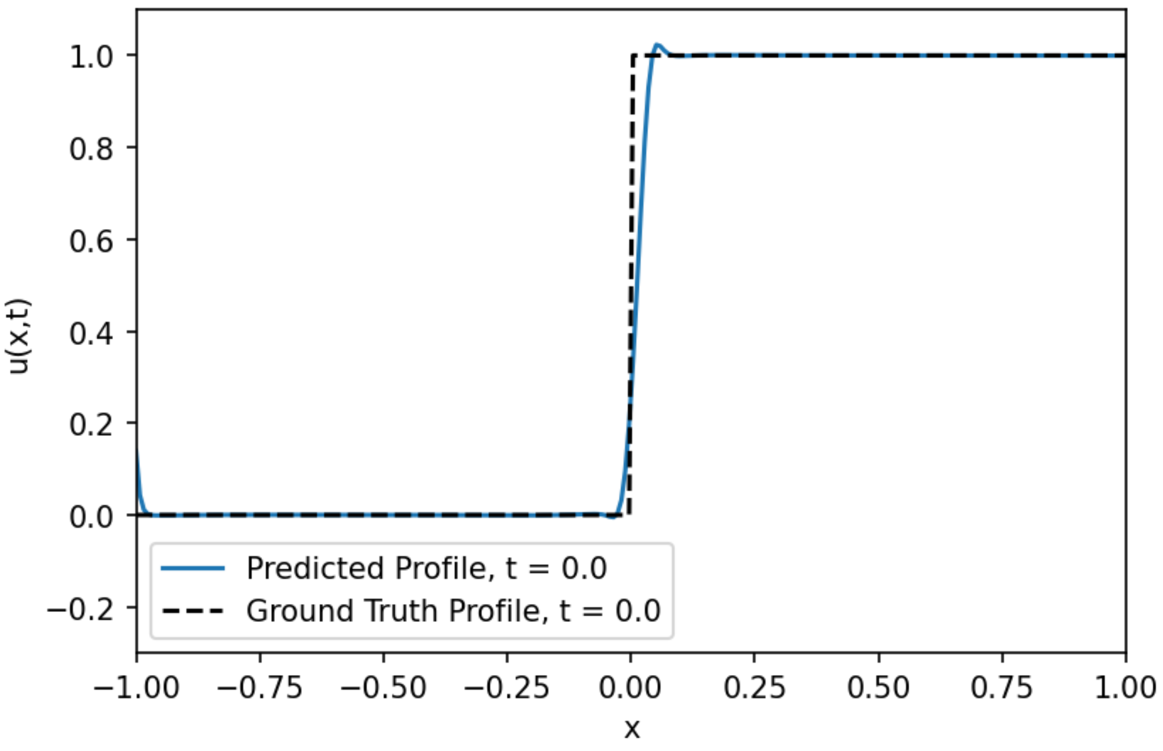
```
Normalized L2 Error: 2.249176e+04
```

