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
```

```
Requirement already satisfied: cvxopt in c:\users\gandi\anaconda3\lib\site-packages (1.
3.0)
```

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[:, 2]

import cvxopt.solvers
from datasets import get_dataset
import linearly_separable as ls

y = y.reshape(-1, 1)
m = X.shape[0]

z = y * X
P = cvxopt.matrix(np.dot(z, z.T))
q = cvxopt.matrix(-1 * np.ones((m, 1)))
A = cvxopt.matrix(y.reshape(1, -1))
b = cvxopt.matrix(0.0)

G = cvxopt.matrix(-1 * np.eye(m))
h = cvxopt.matrix(np.zeros(m))

solution = cvxopt.solvers.qp(P, q, G, h, A, b)

multipliers = np.ravel(solution['x'])
has_positive_multiplier = multipliers > 1e-7
sv_multipliers = multipliers[has_positive_multiplier]
support_vectors = X[has_positive_multiplier]
support_vectors_y = y[has_positive_multiplier]

w = np.sum(sv_multipliers[i] * support_vectors_y[i] * support_vectors[i] for i in range(
b = np.sum([support_vectors_y[i] - np.dot(w, support_vectors[i]) for i in range(len(supp
```

```
     pcost       dcost       gap    pres   dres
 0: -1.2293e+01 -2.8391e+01  1e+02  1e+01  2e+00
 1: -2.5419e+01 -3.4794e+01  3e+01  3e+00  5e-01
 2: -3.6313e+01 -4.5893e+01  3e+01  2e+00  4e-01
 3: -4.3790e+01 -4.5825e+01  8e+00  4e-01  7e-02
 4: -4.3706e+01 -4.3902e+01  5e-01  2e-02  4e-03
```

```
 5: -4.3700e+01 -4.3727e+01  3e-02  5e-05  8e-06
 6: -4.3721e+01 -4.3723e+01  1e-03  2e-06  4e-07
 7: -4.3723e+01 -4.3723e+01  1e-05  2e-08  4e-09
Optimal solution found.
```

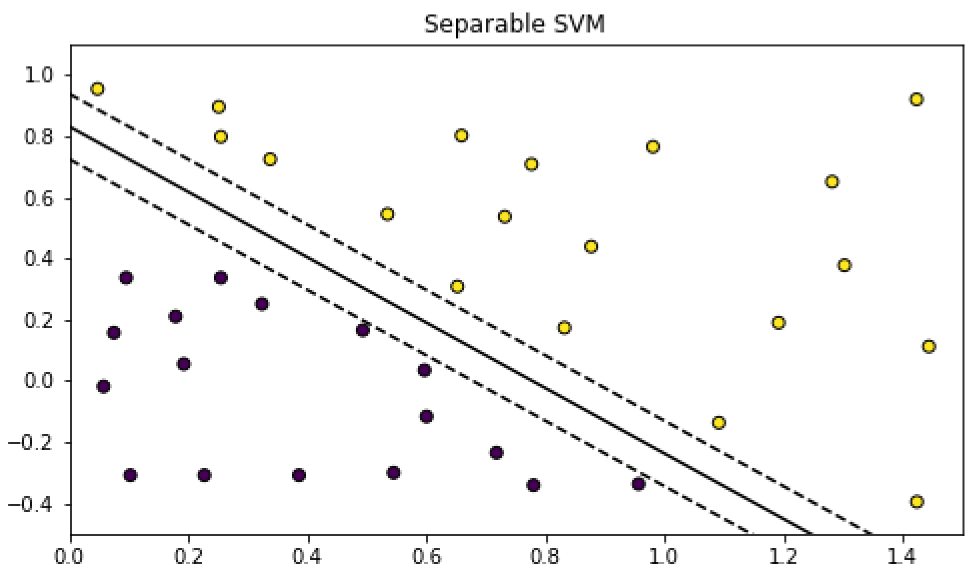
```
C:\Users\gandi\AppData\Local\Temp\ipykernel_40204\2315617634.py:30: DeprecationWarning:
Calling np.sum(generator) is deprecated, and in the future will give a different result.
Use np.sum(np.fromiter(generator)) or the python sum builtin instead.
  w = np.sum(sv_multipliers[i] * support_vectors_y[i] * support_vectors[i] for i in rang
e(len(support_vectors_y)))
```

In [4]:

```
# plot the data
margin = 1/np.linalg.norm(w)
ax1 = plt.figure(figsize=(8, 4.5))
plt.scatter(X[:, 0], X[:, 1], c=y, zorder=10, edgecolors='k')
x_hyperplane = np.linspace(0, 11)
y_hyperplane = -b / w[1]
upper_margin = y_hyperplane + margin
lower_margin = y_hyperplane - margin
plt.plot(x_hyperplane, x_hyperplane * -w[0] / w[1] + y_hyperplane, 'k-')
plt.plot(x_hyperplane, x_hyperplane * -w[0] / w[1] + upper_margin, 'k--')
plt.plot(x_hyperplane, x_hyperplane * -w[0] / w[1] + lower_margin, 'k--')

plt.xlim([0, 1.5])
plt.ylim([-0.5, 1.1])
plt.title('Separable SVM')
```

Out[4]:

```
Text(0.5, 1.0, 'Separable SVM')
```



b) and c) : Linearly Non-separable Dataset

In [5]:

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

m, n = x.shape
def nonlin_SVM(c, x, y, m):
    y = y.reshape(-1, 1)
    z = y * x
    p = matrix(np.dot(z, z.T))
    q = matrix(-1 * np.ones((m, 1)))
    A = matrix(y.reshape(1, -1))
    b = matrix(0.0)

    G = matrix(np.vstack((-1 * np.eye(m), np.eye(m))))
```

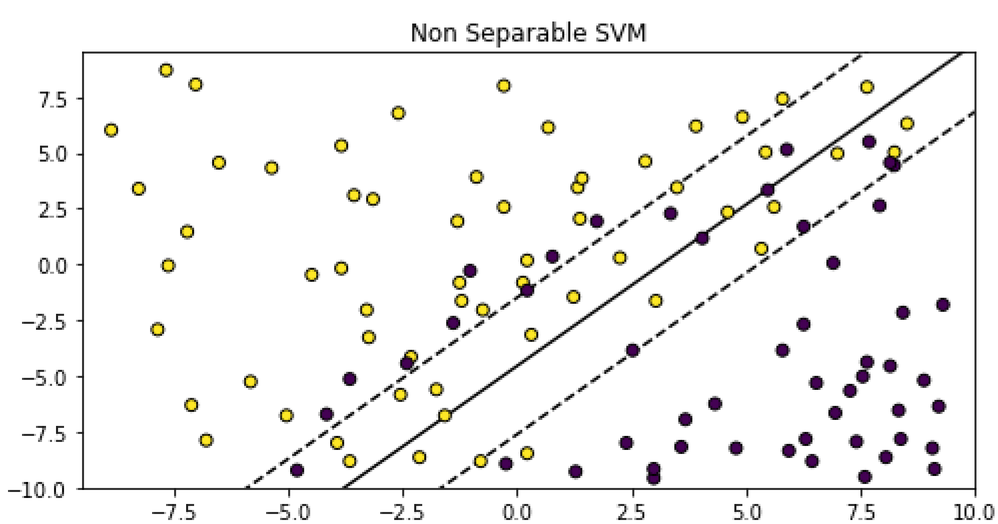
```python
    h = matrix(np.hstack((np.zeros(m), np.ones(m) * c)))
    #solvers.options['show_progress'] = False  # Turn off the summary
    solution = solvers.qp(p, q, G, h, A, b)
    multipliers = np.ravel(solution['x'])
    has_positive_multiplier = multipliers > 1e-7
    sv_multipliers = multipliers[has_positive_multiplier]
    support_vectors = x[has_positive_multiplier]
    support_vectors_y = y[has_positive_multiplier]
    w = np.sum(sv_multipliers[i] * support_vectors_y[i] * support_vectors[i] for i in ra
    b = np.sum([support_vectors_y[i] - np.dot(w, support_vectors[i]) for i in range(len(
    return w, b
w, b = nonlin_SVM(0.05, x, y, m)

margin = 1 / np.linalg.norm(w)
ax1 = plt.figure(figsize=(8, 4))
x_hyperplane = np.linspace(-10, 10)
y_hyperplane = -b / w[1]
upper_margin = y_hyperplane + margin
lower_margin = y_hyperplane - margin
plt.scatter(x[:, 0], x[:, 1], c=y, zorder=10, edgecolors='k')
plt.plot(x_hyperplane, x_hyperplane * -w[0] / w[1] + y_hyperplane, 'k-')
plt.plot(x_hyperplane, x_hyperplane * -w[0] / w[1] + upper_margin, 'k--')
plt.plot(x_hyperplane, x_hyperplane * -w[0] / w[1] + lower_margin, 'k--')

plt.xlim([-9.5, 10])
plt.ylim([-10, 9.5])
plt.title('Non Separable SVM')
```

```
     pcost       dcost       gap    pres   dres
 0: -3.0198e+01 -1.1038e+01  6e+02  3e+01  5e-14
 1: -3.0101e+00 -1.0669e+01  3e+01  8e-01  5e-14
 2: -2.1035e+00 -6.2126e+00  6e+00  1e-01  8e-15
 3: -2.0595e+00 -2.6797e+00  7e-01  1e-02  5e-15
 4: -2.1895e+00 -2.4041e+00  2e-01  3e-03  4e-15
 5: -2.2354e+00 -2.3269e+00  1e-01  1e-03  4e-15
 6: -2.2630e+00 -2.2889e+00  3e-02  2e-04  4e-15
 7: -2.2726e+00 -2.2755e+00  3e-03  1e-16  4e-15
 8: -2.2739e+00 -2.2740e+00  7e-05  2e-16  4e-15
 9: -2.2739e+00 -2.2739e+00  7e-07  1e-16  4e-15
Optimal solution found.
```

```
C:\Users\gandi\AppData\Local\Temp\ipykernel_40204\3918038852.py:24: DeprecationWarning:
Calling np.sum(generator) is deprecated, and in the future will give a different result.
Use np.sum(np.fromiter(generator)) or the python sum builtin instead.
  w = np.sum(sv_multipliers[i] * support_vectors_y[i] * support_vectors[i] for i in rang
e(len(support_vectors_y)))
```

Out[5]:

