Final Project Report

Automatic Generation of Simulated Data in Dymola for Training of the Deep Learning Model Used in Power System

Shunyao Xu
ECSE-4170 Modeling and Simulation for Cyber-Physical Systems
Spring 2021

1. Introduction and Background Information:

The modern revolution in machine learning has been largely enabled by access to a large amount of labeled training data because machine learning approaches to tasks like object detection require massive amounts of labeled training data. However, in many cases, obtaining real-world data can be expensive, time-consuming, and inconvenient. In response, generating close-to-reality and limitless data use simulation tools like Dymola seems to be a good choice.

To detect forced oscillation in power systems, 1D and 2D Convolutional Neural Network (CNN) models have been constructed under the Tensorflow and Keras framework in Python, and the simulated data has previously been generated from Python and the Analog Discovery Board. Nevertheless, the data generated by Python lacks randomness, which weakens the ML model's ability to predict on real-world data. The dataset generated by Analog Discovery Board also shares the same disadvantages, as it fails to emulate what really happens in the power system.

Fortunately, Dymola makes it possible to generate close-to-reality data by simulating a model of the power system. Using Dymola Python interfaces, we could also extract as much data as we want by automating the model simulation and data generation process. This report will first talk about the construction of a power system model with noise injection. The injected noise signal will be generated by a signal model, and the noise will induce a forced oscillation into the power system. The whole model will be separated into two subsystems, including “system” and “input signal”. Each subsystem will be configured to be replaceable. Also, this report will describe the generation of simulation data by executing the power system model using Dymola Python interfaces. Enabling the model to be run in Python, the Dymola-Python API will reduce the complexity of extracting the data from the model. Moreover, the data execution process will be automated in python so that we can generate whatever amount of training data we want. At the end, the data will be used to train the machine learning models, and the training result will be analyzed and compared with the previous results.

2. Build the Model in Dymola:

2.1 The Signal Model and The Power System Model

To inject a noise with oscillation into a power system model, we need a signal model and a power system model. The signal model, which generates the synthetic data, is obtained from [1]. It's diagram layer is shown below.

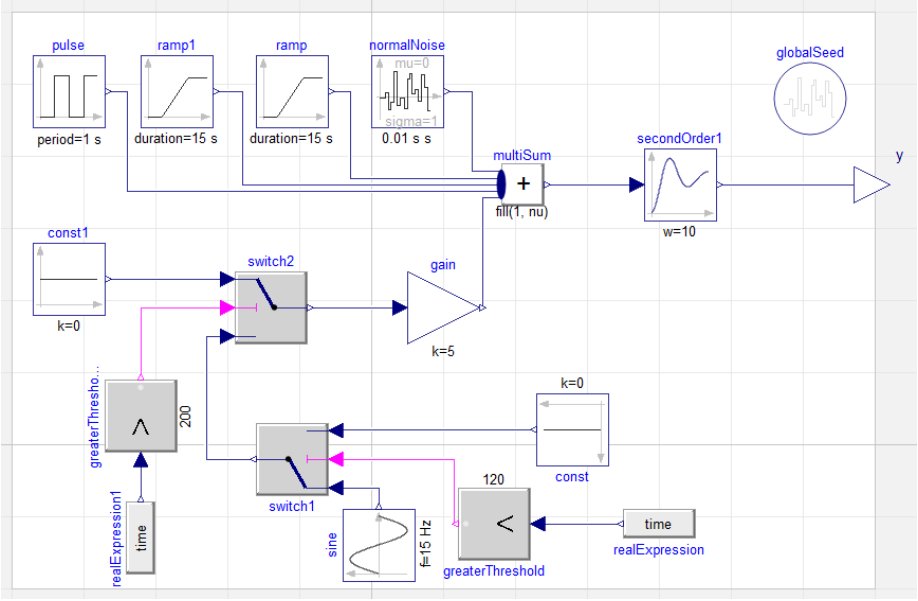


Figure 2.1 Diagram Layer of the Simple Signal Model

Simulating this model for 300s, we could have a basic understanding of its behavior. Figure 2.2 shows a plot of the signal model's output. From the plot, it's easy to observe that the oscillation starts at t=120s and ends at t=200s. This characteristic is adjustable in the model. In the diagram layer, the oscillation start time can be changed by double-clicking the "greaterThreshold" component and modifying the "threshold" parameter. The oscillation end time can be changed by double-clicking the "greaterThreshold1" component and modifying the "threshold" parameter.

