TIME SERIES DECODER ONLY TRANSFORMER

In [ ]:
```
# IMPORTS:
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as data
import math
import copy
import numpy as np
from transformer_components import *
import pandas as pd
from torch.utils.data import DataLoader, TensorDataset
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error, r2_score

# References:
# https://www.datacamp.com/tutorial/building-a-transformer-with-py-torch
```

Define transformer

In [ ]:
```
class Decoder_Transformer(nn.Module):
    def __init__(self,setups, tgt_vocab_size, d_model, num_heads, num_layers, d_ff,
        super(Decoder_Transformer, self).__init__()
        #self.decoder_embedding = nn.Embedding(tgt_vocab_size, d_model) #REMOVE EMB
        self.positional_encoding = PositionalEncoding(d_model, max_seq_length,setup
        self.decoder_layers = nn.ModuleList([OnlyDecoderLayer(d_model, num_heads, d
        self.fc = nn.Linear(d_model, 2)
        self.dropout = nn.Dropout(dropout)

    def generate_mask(self, tgt):
        tgt_mask = (tgt != 0).unsqueeze(3)
        # print('target', tgt)
        # print('target mask',tgt_mask.shape,tgt_mask)

        seq_length = tgt.size(-1)
        # print('seq leng', seq_length)
        nopeak_mask = (1 - torch.triu(torch.ones(1,2, seq_length, seq_length), diag
        #print(nopeak_mask)
        # print('no peak mask',nopeak_mask.shape)

        tgt_mask = tgt_mask & nopeak_mask
        # print('mask',tgt_mask)


        return tgt_mask

    def forward(self, tgt):
```

```
        tgt_mask = self.generate_mask(tgt)
        # print("tgt",tgt)
        # print('pos enc', self.positional_encoding(tgt))
        tgt_embedded = self.dropout(self.positional_encoding(tgt))


        dec_output = tgt_embedded
        for dec_layer in self.decoder_layers:
            dec_output = dec_layer(dec_output, tgt_mask)

        # print('dec output', dec_output.shape,dec_output)
        # print('dec output[:, -1, :]', dec_output[:, -1, :].shape,dec_output[:, -1
        output = self.fc(dec_output[:, -1, :])
        return output
```

Prepare Data

Data is generated from an optimal control problem with input var x and control var u subject to

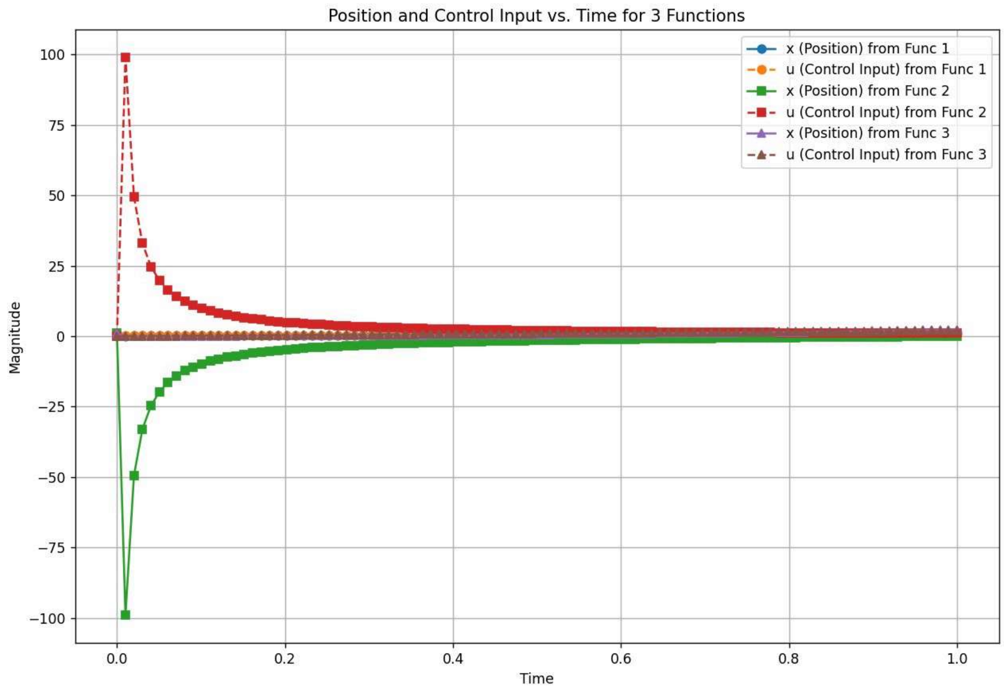
```
dx/dt = 1 + u(t)**2
x(0) = 1
```

Data is generated using the follwoing functions:

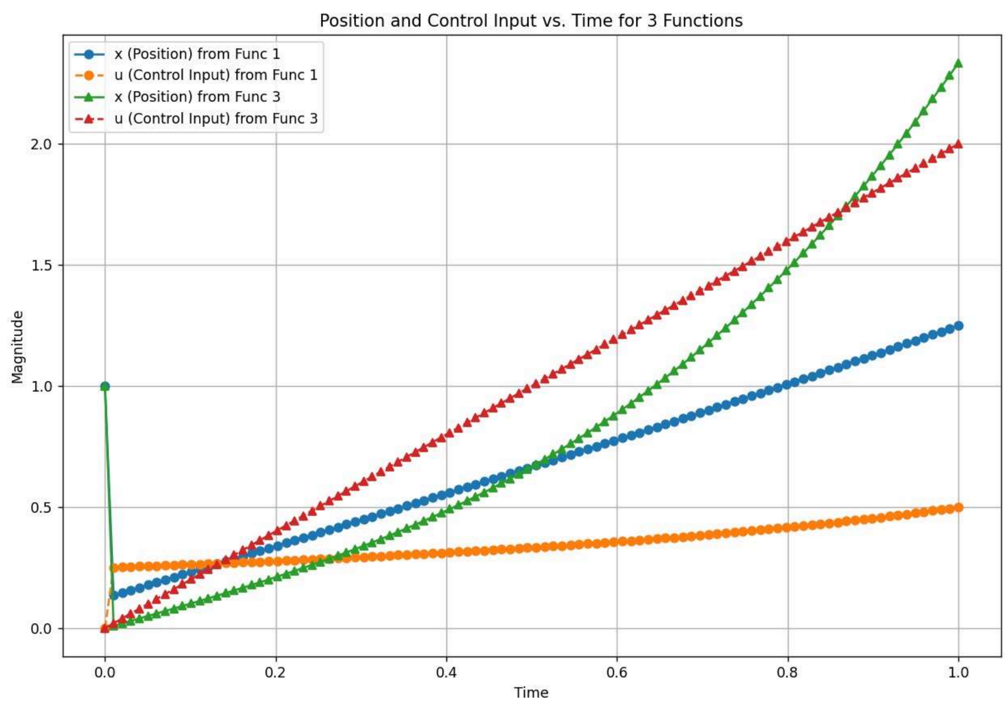
Function 1 (optimal): u = 1/(2*(2-t)), x = t + 1/(8-(4*t))