```
Text(0.5, 1.0, 'Non Separable SVM')
```



In [6]:

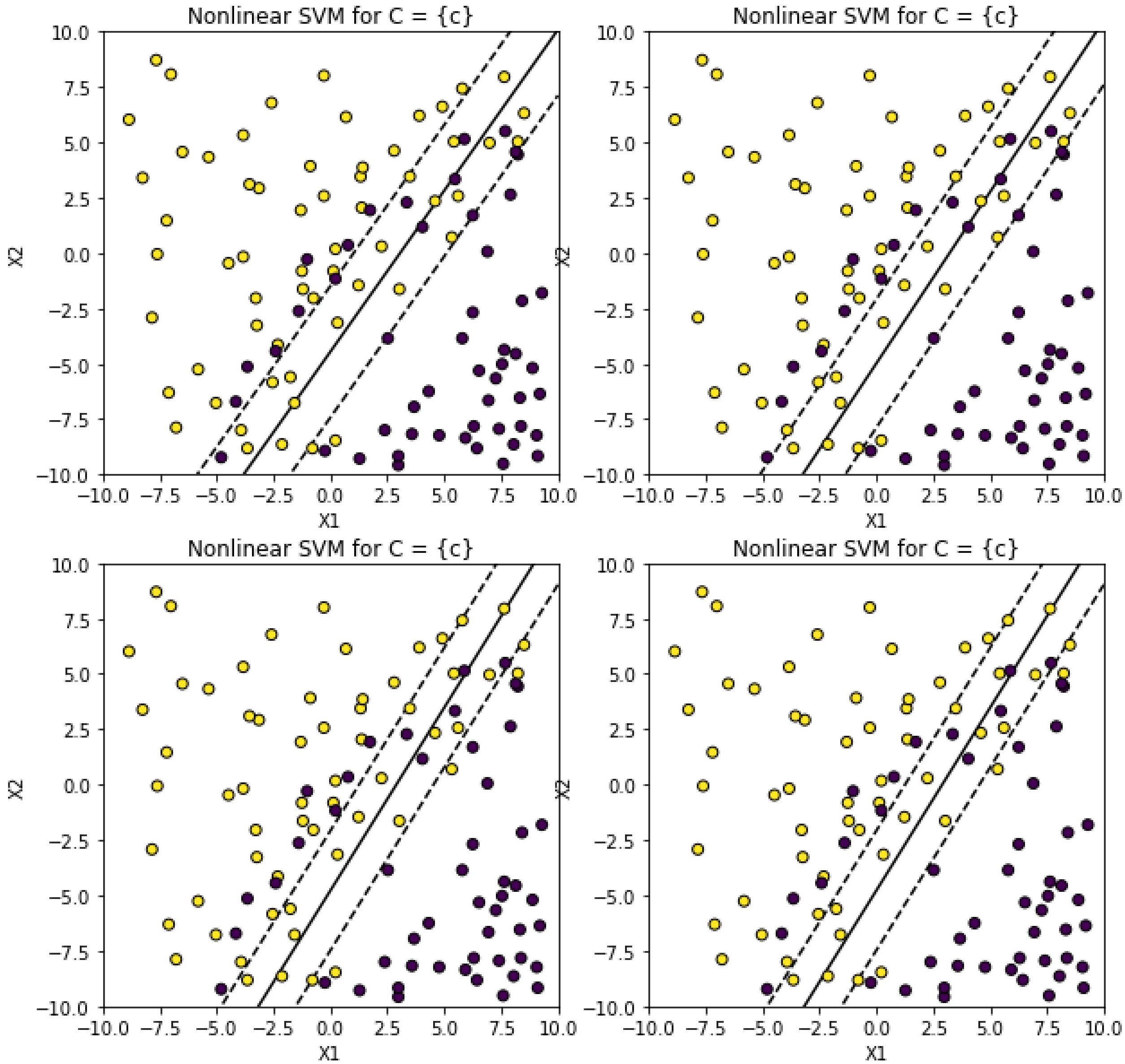
```python
#Part C
```

```
def compare_c(x, y, w, b, p, f, ax, c):
    y_hyperplane = -b / w[1]
    x_hyperplane = np.linspace(-10, 10)
    ax[p[0], p[1]].plot(x_hyperplane, x_hyperplane * -w[0] / w[1]  + y_hyperplane, 'k')
    margin = 1 / np.linalg.norm(w)
    upper_margin = y_hyperplane + margin
    lower_margin = y_hyperplane - margin
    ax[p[0], p[1]].scatter(x[:, 0], x[:, 1], c=y, zorder=10, edgecolors='k')
    ax[p[0], p[1]].plot(x_hyperplane, x_hyperplane * -w[0] / w[1] + upper_margin, 'k--')
    ax[p[0], p[1]].plot(x_hyperplane, x_hyperplane * -w[0] / w[1] + lower_margin, 'k--')

    # Lastly label each subplot:
    ax[p[0], p[1]].set_xlim([-10, 10])
    ax[p[0], p[1]].set_ylim([-10, 10])
    ax[p[0], p[1]].set_title('Nonlinear SVM for C = {c}')
    ax[p[0], p[1]].set_xlabel('X1')
    ax[p[0], p[1]].set_ylabel('X2')

data = np.loadtxt('dirty_nonlin.txt', delimiter='\t')
x = data[:, 0:2]
y = data[:, 2]
m, n = x.shape

figure, ax = plt.subplots(2, 2, figsize=(10, 10))
w, b = nonlin_SVM(0.1, x, y, m)
compare_c(x, y, w, b, (0,0), figure, ax, 0.1)
w, b = nonlin_SVM(1, x, y, m)
compare_c(x, y, w, b, (0,1), figure, ax, 1)
w, b = nonlin_SVM(100, x, y, m)
compare_c(x, y, w, b, (1,0), figure, ax, 100)
w, b = nonlin_SVM(1000000, x, y, m)
compare_c(x, y, w, b, (1,1), figure, ax, 1000000)
```

```
     pcost       dcost       gap    pres   dres
 0: -3.1636e+01 -2.2062e+01  7e+02  2e+01  6e-14
 1: -5.6226e+00 -2.0649e+01  5e+01  1e+00  6e-14
 2: -4.1036e+00 -1.1247e+01  1e+01  2e-01  1e-14
 3: -4.1158e+00 -5.1806e+00  1e+00  2e-02  6e-15
 4: -4.3486e+00 -4.7028e+00  4e-01  5e-03  7e-15
 5: -4.4340e+00 -4.5760e+00  2e-01  1e-03  8e-15
 6: -4.4708e+00 -4.5230e+00  5e-02  3e-04  7e-15
 7: -4.4851e+00 -4.5034e+00  2e-02  8e-05  6e-15
 8: -4.4920e+00 -4.4945e+00  3e-03  6e-06  8e-15
 9: -4.4931e+00 -4.4932e+00  3e-05  8e-08  7e-15
10: -4.4931e+00 -4.4931e+00  3e-07  8e-10  9e-15
Optimal solution found.
     pcost       dcost       gap    pres   dres
 0: -5.7505e+01 -2.5873e+02  1e+03  3e+00  1e-13
 1: -4.0125e+01 -1.6918e+02  2e+02  3e-01  9e-14
 2: -3.7704e+01 -5.8598e+01  2e+01  2e-02  6e-14
 3: -4.1093e+01 -5.0709e+01  1e+01  8e-03  6e-14
 4: -4.2959e+01 -4.6523e+01  4e+00  2e-03  5e-14
 5: -4.3779e+01 -4.5280e+01  2e+00  7e-04  7e-14
 6: -4.4015e+01 -4.4900e+01  9e-01  2e-04  6e-14
 7: -4.4311e+01 -4.4534e+01  2e-01  5e-05  7e-14
 8: -4.4341e+01 -4.4483e+01  1e-01  3e-05  8e-14
 9: -4.4388e+01 -4.4429e+01  4e-02  3e-15  1e-13
10: -4.4407e+01 -4.4408e+01  8e-04  3e-16  9e-14
11: -4.4407e+01 -4.4407e+01  8e-06  2e-16  9e-14
Optimal solution found.
     pcost       dcost       gap    pres   dres
 0: -2.8226e+03 -7.5705e+05  2e+06  6e-01  6e-12
 1: -2.4424e+03 -1.7816e+05  2e+05  1e-02  5e-12
 2: -2.6870e+03 -8.8257e+03  6e+03  3e-04  4e-12
 3: -3.5792e+03 -5.5403e+03  2e+03  8e-05  5e-12
 4: -3.8293e+03 -5.3116e+03  1e+03  5e-05  5e-12
```

```
 5: -4.0971e+03 -4.8435e+03  7e+02  2e-05  6e-12
 6: -4.2526e+03 -4.6780e+03  4e+02  1e-05  7e-12
 7: -4.3960e+03 -4.4924e+03  1e+02  4e-07  7e-12
 8: -4.4074e+03 -4.4687e+03  6e+01  1e-07  7e-12
 9: -4.4342e+03 -4.4356e+03  1e+00  2e-09  8e-12
10: -4.4348e+03 -4.4348e+03  6e-02  1e-10  8e-12
11: -4.4348e+03 -4.4348e+03  1e-03  2e-12  9e-12
Optimal solution found.
     pcost       dcost       gap    pres   dres
 0:  8.1127e+09 -7.3857e+13  2e+14  6e-01  6e-08
 1:  1.8686e+10 -1.6606e+13  2e+13  9e-03  5e-05
 2: -1.4230e+07 -2.8241e+11  3e+11  1e-04  6e-07
 3: -2.6039e+07 -2.9005e+09  3e+09  1e-06  4e-08
 4: -2.6381e+07 -1.0339e+08  8e+07  3e-08  4e-08
 5: -3.4041e+07 -5.7464e+07  2e+07  8e-09  6e-08
 6: -3.7746e+07 -5.3491e+07  2e+07  5e-09  5e-08
 7: -4.0548e+07 -4.9490e+07  9e+06  4e-09  7e-08
 8: -4.2701e+07 -4.6567e+07  4e+06  4e-09  6e-08
 9: -4.3915e+07 -4.5004e+07  1e+06  1e-08  8e-08
10: -4.4016e+07 -4.4780e+07  8e+05  3e-09  8e-08
11: -4.4290e+07 -4.4418e+07  1e+05  1e-09  1e-07
12: -4.4346e+07 -4.4349e+07  3e+03  2e-09  9e-08
13: -4.4347e+07 -4.4347e+07  2e+02  4e-09  8e-08
14: -4.4347e+07 -4.4347e+07  2e+00  4e-09  9e-08
Optimal solution found.
```