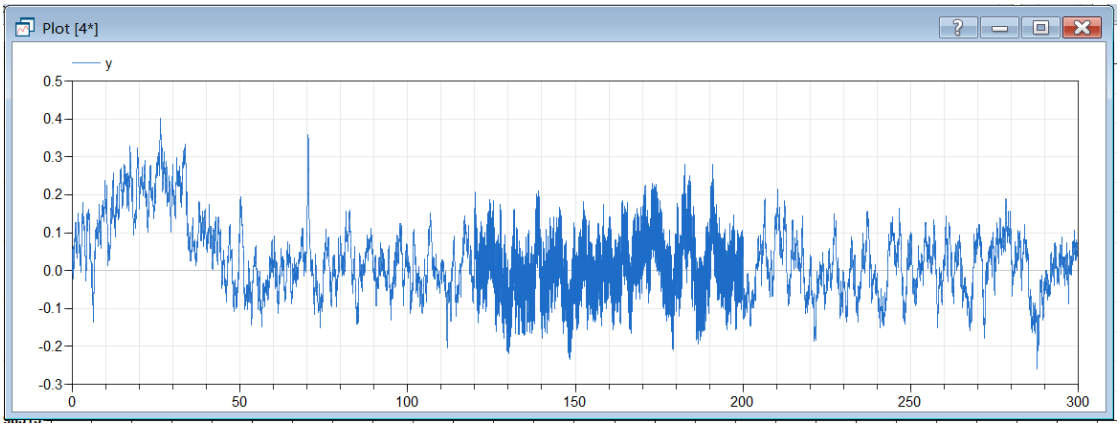


Figure 2.2 Output of the Signal Model

A Single Machine Infinite Bus (SMIB) power system model with load, generator, and noise injection port is obtained from [2], as shown below.