Function 2: u = 1/t, x = t - (1/t)

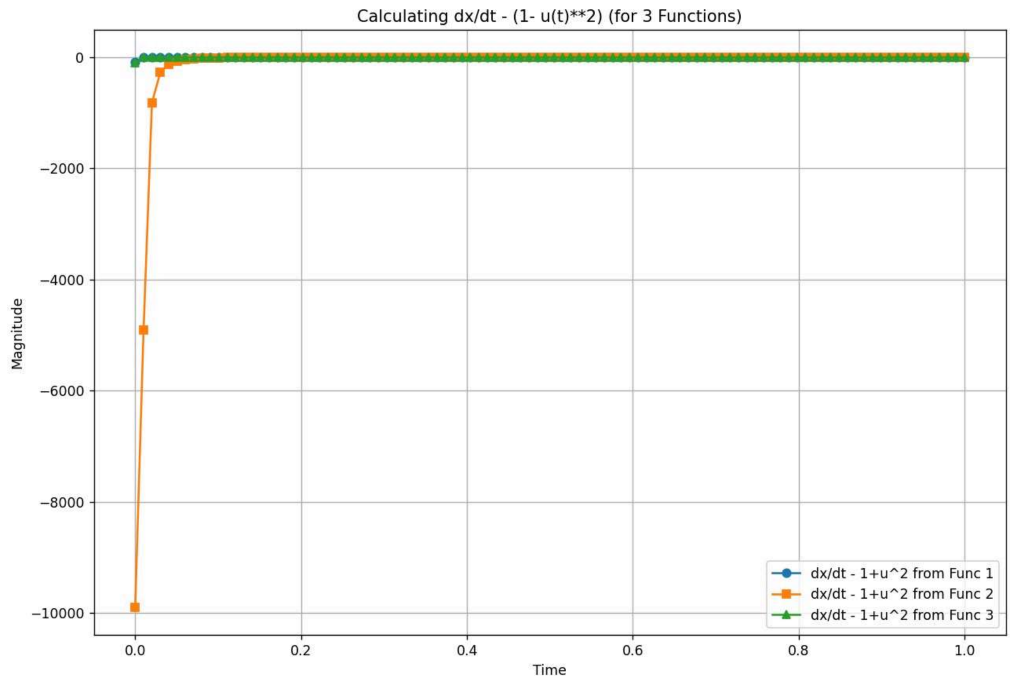
Function 3: u = 2*t, x = t + (4*(t**3))/3

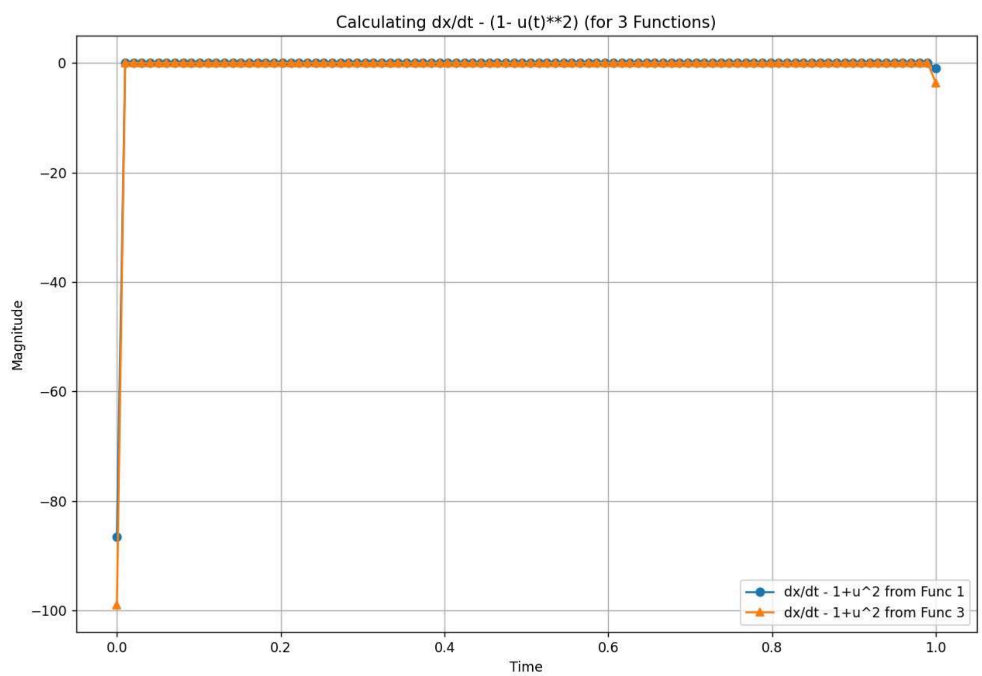


Position and Control Input vs. Time for 3 Functions
x (Position) from Func 1
u (Control Input) from Func 1
x (Position) from Func 2
u (Control Input) from Func 2
x (Position) from Func 3
u (Control Input) from Func 3
Magnitude
Time
100
75
50
25
0
−25
−50
−75
−100
0.0
0.2
0.4
0.6
0.8
1.0



Position and Control Input vs. Time for 3 Functions
x (Position) from Func 1
u (Control Input) from Func 1
x (Position) from Func 3
u (Control Input) from Func 3
Magnitude
Time
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0





In [ ]:
```
# Define data processing parameters
num_setups = 3      # number of different u functions used to generate data
split_time = 0.5   # time to split data from a scale of 0 to 1
```

In [ ]:
```
# Read data file
df = pd.read_csv("data.csv",sep=',', header=0,index_col=False)

# Replace inf with NaN
df.replace([np.inf, -np.inf], np.nan, inplace=True)

# Forward fill NaNs (fill with last valid value)
df.fillna(method='ffill', inplace=True)
```

```
# Split timeseries data into training and test data
train_data = df[df['time'] < split_time]
test_data = df[df['time'] >= split_time]

# Select position and control columns
train_array = train_data[['x', 'u']]
test_array = test_data[['x', 'u']]

print('train shape: ', train_array.shape)
print('test shape:  ', test_array.shape)
#Make noisey to test data
#test_array['x'] = 10*(np.exp( np.linspace(0,1,num=150))+ 7*np.random.rand(test_arr
#print(test_array.shape)

# Rescale data
scaler = StandardScaler() #rescale z = (x - mean) / std --> mean 0 std of 1
scaled_train_data = scaler.fit_transform(train_array)#.flatten().tolist()
scaled_test_data = scaler.transform(test_array)#.flatten().tolist()

# Print
print('df: \n',df.shape,df.head())
print('scaled train shape: ', scaled_train_data.shape)
print('scaled test shape:  ', scaled_test_data.shape)
```