```
C:\Users\gandi\AppData\Local\Temp\ipykernel_40204\3918038852.py:24: DeprecationWarning:
Calling np.sum(generator) is deprecated, and in the future will give a different result.
Use np.sum(np.fromiter(generator)) or the python sum builtin instead.
  w = np.sum(sv_multipliers[i] * support_vectors_y[i] * support_vectors[i] for i in rang
e(len(support_vectors_y)))
C:\Users\gandi\AppData\Local\Temp\ipykernel_40204\3918038852.py:24: DeprecationWarning:
Calling np.sum(generator) is deprecated, and in the future will give a different result.
Use np.sum(np.fromiter(generator)) or the python sum builtin instead.
  w = np.sum(sv_multipliers[i] * support_vectors_y[i] * support_vectors[i] for i in rang
e(len(support_vectors_y)))
C:\Users\gandi\AppData\Local\Temp\ipykernel_40204\3918038852.py:24: DeprecationWarning:
Calling np.sum(generator) is deprecated, and in the future will give a different result.
Use np.sum(np.fromiter(generator)) or the python sum builtin instead.
  w = np.sum(sv_multipliers[i] * support_vectors_y[i] * support_vectors[i] for i in rang
e(len(support_vectors_y)))
C:\Users\gandi\AppData\Local\Temp\ipykernel_40204\3918038852.py:24: DeprecationWarning:
Calling np.sum(generator) is deprecated, and in the future will give a different result.
Use np.sum(np.fromiter(generator)) or the python sum builtin instead.
  w = np.sum(sv_multipliers[i] * support_vectors_y[i] * support_vectors[i] for i in rang
e(len(support_vectors_y)))
```



Explain your observations here:

As c increases, we can see a very minimal effect on the margin and decision boundaries; the boundary line moves towards the right slightly with the increase. Another key thing we see is that the slope of the line changes as c increases. But the issue with this is that there is no direct correlation because we see that the intercept decreases as c goes from 0.1 to 1, but it also increases as c goes from 100 to 1000000.

Assignment 5, Question 2: Physics Informed Neural Networks

Importing the necessary libraries

In [1]:
```
import sys
import torch
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 math
import matplotlib.pyplot as plt
import torch.nn as nn

np.random.seed(1234)
```

In [2]:
```
# Uncomment this to install the PyDOE package.
!pip install pyDOE
```

```
Requirement already satisfied: pyDOE in c:\users\gandi\anaconda3\lib\site-packages (0.3.
8)
Requirement already satisfied: numpy in c:\users\gandi\anaconda3\lib\site-packages (from
pyDOE) (1.21.5)
Requirement already satisfied: scipy in c:\users\gandi\anaconda3\lib\site-packages (from
pyDOE) (1.7.3)
```

In [3]:
```
# Enable use of the GPU
device = torch.device('cpu')
```

Physics-informed Neural Networks

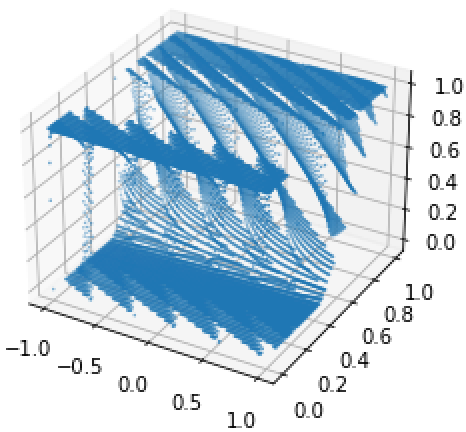
We will use a Fully Connected Neural Network to solve this PDE. The network will take in two features as input, x, the spatial co-ordinate, and t, the time co-ordinate. From these two inputs, the network should output the solution to the PDE at that point in space and time. For instance, the solution 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 [4]:
```
#Part A
dataset = np.load('q2_data.npy')

x = np.linspace(-1,1,256)
t = np.linspace(0,1,100)
```

```
u = dataset
X, T = np.meshgrid(x,t)

ax = plt.axes(projection ='3d')
ax.scatter(torch.from_numpy(X),torch.from_numpy(T),torch.from_numpy(u), s = .3)
```

Out[4]: <mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x1a70a4898e0>



In [5]:
```
#Part B
class FCNN(torch.nn.Module):

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

        # Define layers and network components here
        layers = np.array([2, 20, 20, 20, 20, 20, 20, 20, 20, 1])
        self.activation = nn.Tanh()
        self.loss_function = nn.MSELoss(reduction ='mean')
        self.linears = nn.ModuleList([nn.Linear(layers[i], layers[i + 1]) for i in range
        self.iter = 0
        for i in range(len(layers) - 1):
            nn.init.xavier_normal_(self.linears[i].weight.data, gain = 1)
            nn.init.zeros_(self.linears[i].bias.data)
    def forward(self, x):
        layers = np.array([2, 20, 20, 20, 20, 20, 20, 20, 20, 1])
        if torch.is_tensor(x) != True:
            x = torch.from_numpy(x)
        u_b_val = ub_torch
        l_b_val = lb_torch
        x = (x - l_b_val)/(u_b_val - l_b_val)
        a = x.float()
        for i in range(len(layers) - 2):
            z = self.linears[i](a)
            a = self.activation(z)
        a = self.linears[-1](a)
        return a
```

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:

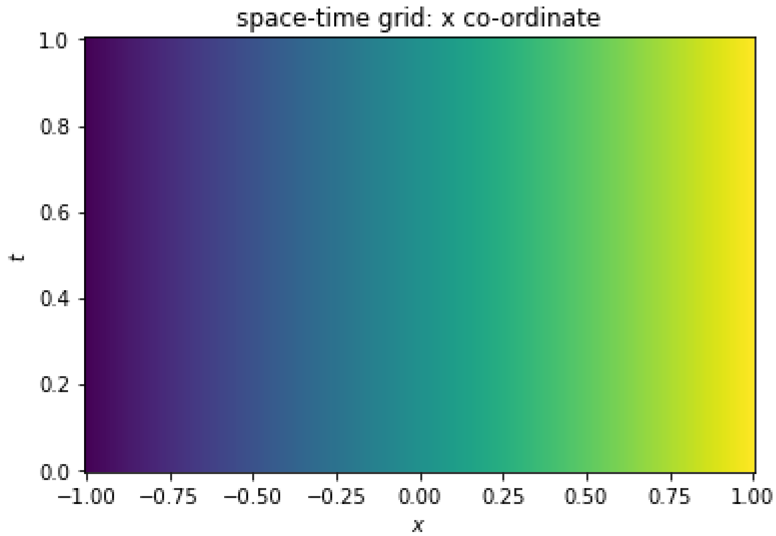
1. Define a grid spanning all possible x, t combinations. If we have N possible values of x and M possible values of t, we'll have $N_{samples} = N \times M$ combinations of x and t.
2. Reshape this grid into a feature matrix, of shape ($N_{samples}$, $N_{features}$). We'll use $N_{samples}$ from (1.), 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.

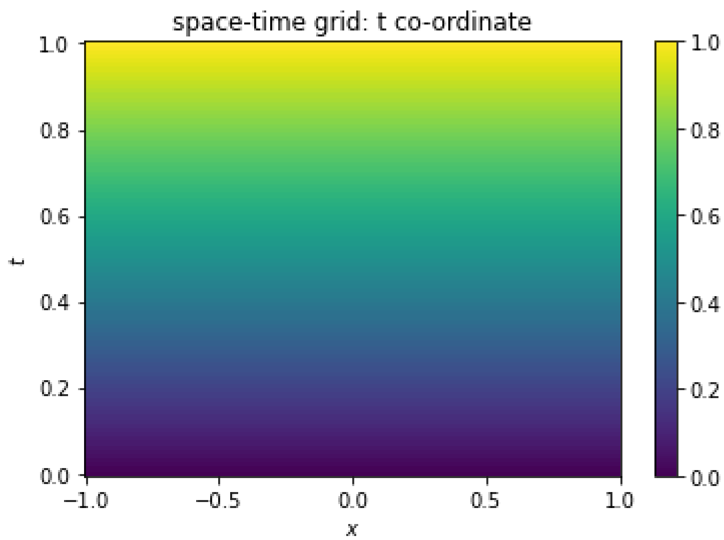
We'll use a direct MSE loss to make sure that the predictions of the network at the boundary and initial conditions are the correct values, and a loss based on the PDE residual for points in (x, t) space that do not lie on the boundary or initial conditions.

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

In [6]:
```
# Nu defines the value for viscosity
nu = 1

# 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

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

# 256 elements in the space dimension (Note, shape should be (256,1). You may have to re
x_vector = np.linspace(-1, 1, 256)

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

In [7]:
```
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()
```



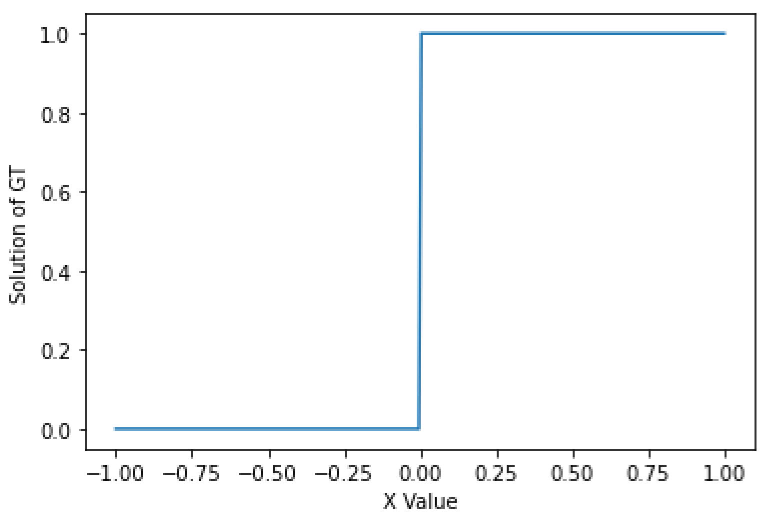


In [8]:
```
# TO-DO: Next, reshape the xx and tt arrays to be of shape (25600, 1), and stack them to
xt_combined_flat = np.hstack((xx.flatten()[:, None], tt.flatten()[:, None]))

# Reshape the gt_solution data array to be of shape (25600, 1) in an array called u_flat
u_flat = gt_solution.flatten('F')[:, None] # u_flat is the ground truth data, defined fo

# Domain bounds
# lb should be the minimum x and t values, stored as a numpy array of shape (1,2). The s
lb_numpy = xt_combined_flat[0]
# ub should be the maximum x and t values, stored as a numpy array of shape (1,2). The s
ub_numpy = xt_combined_flat[-1]


# Define the initial conditions: t = 0.
# initial_conditions_xt is a 256 x 2 vector containing the x and t values at the initial
# 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]


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

```
# right_boundary_u is a 100 x 1 vector storing u at the right boundary.
right_boundary_xt = np.hstack((xx[:, -1][:, None], tt[:, 0][:, None]))
right_boundary_u = gt_solution[:, 0][:, None]