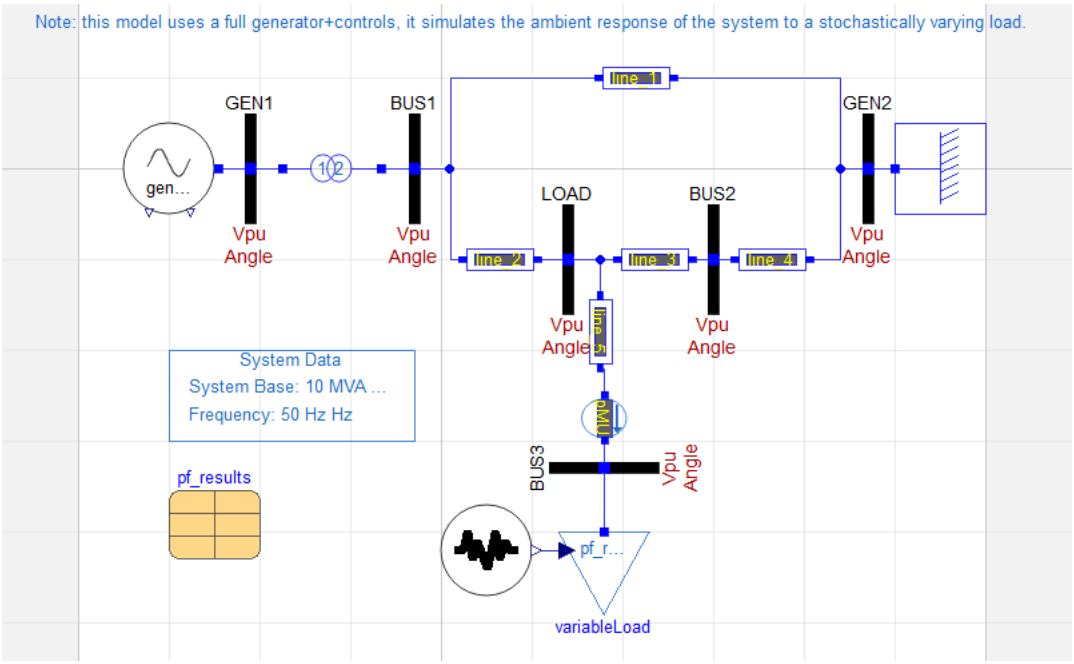


Figure 2.3 Diagram Layer of the SMIB_normal_noise Model

Execution of this model needs two extra libraries - ThermoPower and OpenIPSL. I installed them by first downloading the zip file from Github and then using the Library Management functionality under Tools tab in Dymola. The two libraries are shown below.

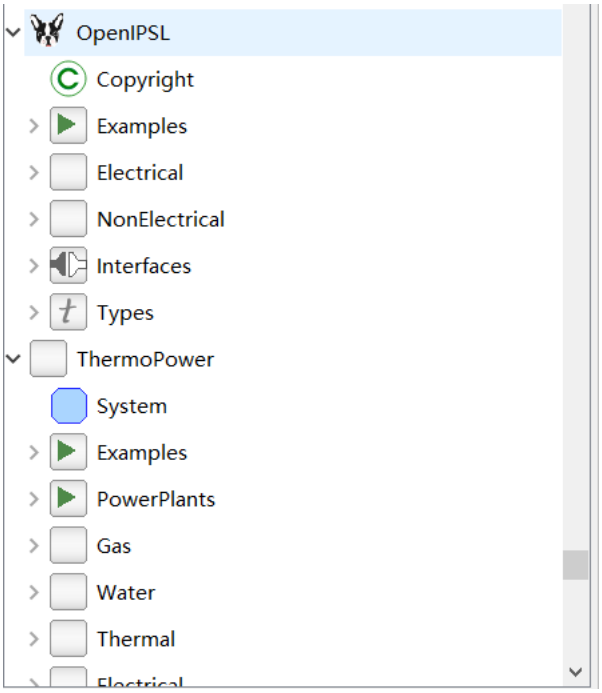


Figure 2.4 Install ThermoPower and OpenIPSL Libraries

2.2 Construct the Replaceable Model

Combining the two models described in section 2.1, we could construct the model we want to generate the data. The whole model will be separated into two subsystems, including “system” and “input signal”. Each subsystem will be configured to be replaceable. In the package browser, I started by creating a new package called “MLTraining” and several sub-packages, as shown below.

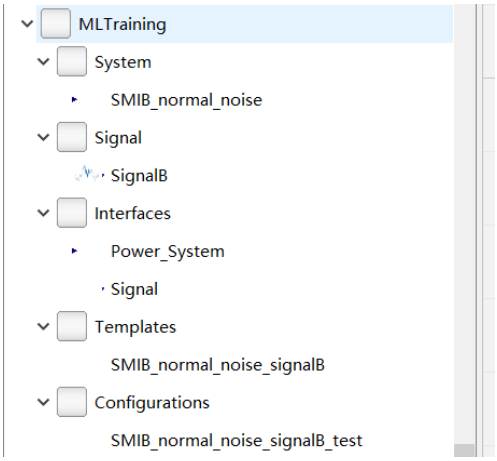


Figure 2.5 Architecture of the Package

In the Interface package, I created two partial models named Power_System and Signal, respectively. Their diagram layers are shown below. The Power_System model contains a Real Input block, and the Signal model contains a Real Output block.

Figure 2.6 Two Interface Models

Then, I created a partial model named “SMIB_normal_noise_signalB” in the “Template” package. A Interfaces.Power_System component and a Interfaces.Signal component are dragged into its diagram layer. The connection is shown below. When the dialog appeared, I clicked on 'make replaceable' to make the components replaceable.