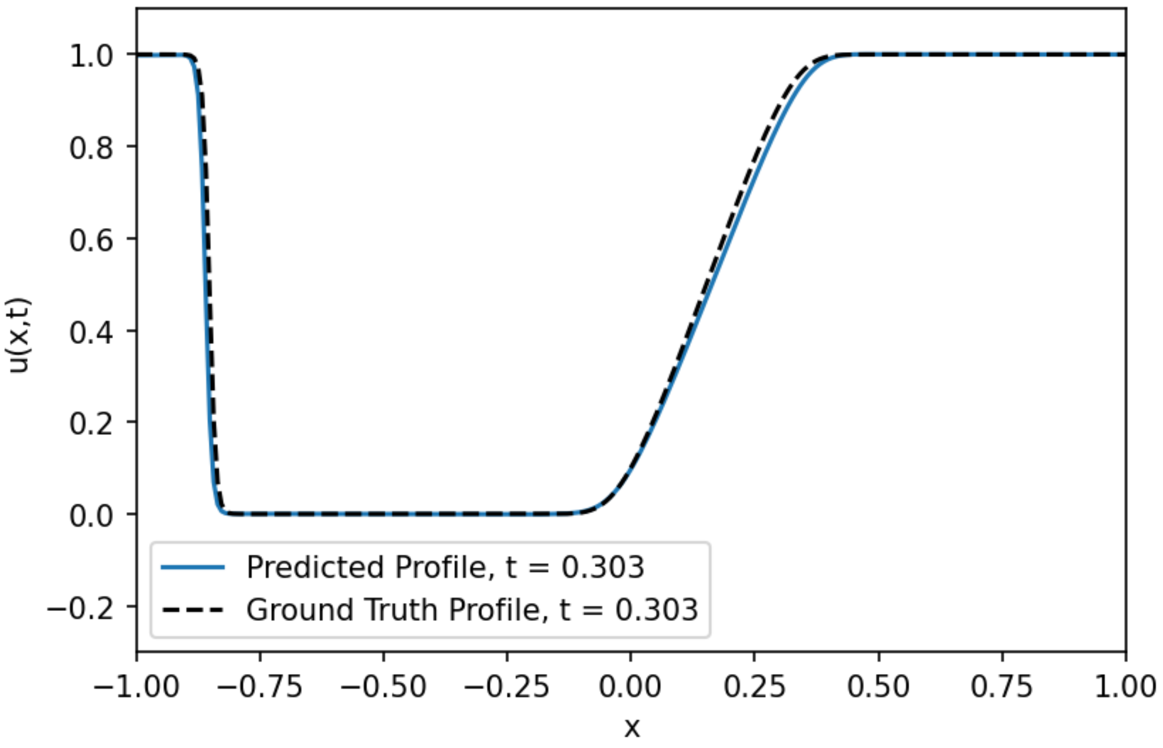
Plotting Results

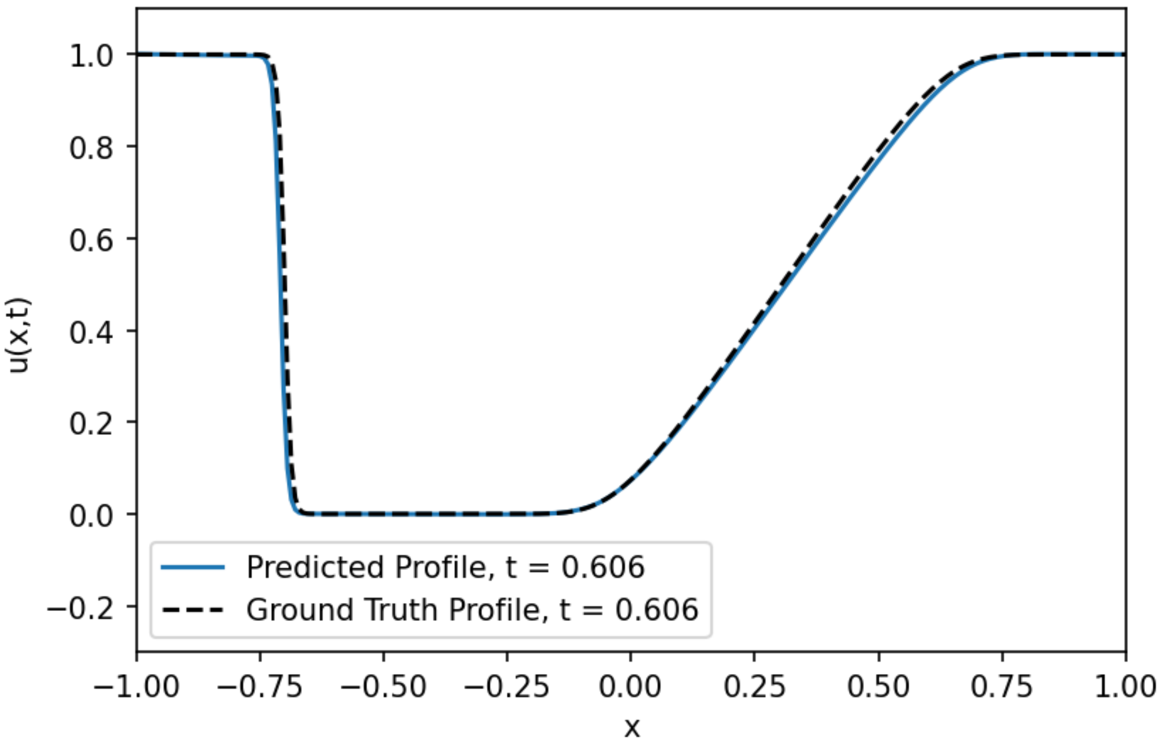
Based on your results from the previous question, we're now going to plot the evolution of the Burgers' equation over time, as well as some comparisons between the ground truth solution and the predicted solution.

Figure 1: Plotting a comparison of the ground truth solution to the predicted solution at various times.

```
In [19]: # Interpolate array for plotting.
u_pred_grid = griddata(xt_combined_flat, u_pred.flatten(), (xx, tt), method=
Error = np.abs(gt_solution - u_pred_grid)

for i in range(0, 100, 30):
    plt.figure(dpi = 150)
    plt.plot(x_vector[:,0],u_pred_grid[i,:], label = 'Predicted Profile, t =
    plt.plot(x_vector[:,0],gt_solution[i,:], 'k--', label = 'Ground Truth Pr
    plt.legend()
    plt.ylim([-0.3,1.1])
    plt.xlim([-1,1])
    plt.xlabel('x')
    plt.ylabel('u(x,t)')
    plt.show()
```