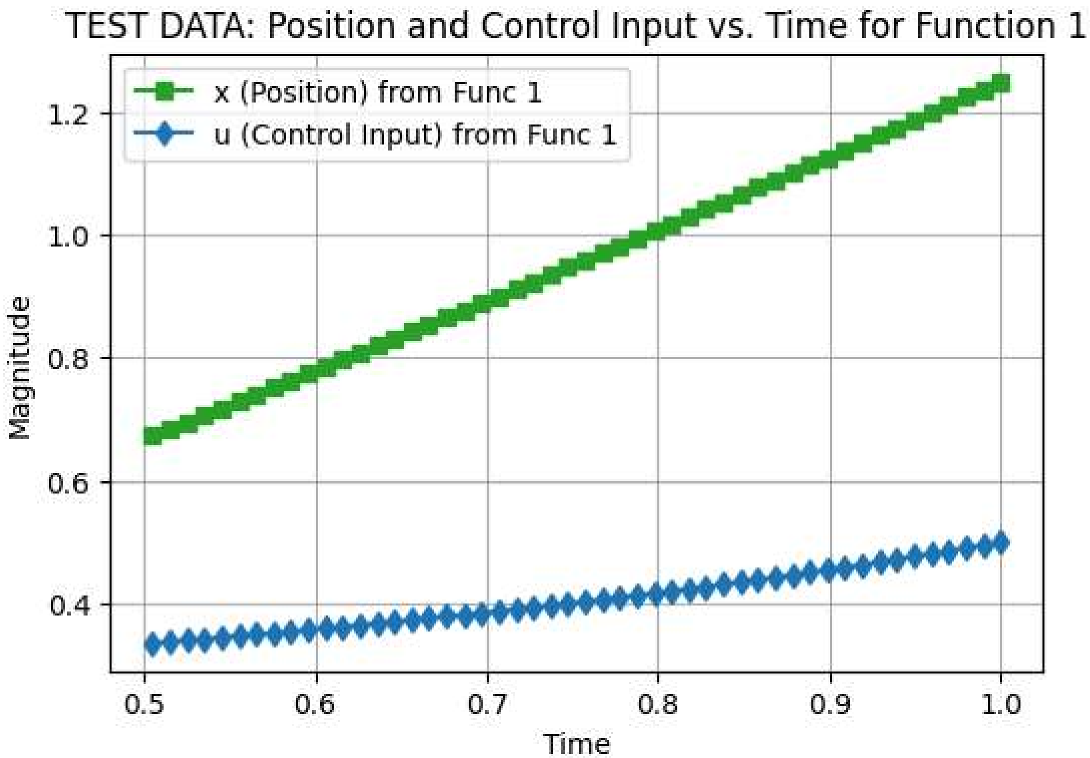
```
train shape:  (150, 2)
test shape:   (150, 2)
df: 
 (300, 3)        time         u         x
0  0.000000  0.000000  1.000000
1  0.010101  0.251269  0.135736
2  0.020202  0.252551  0.146478
3  0.030303  0.253846  0.157226
4  0.040404  0.255155  0.167981
scaled train shape:  (150, 2)
scaled test shape:   (150, 2)
```

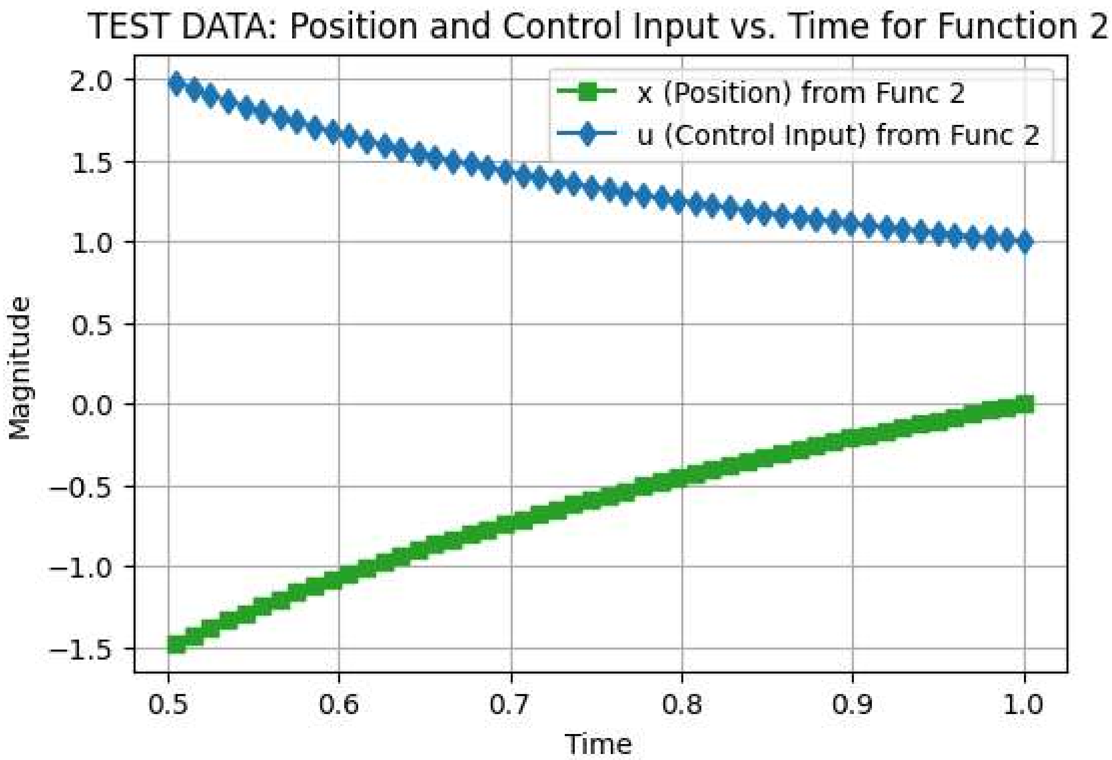
In [ ]:

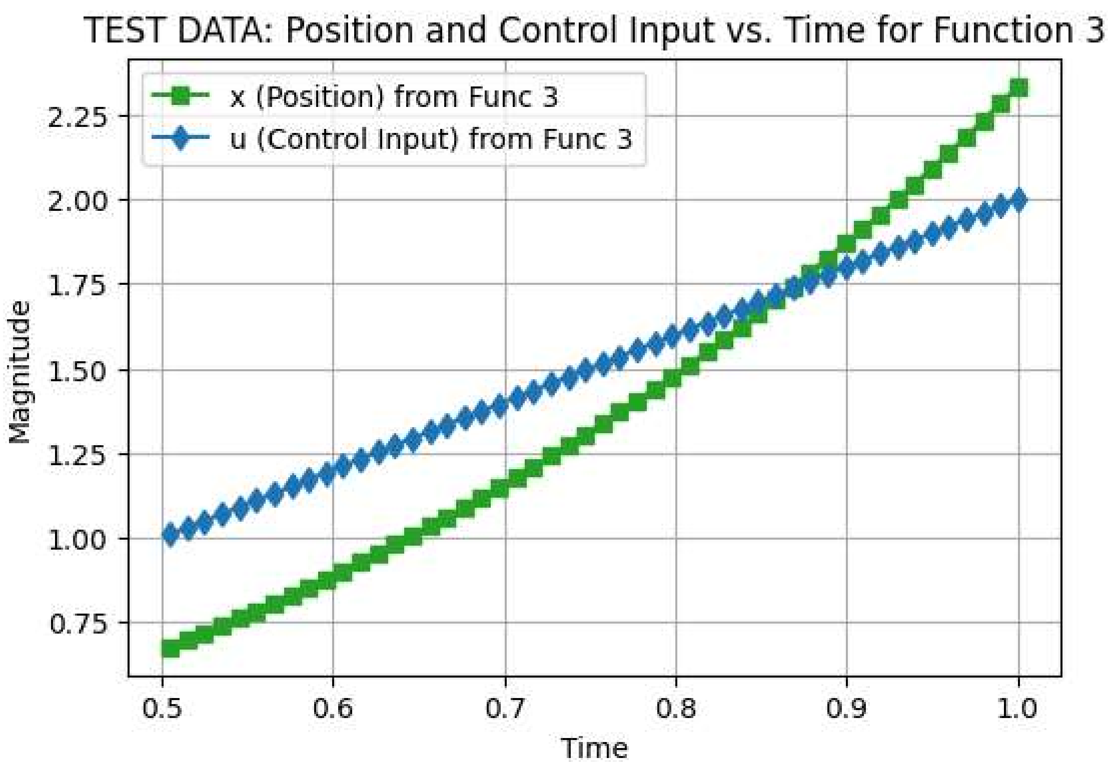
```
# Select position and control columns
plot_train_array = train_data[['x', 'u']]
plot_test_array = test_data[['x', 'u']]
train_time = train_data[['time']]
test_time = test_data[['time']]


#Make noisey to test data
#test_array['x'] = 10*(np.exp( np.linspace(0,1,num=150))+ 7*np.random.rand(test_arr
#print(test_array.shape)

# Reshape test and train arrays
plot_train_array = plot_train_array.to_numpy().reshape(num_setups, plot_train_array
plot_test_array = plot_test_array.to_numpy().reshape(num_setups, plot_test_array.sh
#test_array = test_array.reshape(num_setups, test_array.shape[0]//num_setups,2)
train_time = train_time.to_numpy().reshape(num_setups,  train_time.shape[0]//num_se
test_time = test_time.to_numpy().reshape(num_setups,  test_time.shape[0]//num_setup

print(test_time.shape)
```

```
for i in range(plot_train_array.shape[0]):
    plt.figure(i,figsize=(6, 4))
    plt.plot(test_time[0], plot_test_array[i,:, 0], 's-',color='C2', label=f'x (Pos
    plt.plot(test_time[0], plot_test_array[i,:, 1], 'd-',color='C0', label=f'u (Con

    plt.title(f'TEST DATA: Position and Control Input vs. Time for Function {i+1}')
    plt.xlabel('Time')
    plt.ylabel('Magnitude')
    plt.legend()
    plt.grid(True)
    plt.show()
```

```
(3, 50)
```