Figure 2.7 Build the Template Model

After that, the partial models for the two subsystems are created. The signal model is created in the “Signal” sub-package by extending the Interfaces.Signal model and then duplicating the components from the original signal model. The 2nd order system in the original model is removed since it’s previously used to represent a power grid.

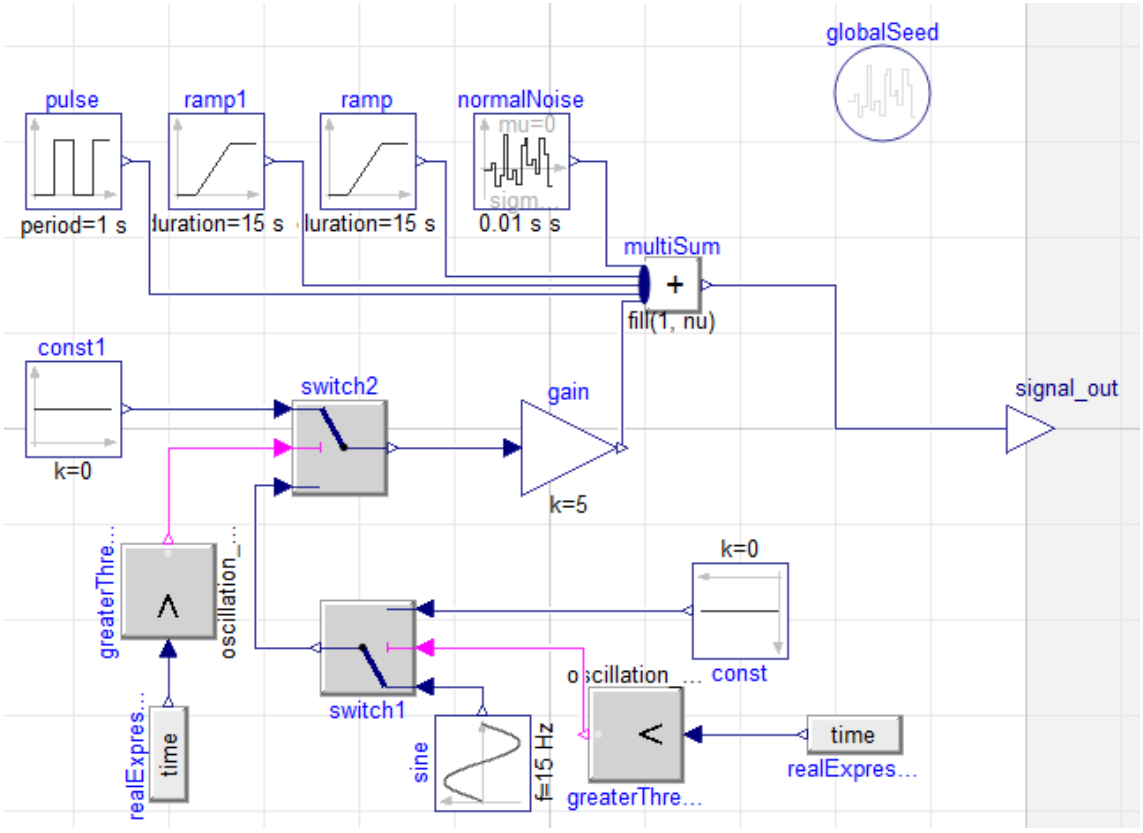


Figure 2.8 Signal Subsystem

The SMIB power system model is created by extending the Interfeaces.Power_System into the "System" sub-package and then duplicating the components from the original SMIB model. The SMIB_normal_noise model is slightly modified. As shown below, serving as the input to the variable load component, the Real Input block replaces the original noise block.