Predicted Profile, t = 0.303
Ground Truth Profile, t = 0.303
u(x,t)
x


Predicted Profile, t = 0.606
Ground Truth Profile, t = 0.606
u(x,t)
x

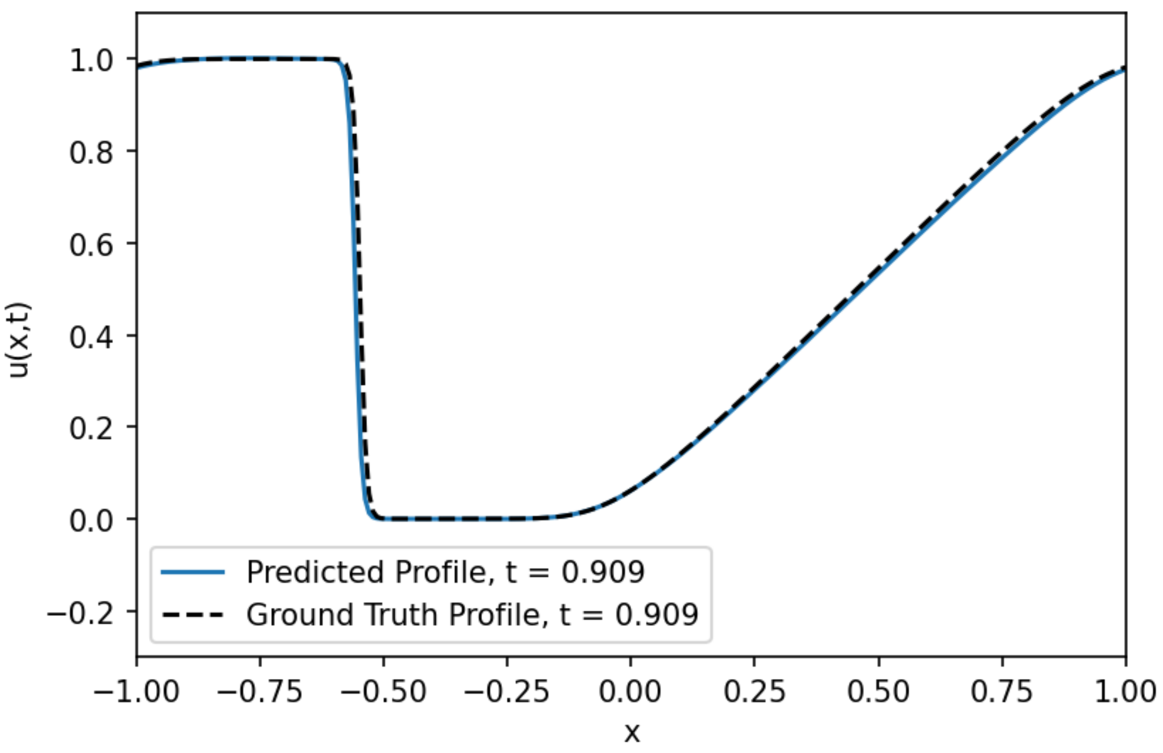
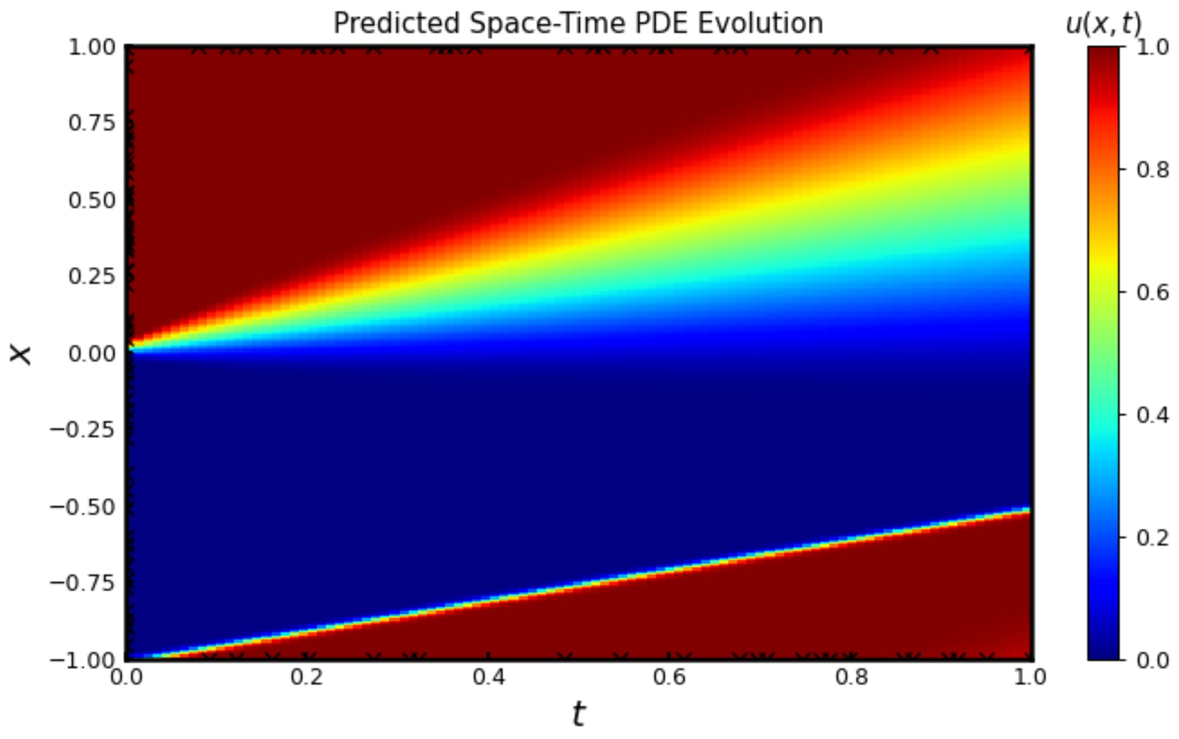
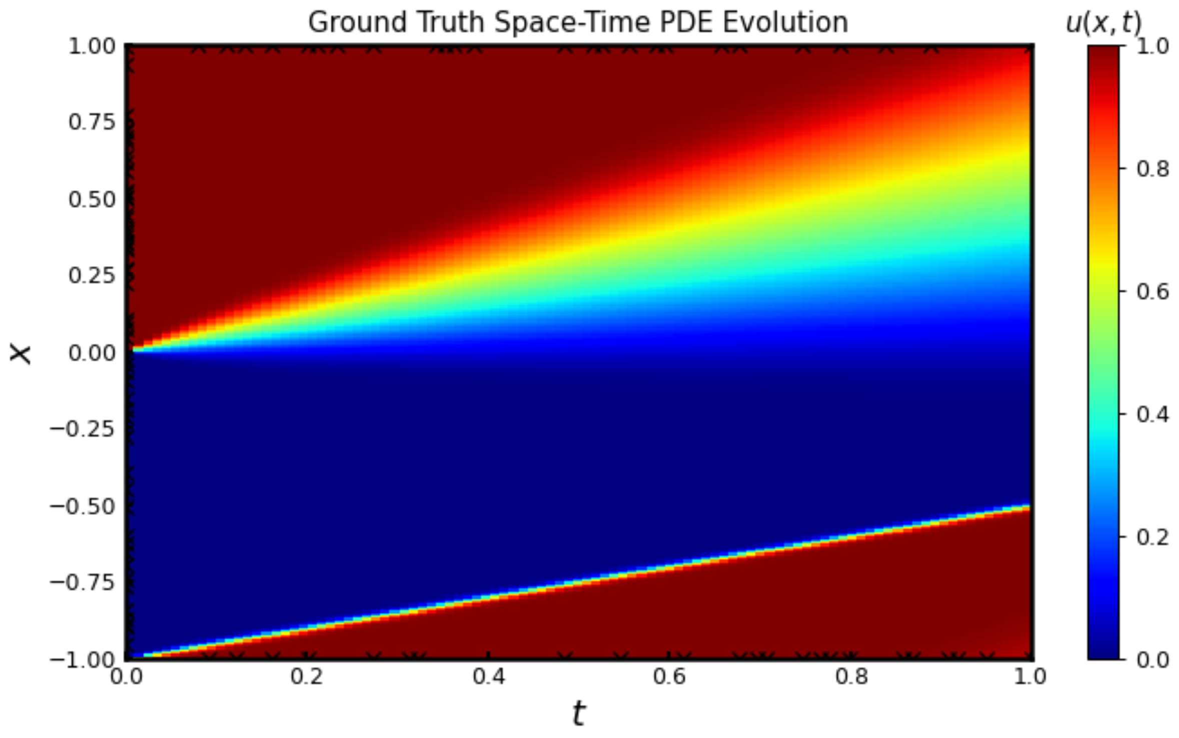