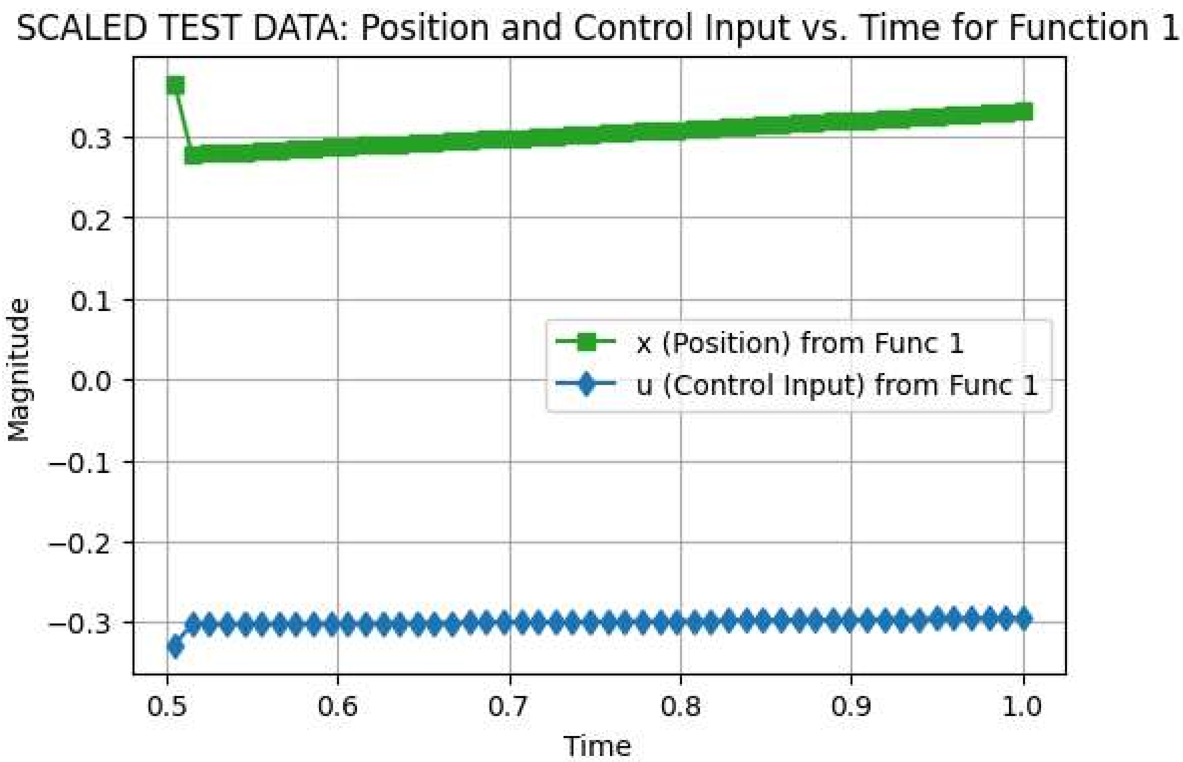


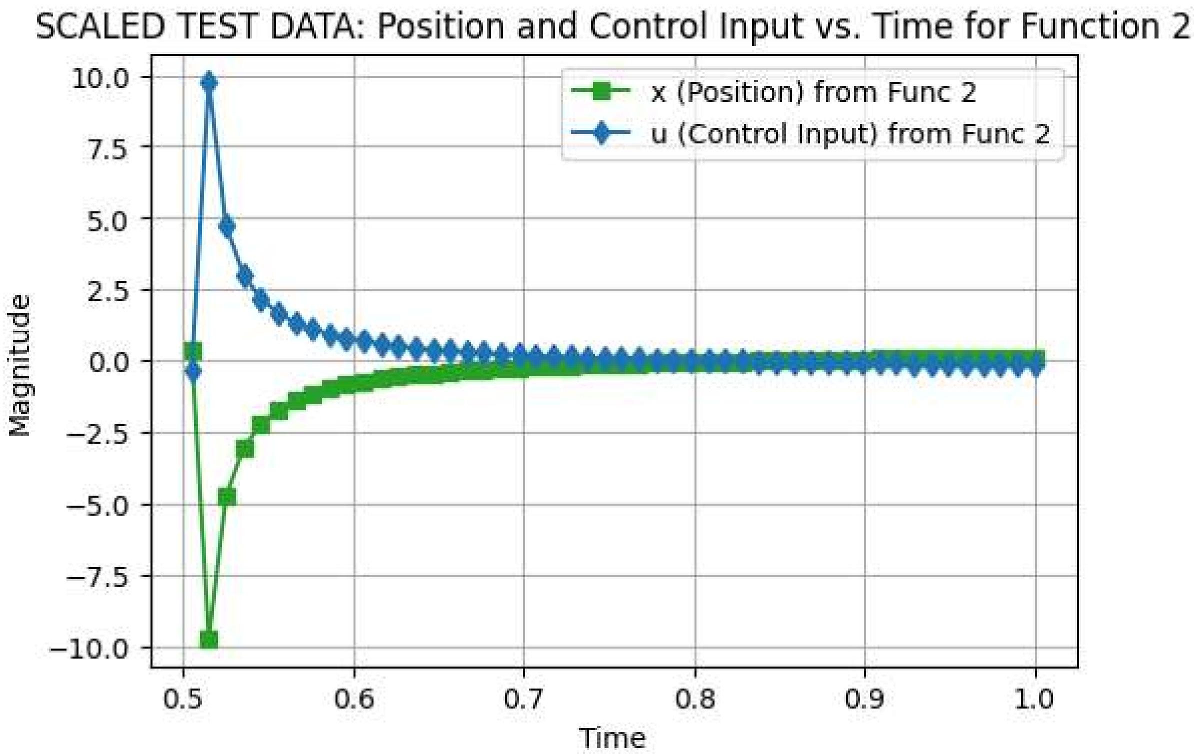
In [ ]:

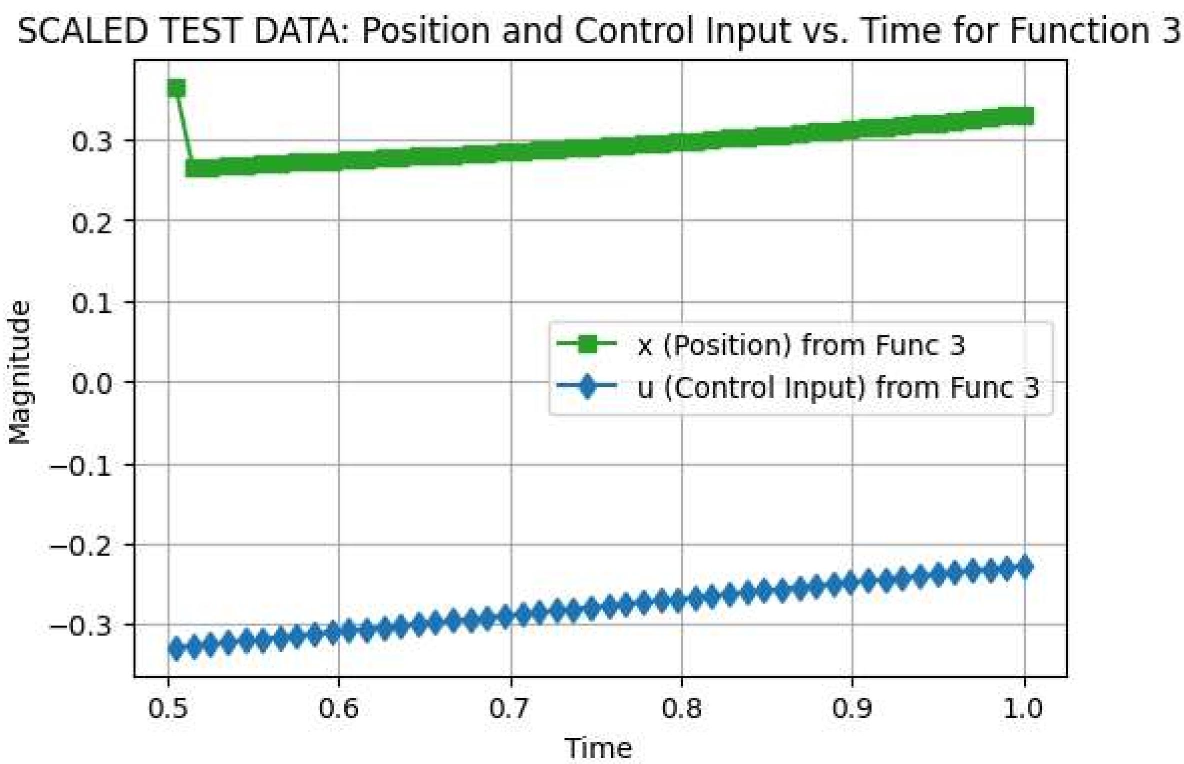
```
# Reshape test and train arrays
train_array = scaled_train_data.reshape(num_setups, scaled_train_data.shape[0]//num
test_array = scaled_train_data.reshape(num_setups, scaled_train_data.shape[0]//num_


# Print results
# print('train: ', train_array)
# print('test:  ', test_array)
```

```python
print('train shape: ', train_array.shape)
print('test shape:  ', test_array.shape)

#print('train data: ',train_array)
for i in range(plot_train_array.shape[0]):
    plt.figure(i,figsize=(6, 4))
    plt.plot(test_time[0], test_array[i,:, 0], 's-',color='C2', label=f'x (Position
    plt.plot(test_time[0], test_array[i,:, 1], 'd-',color='C0', label=f'u (Control

    plt.title(f'SCALED TEST DATA: Position and Control Input vs. Time for Function
    plt.xlabel('Time')
    plt.ylabel('Magnitude')
    plt.legend()
    plt.grid(True)
    plt.show()
```

```
train shape:  (3, 50, 2)
test shape:   (3, 50, 2)
```