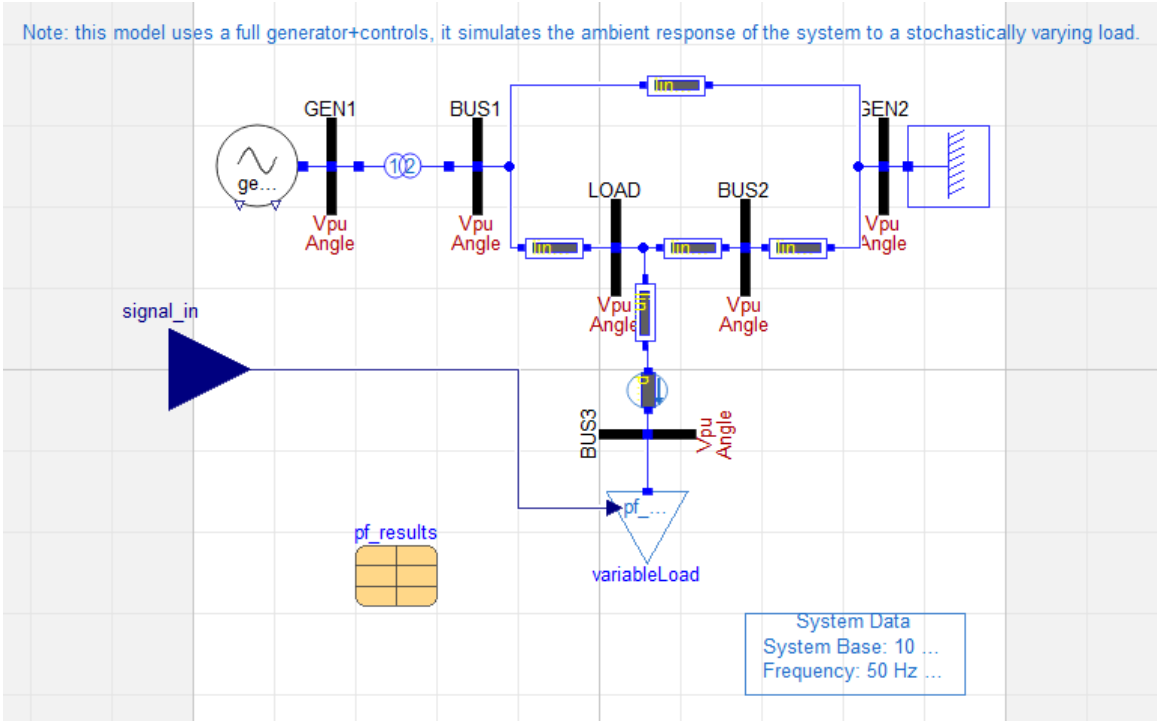


Figure 2.9 SMIB_normal_noise Subsystem

Finally, I extended the template model from Templates.SMIB_normal_noise_signalB to the Configurations sub-package. The new model is named "SMIB_normal_noise_signalB_test". Right-clicking on the signal and power system components, I selected "change class" and "all matching choices". By replacing the components, the diagram layer is shown below. The model now has replaceable subsystems.

Figure 2.10 Finish the Replaceable Model

2.3 Propagate Parameters

To be able to modify the end time and the start time of the oscillation, the parameters are propagated from the subsystem model in Signal.SignalB. The amplitude parameter of the sine source component is also propagated from the subsystem model to modify the amplitude of the oscillation. After propagation, the text layer of Signal.SignalB model is shown below.

```
model SignalB
  extends Interfaces.Signal;

  parameter Real oscillation_amplitude=4 "Amplitude of Oscillation";
  parameter Real oscillation_start=120 "Oscillation Start Time";
  parameter Real oscillation_end=200 "Oscillation End Time";
equation

end SignalB;
```

Figure 2.11 Propagate Three Parameters

In the "SMIB_normal_noise_signalB_test" model, I can now change the value of the propagated parameters. The default value of the oscillation amplitude is set to be 4. The default values of the oscillation start-time and end-time are set to be 120 and 200, respectively.