Figure 2: Plotting the time evolution of the 1-D Burgers equation as a 2-D image

In [20]:

```python
plt.figure(dpi = 90, figsize  = [9,5])
plt.pcolormesh(tt.T,xx.T,u_pred_grid.T, cmap= 'jet', vmin  =0, vmax = 1)
plt.ylabel(r'$x$', fontsize = 16)
plt.xlabel(r'$t$', fontsize = 16)

plt.plot(train_samples_xt[:,1], train_samples_xt[:,0], 'kx')


plt.title('Predicted Space-Time PDE Evolution')
ax = plt.gca()
for axis in ['top', 'bottom', 'left', 'right']:
    ax.spines[axis].set_linewidth(2.0)
plt.tick_params(direction = 'in', width = 1.5)
clb= plt.colorbar()
clb.ax.set_title(r'$u (x,t)$')

plt.show()
plt.figure(dpi = 90, figsize  = [9,5])
plt.pcolormesh(tt.T,xx.T,gt_solution.T, cmap= 'jet', vmin  =0, vmax = 1)
plt.ylabel(r'$x$', fontsize = 16)
plt.xlabel(r'$t$', fontsize = 16)

plt.plot(train_samples_xt[:,1], train_samples_xt[:,0], 'kx')

clb = plt.colorbar()
clb.ax.set_title(r'$u (x,t)$')
plt.title('Ground Truth Space-Time PDE Evolution')
ax = plt.gca()
for axis in ['top', 'bottom', 'left', 'right']:
    ax.spines[axis].set_linewidth(2.0)
plt.tick_params(direction = 'in', width = 1.5)
plt.show()


plt.figure(dpi = 90, figsize  = [9,5])
plt.pcolormesh(tt.T,xx.T,Error.T, cmap= 'jet', vmin = 0, vmax = 1)
plt.ylabel(r'$x$', fontsize = 16)
plt.xlabel(r'$t$', fontsize = 16)

plt.plot(train_samples_xt[:,1], train_samples_xt[:,0], 'kx')

clb = plt.colorbar()
clb.ax.set_title(r'$u (x,t)$')
plt.title('Approximation Error')
ax = plt.gca()
for axis in ['top', 'bottom', 'left', 'right']:
    ax.spines[axis].set_linewidth(2.0)
plt.tick_params(direction = 'in', width = 1.5)
plt.show()
```