In [ ]:
```
def create_sequences(data, sequence_length):
    inputs_var = []
    targets_var =[]
    inputs_u = []
    targets_u =[]
    inputs_setup = []
    targets_setup =[]
    inputs=[]
```

```
    targets=[]

    for setup in range(data.shape[0]):
        #print('setup',setup)
        # print('input', setup[:,:sequence_length-pred_length])
        # print('target', setup[:,pred_length:])

        # inputs.append(setup[:sequence_length-pred_length,:])
        # ##targets.append(setup[pred_length:sequence_length:,:]) # predict only u
        # targets.append(setup[pred_length:,:])
        #print('setup:', setup)
        inputs_var = []
        targets_var =[]

        for i in range(data.shape[1] - sequence_length):
            inputs = []
            targets = []

            for j in range(data.shape[2]):
                setup_var = data[setup,:,j]
                #print(j, setup_var)
                window = setup_var[i:(i + sequence_length)] # take a window startin
                after_window = setup_var[ i + sequence_length] # value after window

                inputs.append(window)
                targets+=[[after_window]]
                #print('inputs',inputs)

            inputs_var+=[inputs]
            targets_var+=[targets]
            #print('input_vars', inputs_var)

        inputs_setup.append(inputs_var)
        targets_setup.append(targets_var)

    return np.array(inputs_setup), np.array(targets_setup)

# Create sequences
#pred_length = 30   # e.g. pred_length=2 --> data: [1,2,3,4,5], x = [1,2,], y = [3,
# with train shpae 50 and pred_length 30 --> x_train: 0:20 and y_train: 30:50 (no o
### ADD VISUALISATION OF DATA SEQUENCES
sequence_len = 10
x_train, y_train = create_sequences(train_array,sequence_len )
x_test, y_test = create_sequences(test_array,sequence_len )

# Print results
x_train = x_train.reshape(x_train.shape[0]*x_train.shape[1],x_train.shape[2],x_trai
y_train = y_train.reshape(y_train.shape[0]*y_train.shape[1],y_train.shape[2], y_tra

x_test = x_test .reshape(x_test.shape[0]*x_test.shape[1],x_test.shape[2],x_test.sha
y_test  = y_test .reshape(y_test.shape[0]*y_test.shape[1],y_test.shape[2], y_test.s
print('x train shape: ', x_train.shape)
print('y train shape: ', y_train.shape)
print('x test shape: ', x_test.shape)
print('y test shape: ', y_test.shape)
# print('x_train: ', x_train)
```

```
# print('y train: ', y_train)
# print('x test: ', x_test)
# print('y test: ', y_test)
```

```
x train shape:  (120, 2, 10)
y train shape:  (120, 2, 1)
x test shape:  (120, 2, 10)
y test shape:  (120, 2, 1)
```

In [ ]:

```
# Convert test and train data to tensors
x_train_tensor = torch.tensor(x_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
x_test_tensor = torch.tensor(x_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)

# Create TensorDatasets
train_dataset = TensorDataset(x_train_tensor, y_train_tensor)
test_dataset = TensorDataset(x_test_tensor, y_test_tensor)

# Create DataLoaders
train_dataloader = DataLoader(train_dataset, batch_size=x_train.shape[0]//num_setup
test_dataloader = DataLoader(test_dataset, batch_size=y_test.shape[0]//num_setups,

#Print
print('x train shape: ', x_train_tensor.shape) #shape: num setups, num inputs, leng
print('y train shape: ', y_train_tensor.shape)
print('x test shape: ', x_test_tensor.shape)
print('y test shape: ', y_test_tensor.shape)

#print('x train: ', x_train_tensor) #shape: num setups, num inputs, length sequence
# print('y train: ', y_train_tensor)
# print('x test: ', x_test_tensor)
# print('y test: ', y_test_tensor)
```

```
x train shape:  torch.Size([120, 2, 10])
y train shape:  torch.Size([120, 2, 1])
x test shape:  torch.Size([120, 2, 10])
y test shape:  torch.Size([120, 2, 1])
```

Set Transformer Parameters

In [ ]:

```
# define transformer size
d_model = 2                                # dimension of data 2 --> x,u
num_heads = 2                              # number of attention heads for multihead a
num_layers = 1                            # number of decoder layers
d_ff = 100                                 # size of feed forward neural network

# define parameters
max_seq_length = max(x_train_tensor.shape[-1],x_test_tensor.shape[-1])   # maximum
print(max_seq_length)
dropout = 0.8                             # dropout
setups = train_array.shape[0]              # number of setups
tgt_vocab_size = 5                        # for embedding but embedding not used
```

```
# create transformer
transformer = Decoder_Transformer(setups, tgt_vocab_size, d_model, num_heads, num_l
```

```
10
```