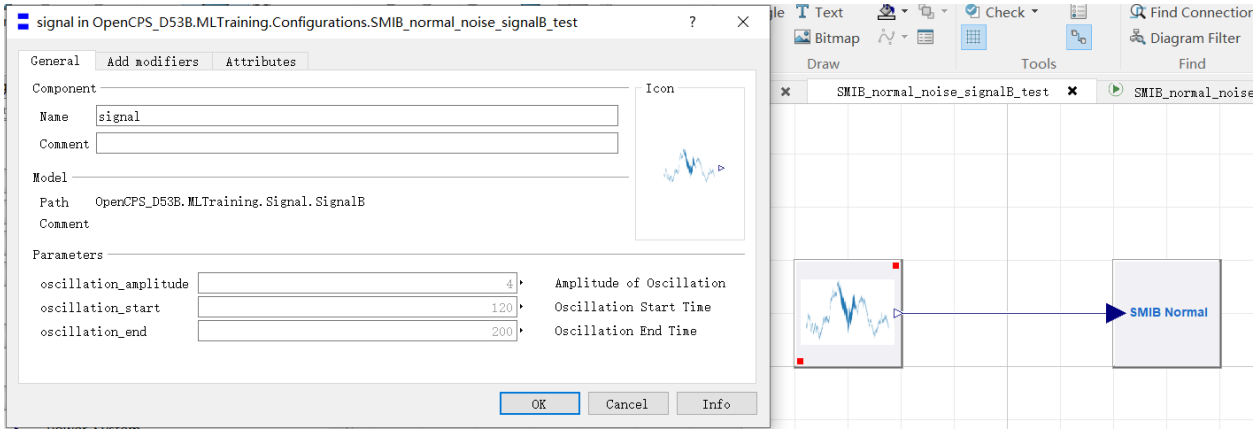


Figure 2.12 Propagate Three Parameters

2.4 Simulate the Model in Dymola

Simulating the SMIB_normal_noise_signalB_test model in Dymola for 300s with Radau solver, I plotted the voltage at bus3. As expected, the oscillation happens at 120s and ends at 200s. By zooming in the oscillation area, it is easy to observe that the oscillation is discernible with good resolution so that the data can be used for ML training.