Predicted Space-Time PDE Evolution
u(x,t)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
x
0.0
0.2
0.4
0.6
0.8
1.0
t
0.0
0.2
0.4
0.6
0.8
1.0


Ground Truth Space-Time PDE Evolution
u(x,t)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
x
0.0
0.2
0.4
0.6
0.8
1.0
t
0.0
0.2
0.4
0.6
0.8
1.0

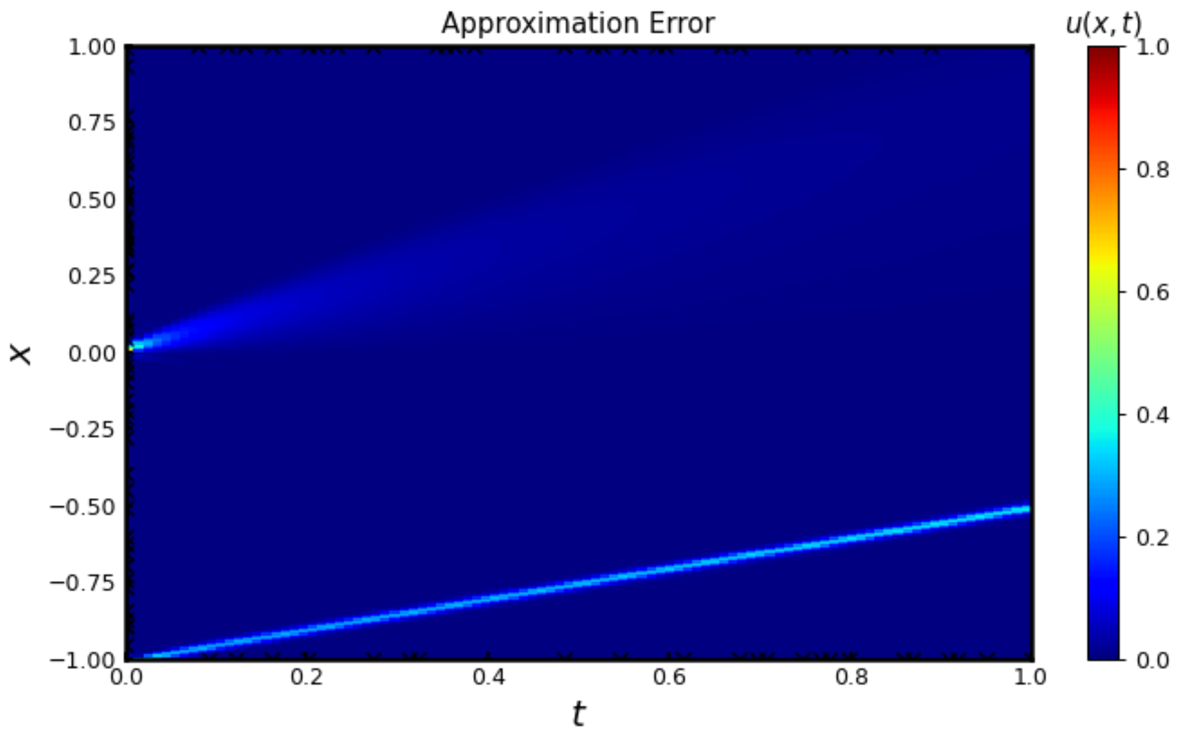


Figure 3: Plotting the solution at t = 1 s for $\nu = 1.0, 0.1, \frac{0.01}{\pi}$ on the same figure.