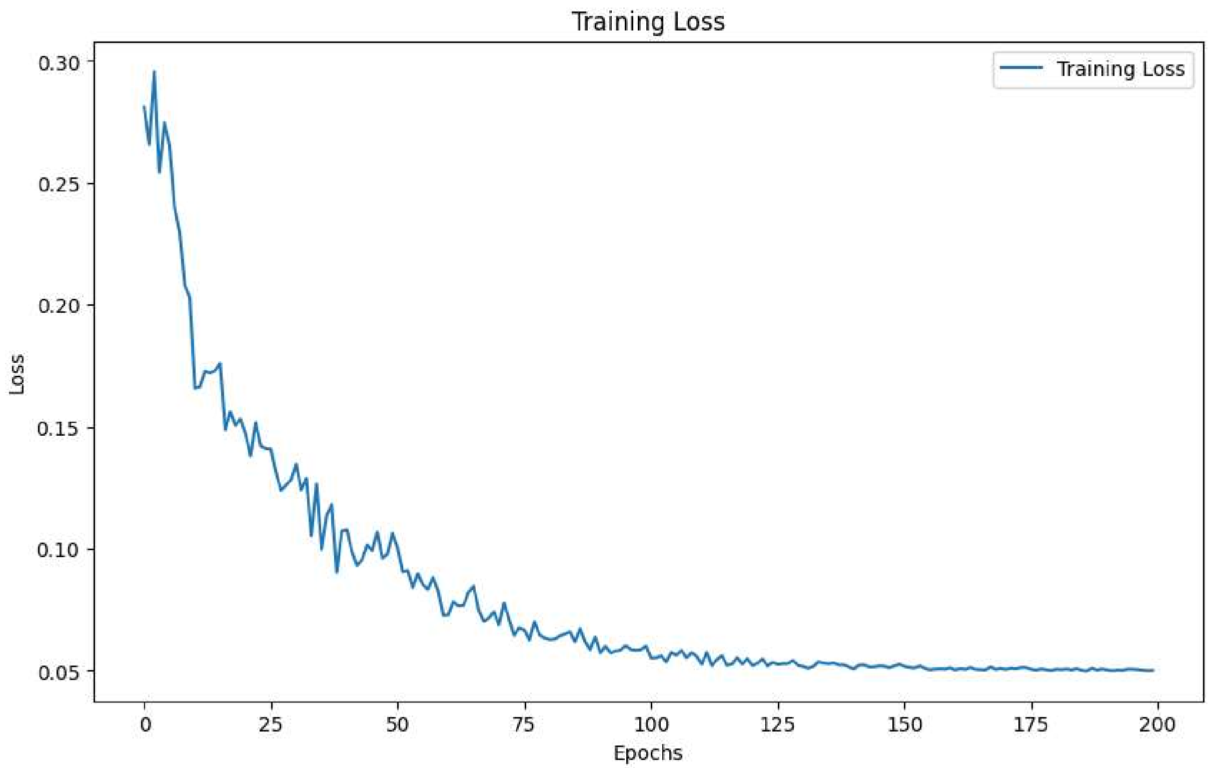
Train Transformer

In [ ]:
```
# define parameters
criterion = nn.MSELoss() #loss criteria
optimizer = optim.Adam(transformer.parameters()) #, lr=0.0001, betas=(0.9, 0.98), e
EPOCH = 200
STOP_EARLY = True #false --> no early stopping
stop_count = 200   # if the loss increases stop_count times in a row, stop early
```

In [ ]:
```
# training loop
transformer.train()

early_stop_count = 0
min_val_loss = float('inf')
train_losses = []
for epoch in range(EPOCH):
    train_loss = 0

    for x, y in train_dataloader: #for each batch
        optimizer.zero_grad()
        #output = transformer(x, y[:, :-1]) #eclude last token from target
        #print('x', x.shape, x)
        output = transformer(x)
        y=y.transpose(1,2).squeeze(1)
        # print('x', x.shape, x)
        # print('y', y.shape, y)
        # print('output',output.shape, output)
        loss = criterion(output, y)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    train_loss /= len(train_dataloader)
    train_losses.append(train_loss)

    if loss <= min_val_loss:
        min_val_loss = loss
        early_stop_count = 0
    else:
        early_stop_count += 1

    #print(f"Epoch: {epoch+1}, Loss: {loss.item()}, Avg Loss of batch: {train_loss}
    if early_stop_count >= stop_count and STOP_EARLY:
        print("Stopping: Loss increasing")
        break
```

In [ ]:
```
#Plot training loss
plt.figure(figsize=(10, 6))
plt.plot(train_losses, label='Training Loss')
```

```python
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss')
plt.legend()
plt.show()
```



Test Transformer

In [ ]:

```python
# testing loop
transformer.eval()

#test_dataloader = DataLoader(test_dataset, batch_size=3, shuffle=False)
test_losses=[]
val_outputs = []
val_inputs = []
val_ys = []
i = 0
with torch.no_grad():
    test_loss=0

    for x, y in test_dataloader:
        print(i+1)
        i += 1

        # print(x.shape)
        # print(y.shape)
        val_output = transformer(x)
        val_y=y.transpose(1,2).squeeze(1)

        val_outputs.append(val_output)
```

```
        val_inputs.append(x)
        val_ys.append(val_y)


print(val_outputs[0].shape)
print(val_ys[0].shape)

# Undo rescaling
for i in range(num_setups):
    val_outputs[i] = scaler.inverse_transform(np.array(val_outputs[i]))
    val_ys[i] = scaler.inverse_transform(np.array(val_ys[i]))

print(val_outputs[0].shape)
print(val_ys[0].shape)
```

```
1
2
3
torch.Size([40, 2])
torch.Size([40, 2])
(40, 2)
(40, 2)
```

In [ ]:

```
# define evaluation functions

def plot_predictions_vs_actual(predictions,inputs, actual, title='Predictions vs Ac
    """
    Plot the actual vs predicted results for data related to a given function of u

    Args:
        predictions (tensor): predicted values (y)
        actual (tensor): actual values including x (inputs) and y (expected output)
        title (str, optional): Title of plot. Defaults to 'Predictions vs Actual'.
    """
    plt.figure(figsize=(6, 4))

    val=""


    for i in range(actual.shape[1]):
        if i ==1:
            val = 'u'
        else:
            val = 'x'



        plt.plot(test_time[0], actual[:,i], '-o',label=f'Actual Values {val}')
        plt.plot(test_time[0,actual.shape[0]-predictions.shape[0]:],predictions[:,i
    plt.axvline(test_time[0,inputs.shape[0]],color='black', linestyle='--', label=f
    plt.title(title)
    plt.xlabel('Time Step')
    plt.ylabel('Value')
    plt.legend()
    plt.show()

def plot_predictions_vs_actual_simple (predictions,actual, title='Predictions vs Ac
```

```
    """
    Plot the actual vs predicted results for data related to a given function of u

    Args:
        predictions (tensor): predicted values (y)
        actual (tensor): actual values including x (inputs) and y (expected output)
        title (str, optional): Title of plot. Defaults to 'Predictions vs Actual'.
    """
    plt.figure(figsize=(6, 4))

    val=""


    for i in range(actual.shape[1]):
        if i ==1:
            val = 'u'
        else:
            val = 'x'

        # print('actual: ',actual[:,i])
        # print('predicted: ',predictions[:,i])
        plt.plot(actual[:,i], '-o',label=f'Actual Values {val}')
        plt.plot(predictions[:,i],'--x', label=f'Predicted Values {val}')
    plt.title(title)
    plt.xlabel('Time Step')
    plt.ylabel('Value')
    plt.legend()
    plt.show()

def rmse_r2(predictions, actual, description ):
    """Calculates goodness of fit using:
    (1) The root mean square error (RMSE)
    (2) The coefficient of determination (R^2)

    Args:
        predictions: predicted values of x,u
        actual: expected values of x,u
        description (_type_): data description

    Returns:
        float: rmse , r_squared
    """
    rmse = []
    r_squared = []
    print(actual.shape)
    for i in range(actual.shape[-1]):
        rmse.append(np.sqrt(mean_squared_error(actual[i], predictions[i])))
        r_squared.append(r2_score(actual[i], predictions[i]))

    df = pd.DataFrame({
        'Description': [description, description],
        'Metric': ['Test RMSE', 'Test R²'],
        'x': [rmse[0], r_squared[0]],
        'u': [rmse[1], r_squared[1]]})

    # Display the DataFrame
```