# Stack the initial condition, left boundary condition, and right boundary condition tog
edge_samples_xt = np.vstack([initial_condition_xt, left_boundary_xt, right_boundary_xt])
# 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 training set
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_boundary_u])

# Randomly sample a training set for the MSE loss from the data at the initial and bound
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]
```

In [9]:
```
# Plot the initial condition as a function of x.
plt.plot(x_vector,initial_condition_u )
plt.xlabel('X Value')
plt.ylabel('Solution of GT')
```

Out[9]:
```
Text(0, 0.5, 'Solution of GT')
```



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 [10]:
```
# TO-DO: Convert lower and upper bounds to torch tensors. No gradient is required for th
lb_torch = torch.tensor(lb_numpy).float().to(device)
ub_torch = torch.tensor(ub_numpy).float().to(device)
print(lb_torch.shape)

# Convert x and t data to torch Tensors. Specify that the gradient is required while con
# train_samples_xt and all_samples_xt are 2-column matrices with x and t as columns. Her
# be x, and t_u and t_f to just be t.
x_boundary_train = torch.tensor(train_samples_xt[:, 0], requires_grad = True).float().to
t_boundary_train =torch.tensor(train_samples_xt[:, 1], requires_grad = True).float().to(
x_sampled_train = torch.tensor(all_samples_xt[:, 0], requires_grad = True).float().to(de
t_sampled_train = torch.tensor(all_samples_xt[:, 1], requires_grad = True).float().to(de
```

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

```
torch.Size([2])
```

Training

In [11]:

```
# b): Initialize the Deep Neural Network, and put it on the GPU.
fcnn = FCNN().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 returns the predic
def net_u(fcnn, xt):
    # Call the fcnn network with the x and t co-ordinates, return the prediction
    return fcnn.forward(xt)
    # return network output

## The function net_f, takes in the net_u function, x, and t, and computes the residual
def net_f(x, t, nu):
    # Calculate the residual of the PDE, using the gradients computed via autograd, and
    g = t.clone()
    nu = nu
    # Step 1: Using the function net_u, calculate the predicted u variable.
    u = net_u(x, g)

    # Step 2: Compute the gradients used in the Burgers' equation PDE using torch.autogr
    u_x_t = torch.autograd.grad(outputs = u, inputs = g, grad_outputs = torch.ones([t.sh
    u_xx_tt = torch.autograd.grad(u_x_t, g, torch.ones(t.shape).to(device), create_graph
    f = u_x_t[:, [1]] + (fcnn.forward(g)) * (u_x_t[:, [0]]) - nu * u_xx_tt[:, [0]]
    f_hat = torch.zeros(t.shape[0], 1).to(device)
    loss_function = nn.MSELoss(reduction ='mean')
    # Step 3: Calculate the residual.
    return loss_function(f, f_hat)
    # return PDE residual

iteration = 0
## The loss function will calculate the loss as a combination of the MSE loss on the bou
def loss_func():
    nu = 0.01
    optimizer.zero_grad()
    # Predict the solution along the intitial and boundary conditions
    x = torch.hstack((x_boundary_train[:, None], t_boundary_train[:, None]))
    u_pred = net_u(fcnn, x)
    # Predict the solution at the sampled co-location points
    xt = torch.hstack((x_sampled_train[:, None], t_sampled_train[:, None]))
    f_pred = net_u(fcnn, xt)
    # Compute MSE loss on (x,t) points that lie on initial and boundary conditions, PDE
    mse_loss = nn.MSELoss(reduction ='mean')(u_pred, u_boundary_train)
    pde_loss = net_f(fcnn, xt, nu)
    total_loss = mse_loss + pde_loss
```

```python
    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, total_loss.ite
      )
    return total_loss

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


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

```
Iter 100, Loss: 4.20611e-02, Loss_u: 3.28939e-02, Loss_f: 9.16723e-03
Iter 200, Loss: 1.05530e-03, Loss_u: 5.10986e-04, Loss_f: 5.44310e-04
Iter 300, Loss: 1.90347e-04, Loss_u: 9.01178e-05, Loss_f: 1.00229e-04
Iter 400, Loss: 1.04771e-04, Loss_u: 5.47703e-05, Loss_f: 5.00004e-05
Iter 500, Loss: 8.15868e-05, Loss_u: 5.01841e-05, Loss_f: 3.14028e-05
Iter 600, Loss: 6.70666e-05, Loss_u: 4.47317e-05, Loss_f: 2.23349e-05
Iter 700, Loss: 5.69397e-05, Loss_u: 4.16725e-05, Loss_f: 1.52671e-05
Iter 800, Loss: 5.05721e-05, Loss_u: 3.81502e-05, Loss_f: 1.24220e-05
Iter 900, Loss: 4.61222e-05, Loss_u: 3.51379e-05, Loss_f: 1.09843e-05
Iter 1000, Loss: 4.18428e-05, Loss_u: 3.14472e-05, Loss_f: 1.03957e-05
Iter 1100, Loss: 3.76847e-05, Loss_u: 2.83341e-05, Loss_f: 9.35067e-06
Iter 1200, Loss: 3.46325e-05, Loss_u: 2.55635e-05, Loss_f: 9.06904e-06
Iter 1300, Loss: 3.02563e-05, Loss_u: 2.19635e-05, Loss_f: 8.29278e-06
Iter 1400, Loss: 2.61606e-05, Loss_u: 1.74654e-05, Loss_f: 8.69525e-06
Iter 1500, Loss: 2.14259e-05, Loss_u: 1.43542e-05, Loss_f: 7.07168e-06
Iter 1600, Loss: 1.81102e-05, Loss_u: 1.14037e-05, Loss_f: 6.70652e-06
Iter 1700, Loss: 1.36630e-05, Loss_u: 7.14550e-06, Loss_f: 6.51749e-06
Iter 1800, Loss: 9.83575e-06, Loss_u: 4.70146e-06, Loss_f: 5.13429e-06
Iter 1900, Loss: 7.76961e-06, Loss_u: 3.23145e-06, Loss_f: 4.53816e-06
Iter 2000, Loss: 6.72809e-06, Loss_u: 2.55331e-06, Loss_f: 4.17478e-06
Iter 2100, Loss: 6.00145e-06, Loss_u: 2.19866e-06, Loss_f: 3.80278e-06
Iter 2200, Loss: 5.29277e-06, Loss_u: 1.98714e-06, Loss_f: 3.30563e-06
Iter 2300, Loss: 4.59860e-06, Loss_u: 1.74745e-06, Loss_f: 2.85116e-06
Iter 2400, Loss: 4.32907e-06, Loss_u: 1.67832e-06, Loss_f: 2.65074e-06
Iter 2500, Loss: 4.08246e-06, Loss_u: 1.63050e-06, Loss_f: 2.45196e-06
```

Out[11]:
```
tensor(3.9678e-06, grad_fn=<AddBackward0>)
```

In [12]:
```python
#Part d
# d) Given a 2-D array of space and time variables, predict the corresponding PDE soluti
def predict(fcnn,xt,nu):
    # As before, separate xt into vectors and convert these vectors of time and space in
    x_torch = torch.tensor(xt[:, 0],requires_grad = True).float().to(device)   # assign x
    t_torch = torch.tensor(xt[:, 1],requires_grad = True).float().to(device) # assign t
    fcnn.eval()
    # TO-DO: get predicted u and PDE residual from networks.
    u_pred = net_u(fcnn, torch.hstack((x_torch[:, None],t_torch[:, None])))
    f_pred = net_f(fcnn, torch.hstack((x_torch[:, None],t_torch[:, None])), nu)
    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_{L1} = \sum{x,t} \left | u(x,t){gt} - u(x,t){pred} \right |

```
In [13]: # Predict the PDE solution for every combination of x and t
u_pred, f_pred = predict(fcnn, xt_combined_flat, 0.01)
# Compute the normalized L2 error of the solution
error_el2 = torch.linalg.norm((torch.tensor(u_flat).float().to(device) - u_pred), 2) / t
print('Normalized L2 Error: %e' % (error_el2))
# Compute the L1 error of the solution.
error_el1 = torch.sum(torch.abs(torch.tensor(u_flat).float().to(device) - u_pred))
print('L1 Error: %e' % (error_el1))
```

```
Normalized L2 Error: 9.356611e-01
L1 Error: 1.228037e+04
```

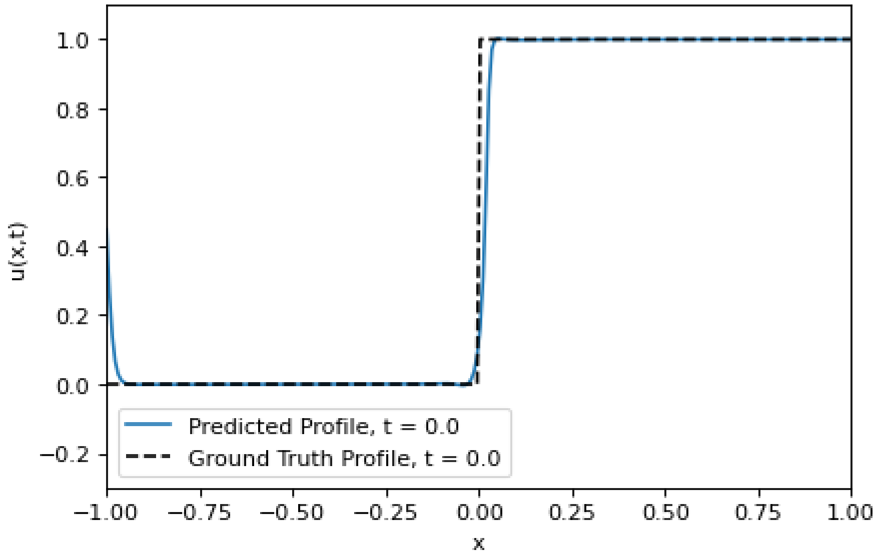
Plotting Results

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.

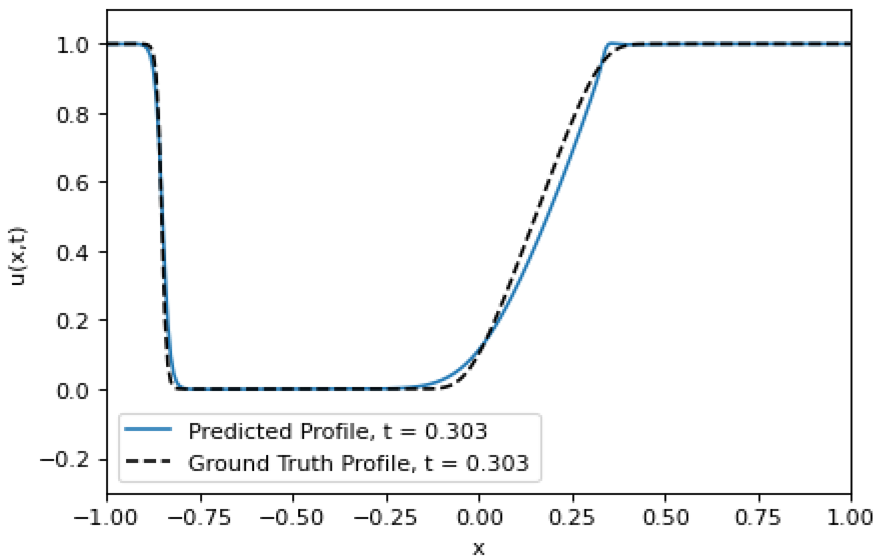
Figure 1: Plotting a comparison of the ground truth solution to the predicted solution at various times.