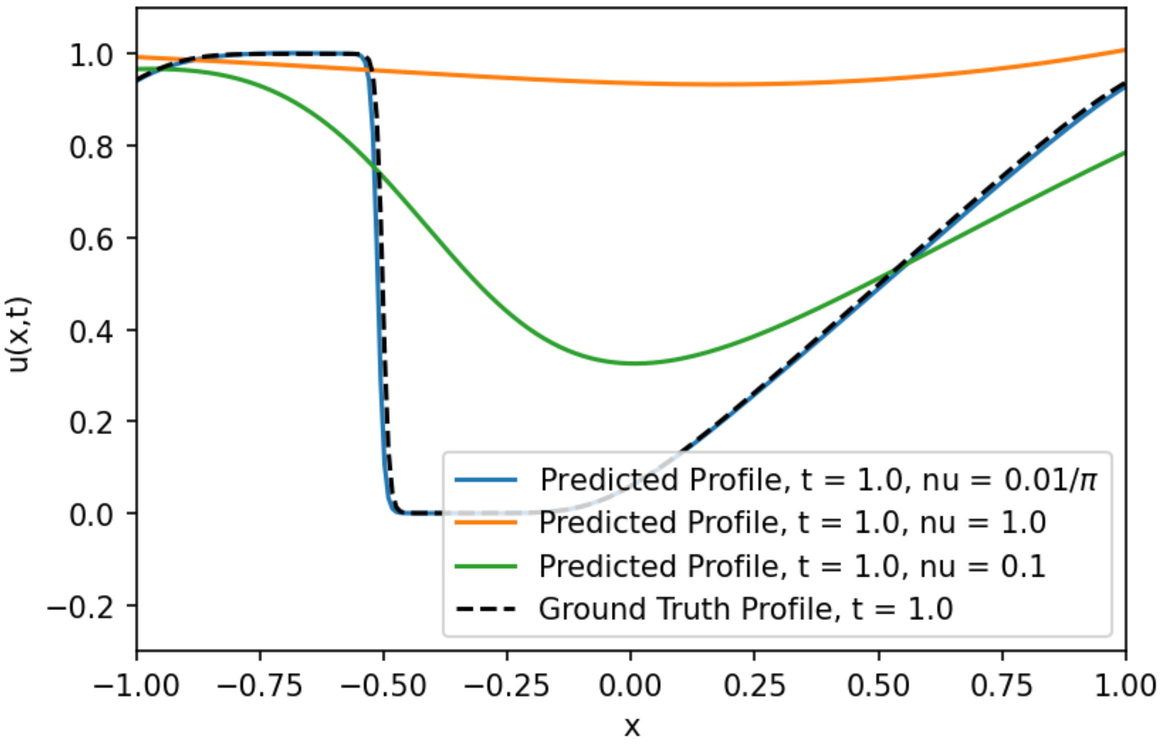
In [21]:
```
nu = 1.0
fcnn = FCNN(layers = layers).to(device)
optimizer = torch.optim.LBFGS(
    fcnn.parameters(),
    lr=1.0,
    max_iter=50000,
    max_eval=50000,
    history_size=50,
    tolerance_grad=1e-5,
    tolerance_change=1.0 * np.finfo(float).eps,
    line_search_fn="strong_wolfe"
)
iter = 0
iteration = 0
fcnn.train()

# This carries out the entire optimization process with L-BFGS, by calling t
optimizer.step(loss_func)

u_pred1, f_pred1 = predict(xt_combined_flat)
```

```
Iter 100, Loss: 3.11683e-02, Loss_u: 2.83997e-02, Loss_f: 2.76866e-03
Iter 200, Loss: 2.45538e-02, Loss_u: 2.37977e-02, Loss_f: 7.56042e-04
Iter 300, Loss: 2.25243e-02, Loss_u: 2.13685e-02, Loss_f: 1.15582e-03
Iter 400, Loss: 2.14146e-02, Loss_u: 1.98038e-02, Loss_f: 1.61083e-03
Iter 500, Loss: 1.97308e-02, Loss_u: 1.79683e-02, Loss_f: 1.76247e-03
Iter 600, Loss: 1.74327e-02, Loss_u: 1.60033e-02, Loss_f: 1.42943e-03
Iter 700, Loss: 1.53105e-02, Loss_u: 1.43197e-02, Loss_f: 9.90798e-04
Iter 800, Loss: 1.43050e-02, Loss_u: 1.29969e-02, Loss_f: 1.30805e-03
Iter 900, Loss: 1.36501e-02, Loss_u: 1.25212e-02, Loss_f: 1.12892e-03
Iter 1000, Loss: 1.27977e-02, Loss_u: 1.16341e-02, Loss_f: 1.16355e-03
Iter 1100, Loss: 1.21068e-02, Loss_u: 1.08090e-02, Loss_f: 1.29780e-03
Iter 1200, Loss: 1.12949e-02, Loss_u: 1.01621e-02, Loss_f: 1.13283e-03
Iter 1300, Loss: 1.06618e-02, Loss_u: 9.89084e-03, Loss_f: 7.70944e-04
Iter 1400, Loss: 1.16151e-02, Loss_u: 9.37138e-03, Loss_f: 2.24377e-03
Iter 1500, Loss: 9.98078e-03, Loss_u: 9.14139e-03, Loss_f: 8.39395e-04
Iter 1600, Loss: 9.74496e-03, Loss_u: 8.90081e-03, Loss_f: 8.44148e-04
Iter 1700, Loss: 9.55934e-03, Loss_u: 8.85661e-03, Loss_f: 7.02736e-04
Iter 1800, Loss: 9.38934e-03, Loss_u: 8.70640e-03, Loss_f: 6.82941e-04
Iter 1900, Loss: 9.25928e-03, Loss_u: 8.64244e-03, Loss_f: 6.16840e-04
Iter 2000, Loss: 9.08366e-03, Loss_u: 8.49210e-03, Loss_f: 5.91553e-04
Iter 2100, Loss: 8.93176e-03, Loss_u: 8.36418e-03, Loss_f: 5.67580e-04
Iter 2200, Loss: 8.81227e-03, Loss_u: 8.21099e-03, Loss_f: 6.01283e-04
Iter 2300, Loss: 8.62738e-03, Loss_u: 7.98270e-03, Loss_f: 6.44677e-04
Iter 2400, Loss: 8.50571e-03, Loss_u: 7.80624e-03, Loss_f: 6.99474e-04
Iter 2500, Loss: 8.39990e-03, Loss_u: 7.69184e-03, Loss_f: 7.08063e-04
Iter 2600, Loss: 8.29886e-03, Loss_u: 7.58831e-03, Loss_f: 7.10554e-04
Iter 2700, Loss: 8.16311e-03, Loss_u: 7.31648e-03, Loss_f: 8.46634e-04
Iter 2800, Loss: 7.95533e-03, Loss_u: 7.17875e-03, Loss_f: 7.76583e-04
Iter 2900, Loss: 7.85236e-03, Loss_u: 7.14173e-03, Loss_f: 7.10637e-04
Iter 3000, Loss: 7.73952e-03, Loss_u: 6.98861e-03, Loss_f: 7.50910e-04
Iter 3100, Loss: 7.65011e-03, Loss_u: 6.92201e-03, Loss_f: 7.28103e-04
Iter 3200, Loss: 7.54145e-03, Loss_u: 6.82025e-03, Loss_f: 7.21198e-04
Iter 3300, Loss: 7.44667e-03, Loss_u: 6.74747e-03, Loss_f: 6.99201e-04
Iter 3400, Loss: 7.36967e-03, Loss_u: 6.69375e-03, Loss_f: 6.75922e-04
Iter 3500, Loss: 7.25534e-03, Loss_u: 6.57116e-03, Loss_f: 6.84182e-04
```