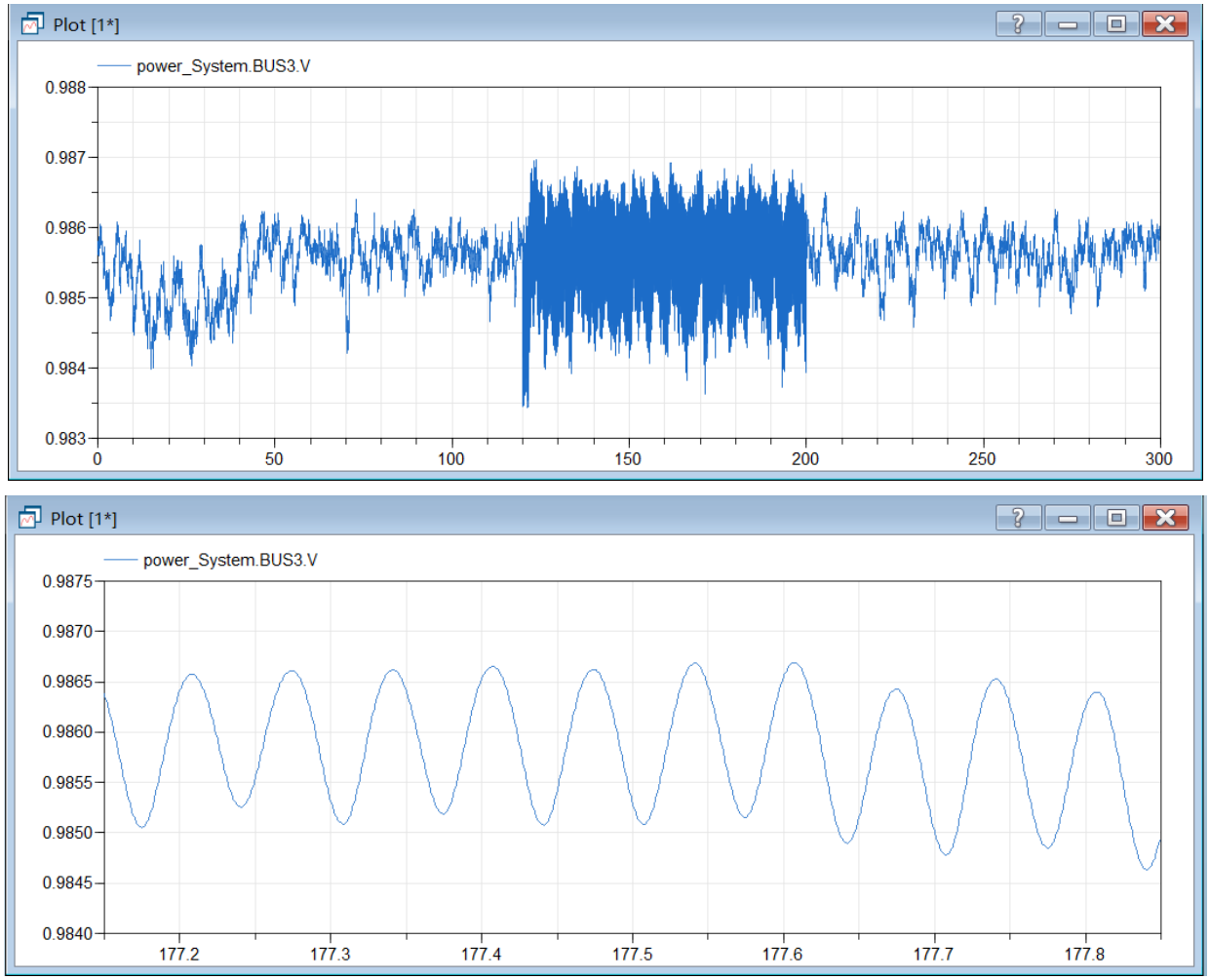


Figure 2.13 Plot of Voltage at BUS3

The frequency measured by the PMU block is also plotted. Again, the oscillation is discernible with good resolution. Therefore, this model is ready to generate data now.