```
In [14]: # Interpolate array for plotting.
u_pred_grid = griddata(xt_combined_flat, u_pred.flatten(), (xx, tt), method='cubic')
Error = np.abs(gt_solution - u_pred_grid)

for i in range(0, 100, 30):
    plt.figure(dpi = 80)
    print(t_vector[i])
    plt.plot(x_vector,u_pred_grid[i], label = 'Predicted Profile, t = {:.03}'.format(t_v
    plt.plot(x_vector,gt_solution[i], 'k--', label = 'Ground Truth Profile, t = {:.03}'.
    plt.legend()
    plt.ylim([-0.3,1.1])
    plt.xlim([-1,1])
    plt.xlabel('x')
    plt.ylabel('u(x,t)')
    plt.show()
```

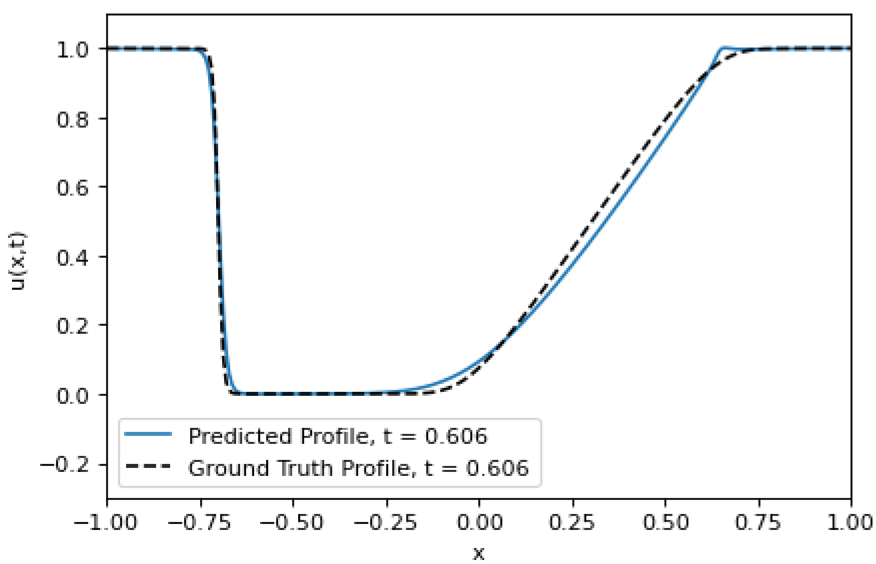
```
0.0
```




```
0.30303030303030304
```




```
0.6060606060606061
```




```
0.9090909090909092
```

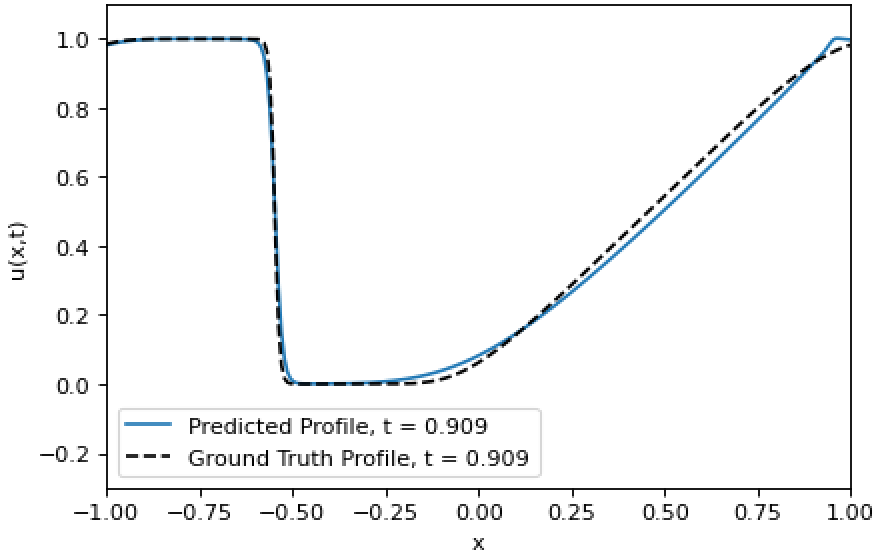
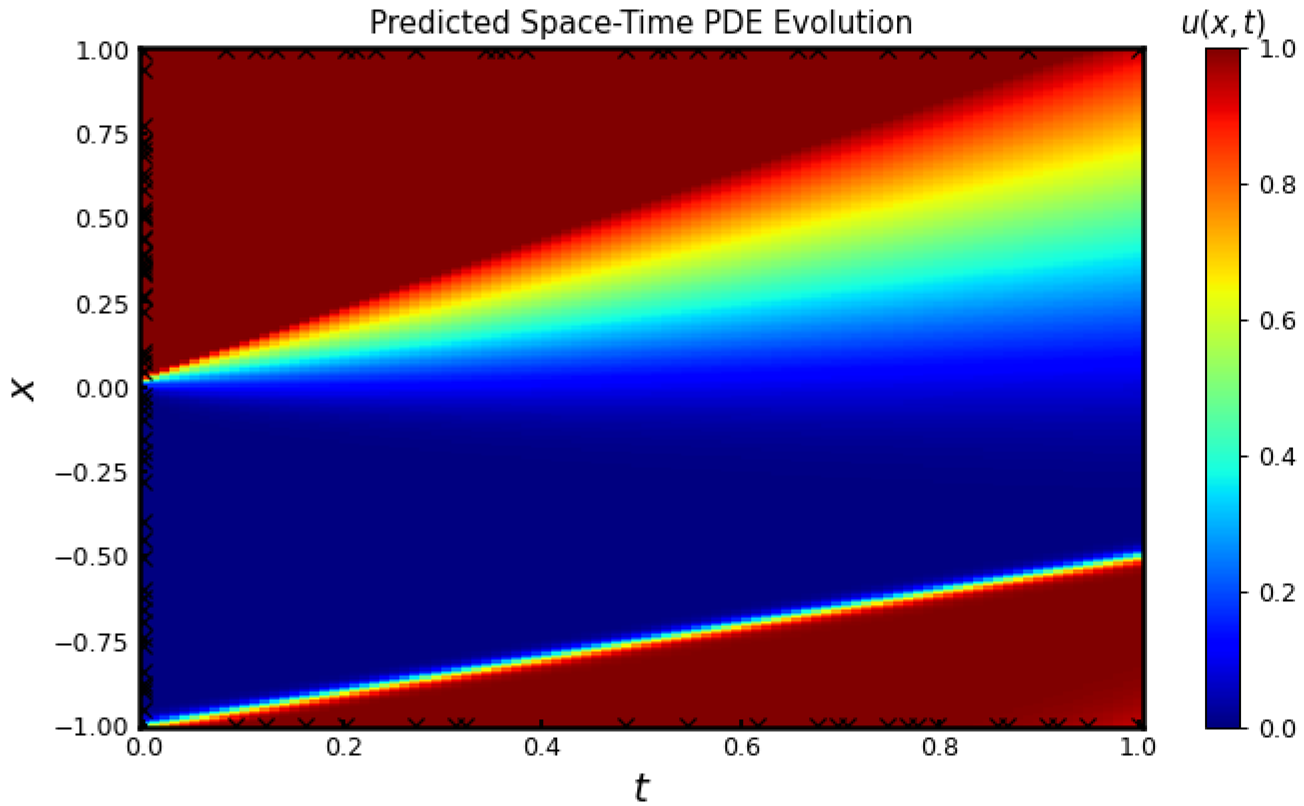
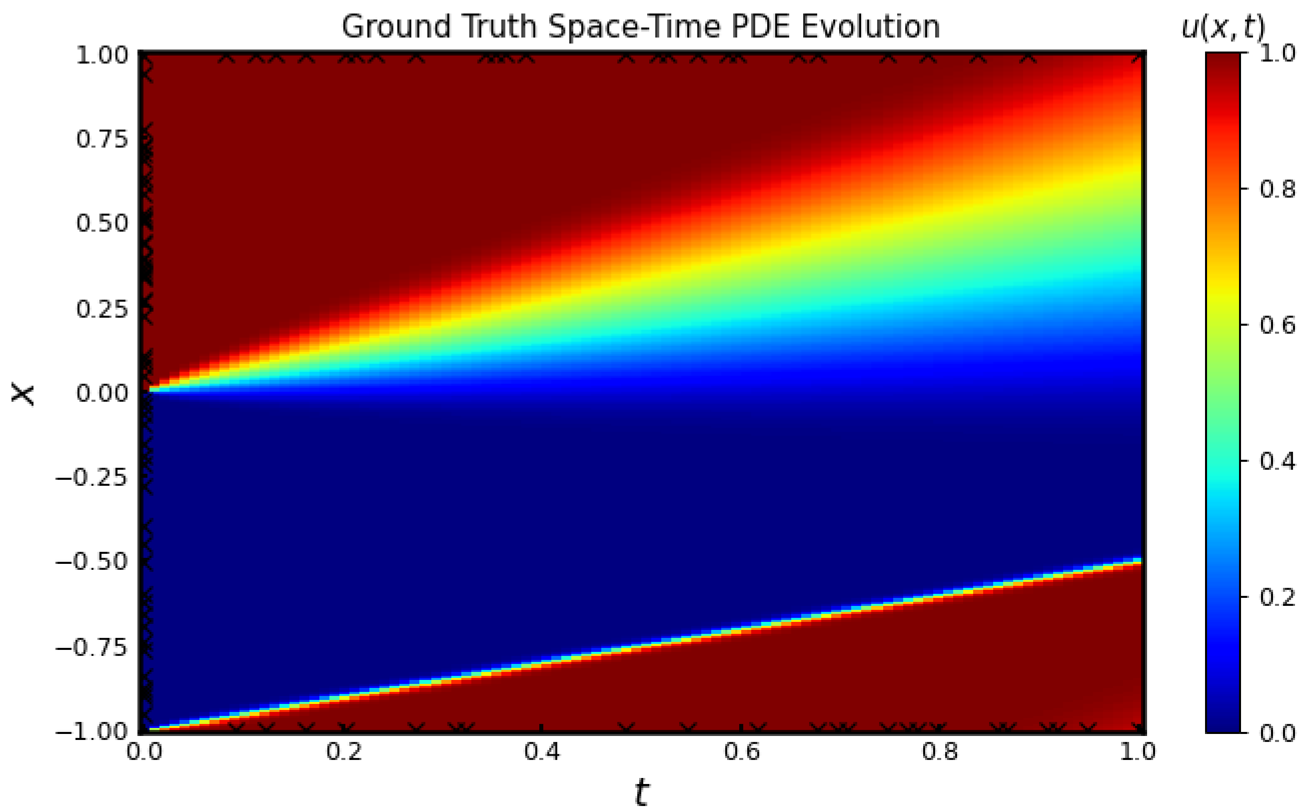


Figure 2: Plotting the time evolution of the 1-D Burgers equation as a 2-D image

In [15]:
```
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()
```





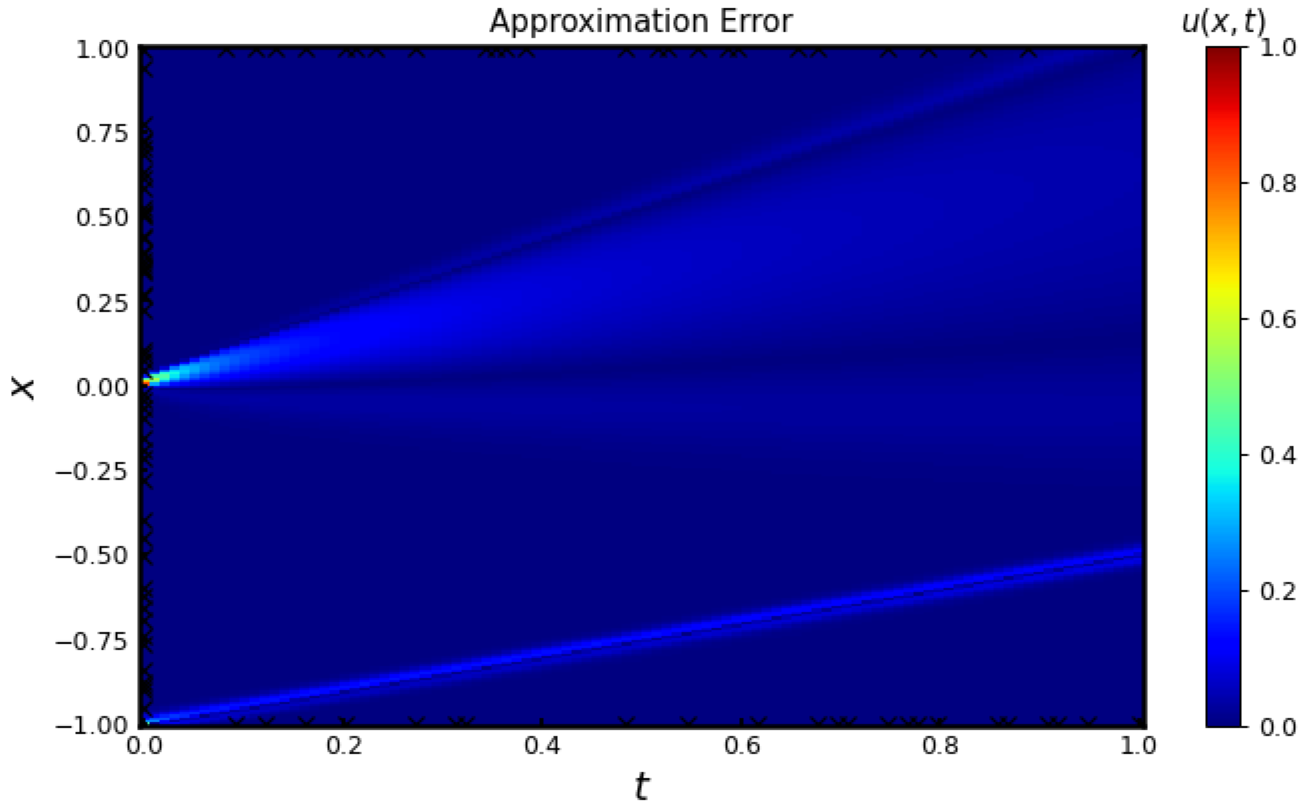


Figure 3: Plotting the solution at t = 1 s for $\nu = 1.0, 0.1, \frac{0.01}{\pi}$ on the same figure.

In [16]:

```
nu = 1
fcnn_1 = FCNN().to(device)
optimizer_1 = torch.optim.LBFGS(
    fcnn_1.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"
)
iteration = 0
def loss_func_1():
    nu = 1
    optimizer_1.zero_grad()
    # Predict the solution along the intitial and boundary conditions
    xtb = torch.hstack((x_boundary_train[:, None], t_boundary_train[:, None]))
    u_pred = net_u(fcnn_1,xtb)
    # Predict the solution at the sampled co-location points
    xtf = torch.hstack((x_sampled_train[:, None], t_sampled_train[:, None]))
    f_pred = net_u(fcnn_1, xtf)
    # Compute MSE loss on (x,t) points that lie on initial and boundary conditions, PDE
    mse_loss = nn.MSELoss(reduction ='mean')(u_pred, u_boundary_train)
    pde_loss = net_f(fcnn_1,xtf,nu)
    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, total_loss.ite
        )
    return total_loss

# This initializes the gradients for training
fcnn_1.train()

# This carries out the entire optimization process with L-BFGS, by calling the loss_func
optimizer_1.step(loss_func_1)
loss_func_1()

u_pred_1, f_pred_1 = predict(fcnn_1, xt_combined_flat, nu)
u_pred_grid_1 = griddata(xt_combined_flat, u_pred_1.flatten(), (xx, tt), method='cubic')
```

```
Iter 100, Loss: 4.46781e-02, Loss_u: 2.86316e-02, Loss_f: 1.60465e-02
Iter 200, Loss: 2.70485e-02, Loss_u: 1.74278e-02, Loss_f: 9.62077e-03
Iter 300, Loss: 1.63878e-02, Loss_u: 1.05637e-02, Loss_f: 5.82417e-03
Iter 400, Loss: 1.13688e-02, Loss_u: 6.51016e-03, Loss_f: 4.85860e-03
Iter 500, Loss: 8.47303e-03, Loss_u: 4.81776e-03, Loss_f: 3.65527e-03
Iter 600, Loss: 6.48051e-03, Loss_u: 3.58784e-03, Loss_f: 2.89267e-03
Iter 700, Loss: 5.48785e-03, Loss_u: 3.27055e-03, Loss_f: 2.21730e-03
Iter 800, Loss: 4.85192e-03, Loss_u: 2.77470e-03, Loss_f: 2.07721e-03
Iter 900, Loss: 4.07316e-03, Loss_u: 2.41823e-03, Loss_f: 1.65493e-03
Iter 1000, Loss: 3.62585e-03, Loss_u: 2.24286e-03, Loss_f: 1.38299e-03
Iter 1100, Loss: 3.23322e-03, Loss_u: 1.91912e-03, Loss_f: 1.31410e-03
Iter 1200, Loss: 2.77435e-03, Loss_u: 1.55388e-03, Loss_f: 1.22047e-03
Iter 1300, Loss: 2.43062e-03, Loss_u: 1.32485e-03, Loss_f: 1.10577e-03
Iter 1400, Loss: 2.14971e-03, Loss_u: 1.13589e-03, Loss_f: 1.01382e-03
Iter 1500, Loss: 1.89782e-03, Loss_u: 9.26974e-04, Loss_f: 9.70841e-04
Iter 1600, Loss: 1.75735e-03, Loss_u: 7.77095e-04, Loss_f: 9.80250e-04
Iter 1700, Loss: 1.44575e-03, Loss_u: 6.71548e-04, Loss_f: 7.74198e-04
Iter 1800, Loss: 1.25043e-03, Loss_u: 5.87337e-04, Loss_f: 6.63088e-04
Iter 1900, Loss: 1.13605e-03, Loss_u: 5.15630e-04, Loss_f: 6.20421e-04
Iter 2000, Loss: 1.08809e-03, Loss_u: 4.74810e-04, Loss_f: 6.13283e-04
Iter 2100, Loss: 1.01045e-03, Loss_u: 4.28636e-04, Loss_f: 5.81814e-04
Iter 2200, Loss: 9.59627e-04, Loss_u: 3.97270e-04, Loss_f: 5.62358e-04
Iter 2300, Loss: 9.09205e-04, Loss_u: 3.67933e-04, Loss_f: 5.41272e-04
Iter 2400, Loss: 8.57125e-04, Loss_u: 3.30492e-04, Loss_f: 5.26633e-04
Iter 2500, Loss: 8.04471e-04, Loss_u: 3.00957e-04, Loss_f: 5.03513e-04
Iter 2600, Loss: 7.63095e-04, Loss_u: 2.96608e-04, Loss_f: 4.66487e-04
Iter 2700, Loss: 7.23654e-04, Loss_u: 2.92488e-04, Loss_f: 4.31166e-04
Iter 2800, Loss: 6.94961e-04, Loss_u: 2.77384e-04, Loss_f: 4.17577e-04
Iter 2900, Loss: 6.64955e-04, Loss_u: 2.69368e-04, Loss_f: 3.95586e-04
Iter 3000, Loss: 6.42623e-04, Loss_u: 2.57212e-04, Loss_f: 3.85411e-04
Iter 3100, Loss: 6.02646e-04, Loss_u: 2.36658e-04, Loss_f: 3.65987e-04
Iter 3200, Loss: 5.77610e-04, Loss_u: 2.19825e-04, Loss_f: 3.57785e-04
Iter 3300, Loss: 5.57566e-04, Loss_u: 2.10861e-04, Loss_f: 3.46705e-04
Iter 3400, Loss: 5.33321e-04, Loss_u: 2.03168e-04, Loss_f: 3.30153e-04
Iter 3500, Loss: 5.17335e-04, Loss_u: 1.97840e-04, Loss_f: 3.19495e-04
Iter 3600, Loss: 5.04372e-04, Loss_u: 1.88898e-04, Loss_f: 3.15475e-04
Iter 3700, Loss: 4.94904e-04, Loss_u: 1.83469e-04, Loss_f: 3.11435e-04
```

In [18]:
```
nu = 0.1
fcnn_0_1 = FCNN().to(device)
optimizer_0_1 = torch.optim.LBFGS(
    fcnn_0_1.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"
)
iteration = 0
def loss_func_0_1():
```