In [22]:
```python
nu = 0.1
fcnn = FCNN(layers = layers).to(device)
optimizer = torch.optim.LBFGS(
    fcnn.parameters(),
    lr=1.0,
    max_iter=50000,
    max_eval=50000,
    history_size=50,
    tolerance_grad=1e-5,
    tolerance_change=1.0 * np.finfo(float).eps,
    line_search_fn="strong_wolfe"
)
iter = 0
iteration = 0
fcnn.train()

# This carries out the entire optimization process with L-BFGS, by calling t
optimizer.step(loss_func)

u_pred2, f_pred2 = predict(xt_combined_flat)
```

```
Iter 100, Loss: 2.67985e-02, Loss_u: 2.37101e-02, Loss_f: 3.08838e-03
Iter 200, Loss: 1.98613e-02, Loss_u: 1.77869e-02, Loss_f: 2.07446e-03
Iter 300, Loss: 1.40118e-02, Loss_u: 1.11976e-02, Loss_f: 2.81419e-03
Iter 400, Loss: 9.09359e-03, Loss_u: 7.50730e-03, Loss_f: 1.58629e-03
Iter 500, Loss: 6.12360e-03, Loss_u: 4.84813e-03, Loss_f: 1.27547e-03
Iter 600, Loss: 4.68538e-03, Loss_u: 3.34009e-03, Loss_f: 1.34530e-03
Iter 700, Loss: 3.75418e-03, Loss_u: 2.76382e-03, Loss_f: 9.90363e-04
Iter 800, Loss: 3.20621e-03, Loss_u: 2.40322e-03, Loss_f: 8.02989e-04
Iter 900, Loss: 2.90996e-03, Loss_u: 2.30399e-03, Loss_f: 6.05970e-04
Iter 1000, Loss: 2.75781e-03, Loss_u: 2.21124e-03, Loss_f: 5.46575e-04
Iter 1100, Loss: 2.60512e-03, Loss_u: 2.13431e-03, Loss_f: 4.70811e-04
Iter 1200, Loss: 2.51155e-03, Loss_u: 2.08214e-03, Loss_f: 4.29417e-04
Iter 1300, Loss: 2.39246e-03, Loss_u: 1.96820e-03, Loss_f: 4.24268e-04
Iter 1400, Loss: 2.29717e-03, Loss_u: 1.87191e-03, Loss_f: 4.25255e-04
Iter 1500, Loss: 2.20910e-03, Loss_u: 1.79512e-03, Loss_f: 4.13975e-04
Iter 1600, Loss: 2.14359e-03, Loss_u: 1.72224e-03, Loss_f: 4.21353e-04
Iter 1700, Loss: 2.07857e-03, Loss_u: 1.68360e-03, Loss_f: 3.94971e-04
Iter 1800, Loss: 1.99428e-03, Loss_u: 1.59676e-03, Loss_f: 3.97517e-04
Iter 1900, Loss: 1.91697e-03, Loss_u: 1.55937e-03, Loss_f: 3.57607e-04
Iter 2000, Loss: 1.85661e-03, Loss_u: 1.50336e-03, Loss_f: 3.53247e-04
Iter 2100, Loss: 1.82083e-03, Loss_u: 1.48529e-03, Loss_f: 3.35542e-04
Iter 2200, Loss: 1.78680e-03, Loss_u: 1.43785e-03, Loss_f: 3.48956e-04
Iter 2300, Loss: 1.75430e-03, Loss_u: 1.42170e-03, Loss_f: 3.32595e-04
Iter 2400, Loss: 1.72070e-03, Loss_u: 1.37660e-03, Loss_f: 3.44095e-04
Iter 2500, Loss: 1.69655e-03, Loss_u: 1.36460e-03, Loss_f: 3.31951e-04
Iter 2600, Loss: 1.67795e-03, Loss_u: 1.34267e-03, Loss_f: 3.35280e-04
Iter 2700, Loss: 1.65900e-03, Loss_u: 1.34144e-03, Loss_f: 3.17568e-04
Iter 2800, Loss: 1.63794e-03, Loss_u: 1.32556e-03, Loss_f: 3.12378e-04
Iter 2900, Loss: 1.61997e-03, Loss_u: 1.30916e-03, Loss_f: 3.10805e-04
Iter 3000, Loss: 1.59846e-03, Loss_u: 1.29852e-03, Loss_f: 2.99938e-04
Iter 3100, Loss: 1.58010e-03, Loss_u: 1.29086e-03, Loss_f: 2.89240e-04
Iter 3200, Loss: 1.56301e-03, Loss_u: 1.28953e-03, Loss_f: 2.73474e-04
Iter 3300, Loss: 1.54972e-03, Loss_u: 1.28597e-03, Loss_f: 2.63753e-04
Iter 3400, Loss: 1.53587e-03, Loss_u: 1.27789e-03, Loss_f: 2.57978e-04
Iter 3500, Loss: 1.52527e-03, Loss_u: 1.27456e-03, Loss_f: 2.50715e-04
Iter 3600, Loss: 1.51432e-03, Loss_u: 1.27621e-03, Loss_f: 2.38111e-04
Iter 3700, Loss: 1.50524e-03, Loss_u: 1.27509e-03, Loss_f: 2.30144e-04
Iter 3800, Loss: 1.49595e-03, Loss_u: 1.26625e-03, Loss_f: 2.29698e-04
Iter 3900, Loss: 1.48760e-03, Loss_u: 1.26366e-03, Loss_f: 2.23933e-04
Iter 4000, Loss: 1.48083e-03, Loss_u: 1.25967e-03, Loss_f: 2.21168e-04
Iter 4100, Loss: 1.47507e-03, Loss_u: 1.25733e-03, Loss_f: 2.17748e-04
Iter 4200, Loss: 1.47036e-03, Loss_u: 1.25085e-03, Loss_f: 2.19510e-04
Iter 4300, Loss: 1.46473e-03, Loss_u: 1.24672e-03, Loss_f: 2.18006e-04
Iter 4400, Loss: 1.45920e-03, Loss_u: 1.24564e-03, Loss_f: 2.13557e-04
```