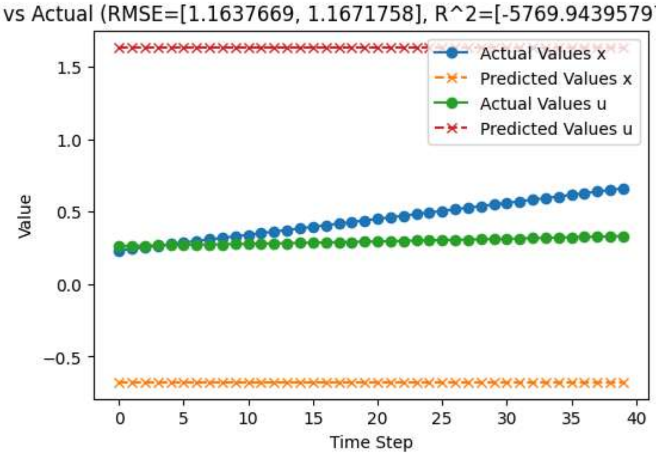
```
    print(df)

    return rmse, r_squared
```

In [ ]:

```
#Metrics function 1 --> batch 1
test_array_tensor = torch.tensor(test_array, dtype=torch.float32)
#print(val_outputs[0]-val_ys[0])
print(val_ys[0].shape)
print(val_outputs[0].shape)
#print('full test sequence',test_array_tensor)

function = 1
rmse, r_squared = rmse_r2(val_outputs[function-1], val_ys[function-1], f'Function {
# plot_predictions_vs_actual(val_outputs[0].squeeze(0),val_inputs[0].transpose(1,2)
plot_predictions_vs_actual_simple(val_outputs[function-1], val_ys[function-1], f'Fu
```

```
(40, 2)
(40, 2)
(40, 2)
  Description     Metric            x            u
0  Function 1  Test RMSE     1.163767     1.167176
1  Function 1    Test R² -5769.943958 -12075.710678
```

Function 1: Predictions vs Actual (RMSE=[1.1637669, 1.1671758], R^2=[-5769.943957976675, -12075.710678418445])

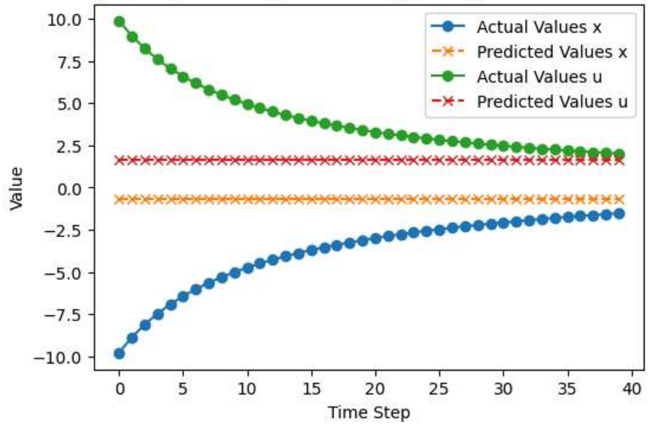


In [ ]:

```
#Metrics function 2
#print(val_outputs[0].squeeze(0)[1])
# print(test_array_tensor)
#rmse_r2(val_outputs[1].squeeze(0),val_ys[1].squeeze(0), 'Function 2')

function = 2
rmse, r_squared = rmse_r2(val_outputs[function-1], val_ys[function-1], f'Function {
# plot_predictions_vs_actual(val_outputs[0].squeeze(0),val_inputs[0].transpose(1,2)
plot_predictions_vs_actual_simple(val_outputs[function-1], val_ys[function-1], f'Fu
```

```
(40, 2)
  Description     Metric         x         u
0  Function 2  Test RMSE  8.704312  7.800214
1  Function 2    Test R²  0.219018  0.239488
```

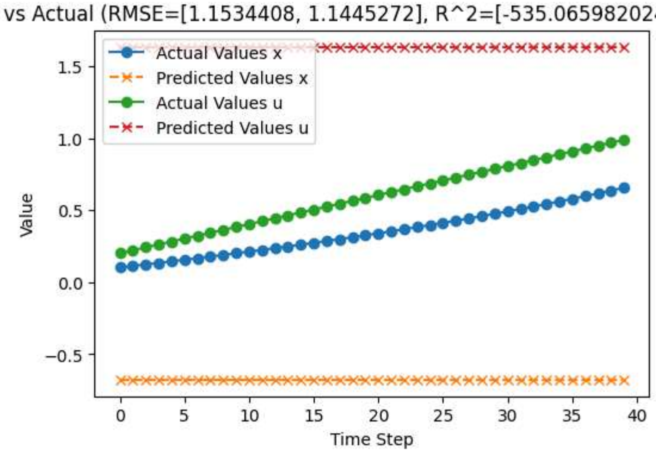
Function 2: Predictions vs Actual (RMSE=[8.704312, 7.8002143], R^2=[0.21901814688050425, 0.23948755336197747])




```
In [ ]: #Metrics function 3

function = 3
rmse, r_squared = rmse_r2(val_outputs[function-1], val_ys[function-1], f'Function {
# plot_predictions_vs_actual(val_outputs[0].squeeze(0),val_inputs[0].transpose(1,2)
plot_predictions_vs_actual_simple(val_outputs[function-1], val_ys[function-1], f'Fu
```

```
(40, 2)
  Description     Metric           x           u
0  Function 3  Test RMSE    1.153441    1.144527
1  Function 3    Test R² -535.065982 -437.746527
```

Function 3: Predictions vs Actual (RMSE=[1.1534408, 1.1445272], R^2=[-535.0659820244839, -437.74652671159987])




```
In [ ]:
```