```
    nu = 0.1
    optimizer_0_1.zero_grad()
    # Predict the solution along the intitial and boundary conditions
    xtb = torch.hstack((x_boundary_train[:,None],t_boundary_train[:,None]))
    u_pred = net_u(fcnn_0_1,xtb)
    # Predict the solution at the sampled co-location points
    xtf = torch.hstack((x_sampled_train[:,None],t_sampled_train[:,None]))
    f_pred = net_u(fcnn_0_1,xtf)
    # Compute MSE loss on (x,t) points that lie on initial and boundary conditions, PDE
    mse_loss = nn.MSELoss(reduction ='mean')(u_pred, u_boundary_train)
    pde_loss = net_f(fcnn_0_1,xtf,nu)
    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, total_loss.ite
      )
    return total_loss

# This initializes the gradients for training
fcnn_0_1.train()

# This carries out the entire optimization process with L-BFGS, by calling the loss_func
optimizer_0_1.step(loss_func_0_1)
loss_func_0_1()

u_pred_0_1, f_pred_0_1 = predict(fcnn_0_1, xt_combined_flat, nu)
u_pred_grid_0_1 = griddata(xt_combined_flat, u_pred_0_1.flatten(), (xx, tt), method='cub
```

```
Iter 100, Loss: 7.04036e-02, Loss_u: 4.88535e-02, Loss_f: 2.15501e-02
Iter 200, Loss: 1.89023e-02, Loss_u: 1.31535e-02, Loss_f: 5.74885e-03
Iter 300, Loss: 6.47775e-03, Loss_u: 4.01045e-03, Loss_f: 2.46729e-03
Iter 400, Loss: 3.66777e-03, Loss_u: 2.78417e-03, Loss_f: 8.83598e-04
Iter 500, Loss: 2.28554e-03, Loss_u: 1.55812e-03, Loss_f: 7.27412e-04
Iter 600, Loss: 1.55143e-03, Loss_u: 1.16520e-03, Loss_f: 3.86234e-04
Iter 700, Loss: 1.24816e-03, Loss_u: 8.86570e-04, Loss_f: 3.61590e-04
Iter 800, Loss: 9.05068e-04, Loss_u: 5.41726e-04, Loss_f: 3.63342e-04
Iter 900, Loss: 7.40414e-04, Loss_u: 3.83941e-04, Loss_f: 3.56473e-04
Iter 1000, Loss: 5.74051e-04, Loss_u: 2.96683e-04, Loss_f: 2.77368e-04
Iter 1100, Loss: 4.17330e-04, Loss_u: 1.84604e-04, Loss_f: 2.32726e-04
Iter 1200, Loss: 3.23564e-04, Loss_u: 1.20563e-04, Loss_f: 2.03001e-04
Iter 1300, Loss: 2.51644e-04, Loss_u: 9.88711e-05, Loss_f: 1.52773e-04
Iter 1400, Loss: 1.90126e-04, Loss_u: 8.09919e-05, Loss_f: 1.09134e-04
Iter 1500, Loss: 1.50781e-04, Loss_u: 7.29511e-05, Loss_f: 7.78298e-05
Iter 1600, Loss: 1.25717e-04, Loss_u: 6.43247e-05, Loss_f: 6.13926e-05
Iter 1700, Loss: 1.04110e-04, Loss_u: 4.51253e-05, Loss_f: 5.89843e-05
Iter 1800, Loss: 8.72761e-05, Loss_u: 3.47668e-05, Loss_f: 5.25093e-05
Iter 1900, Loss: 7.28682e-05, Loss_u: 3.10951e-05, Loss_f: 4.17731e-05
Iter 2000, Loss: 6.25473e-05, Loss_u: 2.89293e-05, Loss_f: 3.36180e-05
Iter 2100, Loss: 5.58105e-05, Loss_u: 2.65133e-05, Loss_f: 2.92972e-05
Iter 2200, Loss: 4.99182e-05, Loss_u: 2.52243e-05, Loss_f: 2.46939e-05
Iter 2300, Loss: 4.63910e-05, Loss_u: 2.36565e-05, Loss_f: 2.27344e-05
Iter 2400, Loss: 4.33007e-05, Loss_u: 2.21033e-05, Loss_f: 2.11974e-05
Iter 2500, Loss: 4.03268e-05, Loss_u: 2.10318e-05, Loss_f: 1.92950e-05
Iter 2600, Loss: 3.78039e-05, Loss_u: 2.00869e-05, Loss_f: 1.77170e-05
Iter 2700, Loss: 3.58539e-05, Loss_u: 1.92299e-05, Loss_f: 1.66240e-05
Iter 2800, Loss: 3.48879e-05, Loss_u: 1.89074e-05, Loss_f: 1.59805e-05
Iter 2900, Loss: 3.34817e-05, Loss_u: 1.81404e-05, Loss_f: 1.53413e-05
Iter 3000, Loss: 3.23483e-05, Loss_u: 1.81966e-05, Loss_f: 1.41517e-05
```

```
Iter 3100, Loss: 3.13128e-05, Loss_u: 1.79701e-05, Loss_f: 1.33427e-05
Iter 3200, Loss: 3.00095e-05, Loss_u: 1.75132e-05, Loss_f: 1.24963e-05
Iter 3300, Loss: 2.88110e-05, Loss_u: 1.73660e-05, Loss_f: 1.14449e-05
Iter 3400, Loss: 2.80187e-05, Loss_u: 1.71812e-05, Loss_f: 1.08375e-05
Iter 3500, Loss: 2.69682e-05, Loss_u: 1.67598e-05, Loss_f: 1.02084e-05
Iter 3600, Loss: 2.60476e-05, Loss_u: 1.63754e-05, Loss_f: 9.67220e-06
Iter 3700, Loss: 2.50480e-05, Loss_u: 1.61046e-05, Loss_f: 8.94340e-06
Iter 3800, Loss: 2.41962e-05, Loss_u: 1.56186e-05, Loss_f: 8.57753e-06
Iter 3900, Loss: 2.34388e-05, Loss_u: 1.52525e-05, Loss_f: 8.18625e-06
Iter 4000, Loss: 2.27102e-05, Loss_u: 1.50505e-05, Loss_f: 7.65967e-06
Iter 4100, Loss: 2.21707e-05, Loss_u: 1.49950e-05, Loss_f: 7.17564e-06
Iter 4200, Loss: 2.16707e-05, Loss_u: 1.49882e-05, Loss_f: 6.68249e-06
Iter 4300, Loss: 2.12378e-05, Loss_u: 1.49758e-05, Loss_f: 6.26192e-06
Iter 4400, Loss: 2.08650e-05, Loss_u: 1.48649e-05, Loss_f: 6.00019e-06
Iter 4500, Loss: 2.05685e-05, Loss_u: 1.46971e-05, Loss_f: 5.87144e-06
Iter 4600, Loss: 2.02867e-05, Loss_u: 1.45492e-05, Loss_f: 5.73748e-06
Iter 4700, Loss: 1.98643e-05, Loss_u: 1.42113e-05, Loss_f: 5.65300e-06
Iter 4800, Loss: 1.95529e-05, Loss_u: 1.39849e-05, Loss_f: 5.56801e-06
Iter 4900, Loss: 1.92663e-05, Loss_u: 1.37903e-05, Loss_f: 5.47597e-06
Iter 5000, Loss: 1.89228e-05, Loss_u: 1.35481e-05, Loss_f: 5.37472e-06
Iter 5100, Loss: 1.84236e-05, Loss_u: 1.34157e-05, Loss_f: 5.00791e-06
Iter 5200, Loss: 1.80063e-05, Loss_u: 1.32191e-05, Loss_f: 4.78724e-06
Iter 5300, Loss: 1.76915e-05, Loss_u: 1.30139e-05, Loss_f: 4.67755e-06
Iter 5400, Loss: 1.73783e-05, Loss_u: 1.29044e-05, Loss_f: 4.47391e-06
Iter 5500, Loss: 1.70338e-05, Loss_u: 1.28041e-05, Loss_f: 4.22965e-06
Iter 5600, Loss: 1.67837e-05, Loss_u: 1.25964e-05, Loss_f: 4.18733e-06
Iter 5700, Loss: 1.65015e-05, Loss_u: 1.23539e-05, Loss_f: 4.14762e-06
Iter 5800, Loss: 1.60735e-05, Loss_u: 1.19513e-05, Loss_f: 4.12214e-06
Iter 5900, Loss: 1.56812e-05, Loss_u: 1.17418e-05, Loss_f: 3.93935e-06
Iter 6000, Loss: 1.53537e-05, Loss_u: 1.15571e-05, Loss_f: 3.79661e-06
Iter 6100, Loss: 1.50212e-05, Loss_u: 1.13536e-05, Loss_f: 3.66758e-06
Iter 6200, Loss: 1.47634e-05, Loss_u: 1.11160e-05, Loss_f: 3.64741e-06
Iter 6300, Loss: 1.45156e-05, Loss_u: 1.09330e-05, Loss_f: 3.58268e-06
Iter 6400, Loss: 1.42926e-05, Loss_u: 1.07367e-05, Loss_f: 3.55599e-06
Iter 6500, Loss: 1.39978e-05, Loss_u: 1.05633e-05, Loss_f: 3.43452e-06
Iter 6600, Loss: 1.37849e-05, Loss_u: 1.04123e-05, Loss_f: 3.37262e-06
Iter 6700, Loss: 1.35610e-05, Loss_u: 1.02190e-05, Loss_f: 3.34198e-06
Iter 6800, Loss: 1.33833e-05, Loss_u: 1.01437e-05, Loss_f: 3.23964e-06
Iter 6900, Loss: 1.32023e-05, Loss_u: 1.00425e-05, Loss_f: 3.15976e-06
```

In [19]:

```python
nu = 0.01/math.pi
fcnn_pi = FCNN().to(device)
optimizer_pi = torch.optim.LBFGS(
    fcnn_pi.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"
)
iteration = 0
def loss_func_pi():
    nu = 0.01/math.pi
    optimizer_pi.zero_grad()
    # Predict the solution along the intitial and boundary conditions
    xtb = torch.hstack((x_boundary_train[:, None], t_boundary_train[:, None]))
    u_pred = net_u(fcnn_pi,xtb)
    # Predict the solution at the sampled co-location points
    xtf = torch.hstack((x_sampled_train[:, None], t_sampled_train[:, None]))
    f_pred = net_u(fcnn_pi,xtf)
    # Compute MSE loss on (x,t) points that lie on initial and boundary conditions, PDE
    mse_loss = nn.MSELoss(reduction ='mean')(u_pred, u_boundary_train)
    pde_loss = net_f(fcnn_pi, xtf, nu)
```

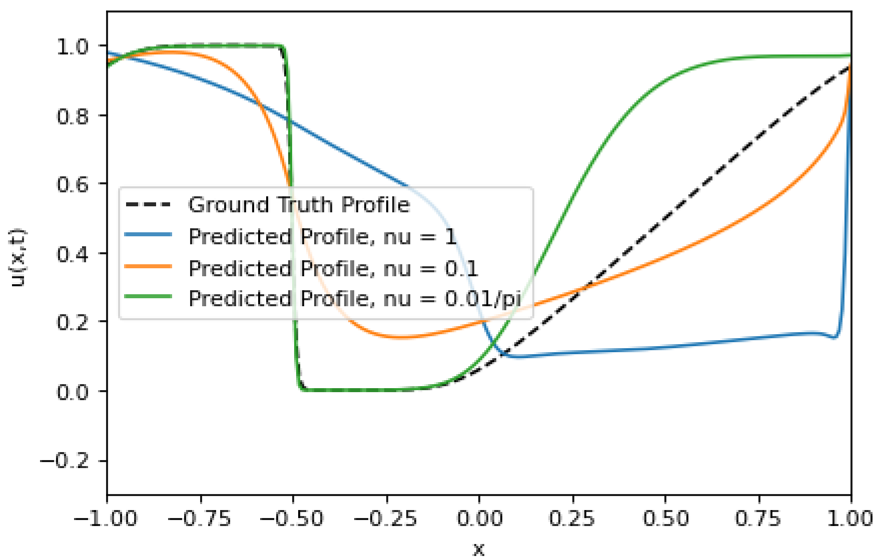
```
    total_loss = mse_loss + pde_loss

    global iteration # iteration keeps track of the current iteration count

    # Uncomment line below to backpropagate loss
    total_loss.backward()
    # Print iteration 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, total_loss.ite
      )
    return total_loss

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

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

u_pred_pi, f_pred_pi = predict(fcnn_pi, xt_combined_flat, nu)
u_pred_grid_pi = griddata(xt_combined_flat, u_pred_pi.flatten(), (xx, tt), method='cubic
```

```
Iter 100, Loss: 7.76655e-02, Loss_u: 5.64516e-02, Loss_f: 2.12139e-02
Iter 200, Loss: 4.93444e-02, Loss_u: 4.04913e-02, Loss_f: 8.85318e-03
Iter 300, Loss: 3.69365e-02, Loss_u: 2.91539e-02, Loss_f: 7.78257e-03
Iter 400, Loss: 7.18876e-03, Loss_u: 8.66090e-04, Loss_f: 6.32267e-03
Iter 500, Loss: 2.11109e-03, Loss_u: 2.86760e-04, Loss_f: 1.82433e-03
Iter 600, Loss: 8.13786e-04, Loss_u: 2.00393e-04, Loss_f: 6.13393e-04
Iter 700, Loss: 5.39842e-04, Loss_u: 1.55354e-04, Loss_f: 3.84487e-04
Iter 800, Loss: 3.92560e-04, Loss_u: 1.56362e-04, Loss_f: 2.36197e-04
Iter 900, Loss: 2.96871e-04, Loss_u: 1.35111e-04, Loss_f: 1.61760e-04
Iter 1000, Loss: 2.50200e-04, Loss_u: 1.14786e-04, Loss_f: 1.35414e-04
Iter 1100, Loss: 2.19269e-04, Loss_u: 1.04692e-04, Loss_f: 1.14577e-04
Iter 1200, Loss: 1.84201e-04, Loss_u: 9.58366e-05, Loss_f: 8.83648e-05
Iter 1300, Loss: 1.56643e-04, Loss_u: 9.19097e-05, Loss_f: 6.47335e-05
Iter 1400, Loss: 1.36957e-04, Loss_u: 8.46569e-05, Loss_f: 5.23000e-05
Iter 1500, Loss: 1.24754e-04, Loss_u: 8.45418e-05, Loss_f: 4.02125e-05
Iter 1600, Loss: 1.18759e-04, Loss_u: 8.01438e-05, Loss_f: 3.86152e-05
Iter 1700, Loss: 1.13472e-04, Loss_u: 7.86811e-05, Loss_f: 3.47904e-05
Iter 1800, Loss: 1.08866e-04, Loss_u: 7.63084e-05, Loss_f: 3.25579e-05
Iter 1900, Loss: 1.04735e-04, Loss_u: 7.45685e-05, Loss_f: 3.01669e-05
Iter 2000, Loss: 9.89730e-05, Loss_u: 6.98107e-05, Loss_f: 2.91623e-05
Iter 2100, Loss: 9.56496e-05, Loss_u: 6.83500e-05, Loss_f: 2.72996e-05
Iter 2200, Loss: 9.06944e-05, Loss_u: 6.40292e-05, Loss_f: 2.66653e-05
Iter 2300, Loss: 8.78539e-05, Loss_u: 6.13927e-05, Loss_f: 2.64612e-05
Iter 2400, Loss: 8.44388e-05, Loss_u: 6.10295e-05, Loss_f: 2.34093e-05
Iter 2500, Loss: 8.20265e-05, Loss_u: 5.82937e-05, Loss_f: 2.37327e-05
Iter 2600, Loss: 8.01809e-05, Loss_u: 5.76960e-05, Loss_f: 2.24850e-05
Iter 2700, Loss: 7.62015e-05, Loss_u: 5.34899e-05, Loss_f: 2.27115e-05
Iter 2800, Loss: 7.32552e-05, Loss_u: 5.18596e-05, Loss_f: 2.13957e-05
Iter 2900, Loss: 7.17739e-05, Loss_u: 5.11188e-05, Loss_f: 2.06551e-05
Iter 3000, Loss: 6.95378e-05, Loss_u: 5.02364e-05, Loss_f: 1.93014e-05
Iter 3100, Loss: 6.80823e-05, Loss_u: 4.95896e-05, Loss_f: 1.84927e-05
Iter 3200, Loss: 6.52783e-05, Loss_u: 4.77751e-05, Loss_f: 1.75032e-05
Iter 3300, Loss: 6.28515e-05, Loss_u: 4.58525e-05, Loss_f: 1.69991e-05
Iter 3400, Loss: 6.11610e-05, Loss_u: 4.43299e-05, Loss_f: 1.68311e-05
Iter 3500, Loss: 5.90505e-05, Loss_u: 4.12629e-05, Loss_f: 1.77876e-05
Iter 3600, Loss: 5.56171e-05, Loss_u: 3.84068e-05, Loss_f: 1.72103e-05
Iter 3700, Loss: 5.25285e-05, Loss_u: 3.47247e-05, Loss_f: 1.78038e-05
Iter 3800, Loss: 4.99070e-05, Loss_u: 3.28798e-05, Loss_f: 1.70272e-05
Iter 3900, Loss: 4.83662e-05, Loss_u: 3.26304e-05, Loss_f: 1.57358e-05
Iter 4000, Loss: 4.62782e-05, Loss_u: 2.98580e-05, Loss_f: 1.64202e-05
Iter 4100, Loss: 4.45460e-05, Loss_u: 2.76783e-05, Loss_f: 1.68677e-05
```

```
Iter 4200, Loss: 4.19520e-05, Loss_u: 2.53785e-05, Loss_f: 1.65735e-05
Iter 4300, Loss: 4.00162e-05, Loss_u: 2.38269e-05, Loss_f: 1.61893e-05
Iter 4400, Loss: 3.80485e-05, Loss_u: 2.26166e-05, Loss_f: 1.54320e-05
Iter 4500, Loss: 3.45227e-05, Loss_u: 2.07978e-05, Loss_f: 1.37249e-05
Iter 4600, Loss: 3.16120e-05, Loss_u: 1.83306e-05, Loss_f: 1.32814e-05
Iter 4700, Loss: 2.99981e-05, Loss_u: 1.69897e-05, Loss_f: 1.30084e-05
Iter 4800, Loss: 2.87751e-05, Loss_u: 1.61158e-05, Loss_f: 1.26593e-05
Iter 4900, Loss: 2.59980e-05, Loss_u: 1.53603e-05, Loss_f: 1.06377e-05
Iter 5000, Loss: 2.42440e-05, Loss_u: 1.46777e-05, Loss_f: 9.56630e-06
Iter 5100, Loss: 2.31490e-05, Loss_u: 1.44830e-05, Loss_f: 8.66598e-06
Iter 5200, Loss: 2.24259e-05, Loss_u: 1.42269e-05, Loss_f: 8.19893e-06
Iter 5300, Loss: 2.15817e-05, Loss_u: 1.41924e-05, Loss_f: 7.38928e-06
Iter 5400, Loss: 2.11293e-05, Loss_u: 1.40662e-05, Loss_f: 7.06311e-06
Iter 5500, Loss: 2.06295e-05, Loss_u: 1.41340e-05, Loss_f: 6.49552e-06
Iter 5600, Loss: 2.03740e-05, Loss_u: 1.39933e-05, Loss_f: 6.38076e-06
```

In [20]:

```
plt.figure(dpi = 80)
plt.plot(x_vector,gt_solution[-1], 'k--', label = 'Ground Truth Profile')
plt.plot(x_vector,u_pred_grid_1[-1], label = 'Predicted Profile, nu = 1')
plt.plot(x_vector,u_pred_grid_0_1[-1], label = 'Predicted Profile, nu = 0.1')
plt.plot(x_vector,u_pred_grid_pi[-1], label = 'Predicted Profile, nu = 0.01/pi')
plt.legend()
plt.ylim([-0.3,1.1])
plt.xlim([-1,1])
plt.xlabel('x')
plt.ylabel('u(x,t)')
plt.show()
```



Part d In terms of the effect of v on the performance, we notice that it took the v/pi state many more iterations to converge to the desired solution but was more accurate to ground truth, with v=0.1 taking the second longest and the quickest being v=1; this proves that there is a speciic v such that we can converge with the least number of iterations required.

Part e PyTorch Autograd functions by first finding the gradient using automatic differentiation of the output corresponding to the differentiation of the inputs.The reason that this is useful when implementing PINNS is due to such layers within the neural network, the gradient count also exponentially rises as we do the back propagation, which Autograd tracks internally if needed.