In [27]:
```
u_pred_grid = griddata(xt_combined_flat, u_pred.flatten(), (xx, tt), method=
u_pred_grid1 = griddata(xt_combined_flat, u_pred1.flatten(), (xx, tt), metho
u_pred_grid2 = griddata(xt_combined_flat, u_pred2.flatten(), (xx, tt), metho

Error = np.abs(gt_solution - u_pred_grid)
Error1 = np.abs(gt_solution - u_pred_grid1)
Error2 = np.abs(gt_solution - u_pred_grid2)

plt.figure(dpi = 150)
plt.plot(x_vector[:,0],u_pred_grid[-1,:], label = 'Predicted Profile, t = {:
plt.plot(x_vector[:,0],u_pred_grid1[-1,:], label = 'Predicted Profile, t = {
plt.plot(x_vector[:,0],u_pred_grid2[-1,:], label = 'Predicted Profile, t = {
plt.plot(x_vector[:,0],gt_solution[-1,:], 'k--', label = 'Ground Truth Profi
plt.legend()
plt.ylim([-0.3,1.1])
plt.xlim([-1,1])
plt.xlabel('x')
plt.ylabel('u(x,t)')
plt.show()
```



It looks like the model performs better with smaller nu values, at 1.0 it doesn't even get close to the ground truth. 0.1 gets closer in its shape from the plot, but clearly 0.01/pi allows the model to converge very close to the ground truth.

Pytorch's AutoGrad automatically calculates parameter's gradients for backprop and then it can update the parameters with their respective gradients. This is extremely useful to solve differentiation problems and models like PINNs.