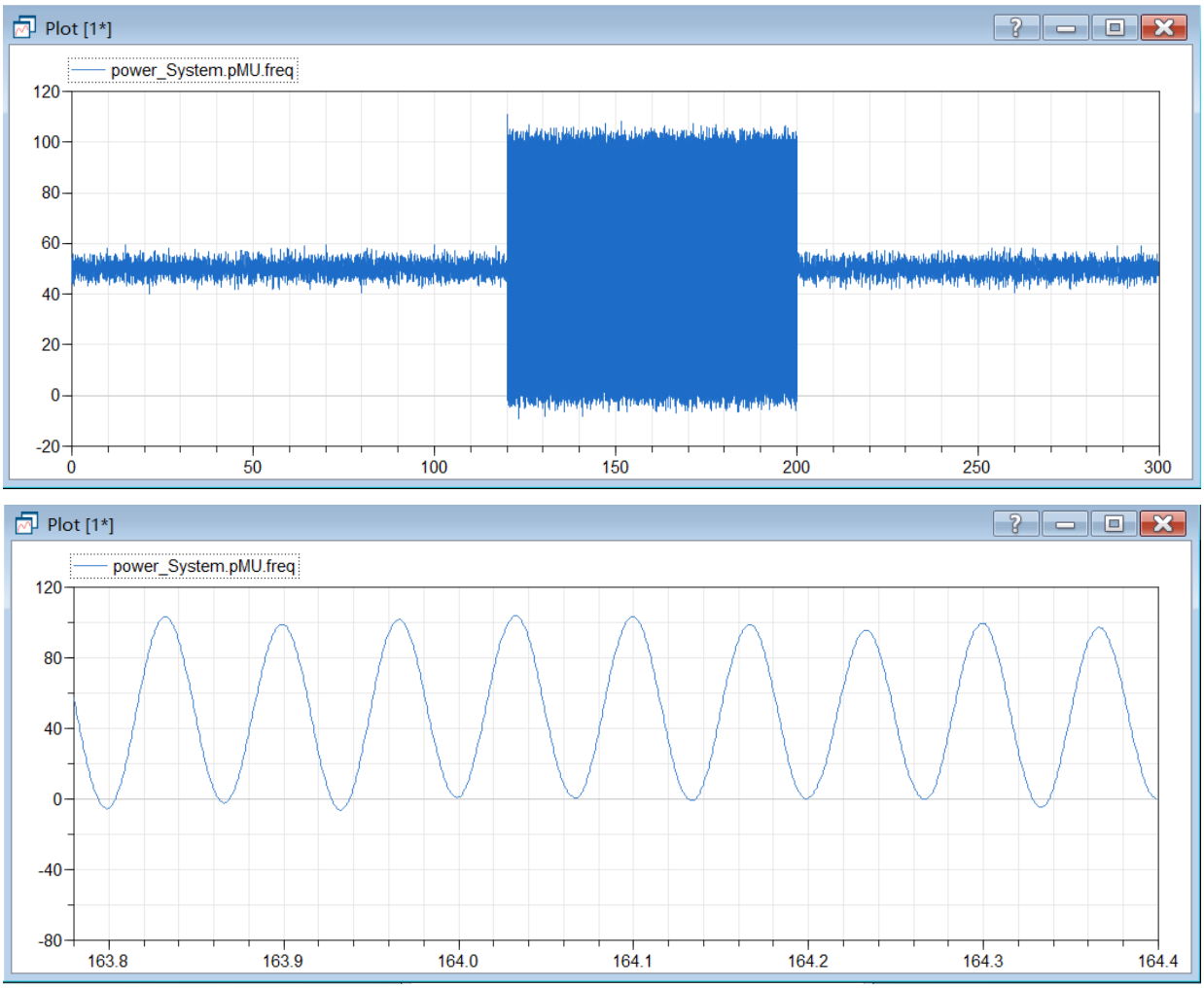
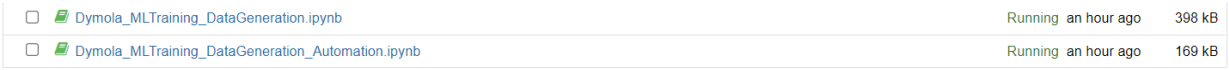


Figure 2.14 Plot of Frequency Measured by PMU

3. Data Generation and Model Training in Python:

The python code is written in two separate files of Jupyter Notebook. One of them just goes through the model simulation, the data generation and the training process. Based on the first file, the second file contains the code that implements the automation methodology for data generation. This section will mainly talk about the result generated by the first file.



3.1 Simulate the Model and Extract the Data using Dymola Python Interfaces

The Dymola Python Interface can be imported as a Python library named dymola. Since this library is not available via pip, the recommended way is to insert the path of the "dymola.egg" file from the Dymola installation repository to the PYTHONPATH environment variable. I did this by using the following code in python.

```
1 import os
2 import sys
3 sys.path.insert(0, os.path.join('E:\\', 'Spring2021', 'MnS4CPS', 'Dymola', 'Modelica',
4                                'Library', 'python_interface', 'dymola.egg'))
```

Figure 3.1 Insert the Path

The code shown in Figure 3.2 is used to simulate the model and extract data from the result file. The code starts by instantiating the Dymola interface. Then, it opens the model I constructed in Section 2. The model is simulated using the "simulateExtendedModel" function since this function allows us to set the initial values of the parameters in the model. After simulation, the data is extracted by opening the .mat file in the working repository. The "readTrajectory" function helps us to specify the data we want. In this case, I only extract data for time, BUS3 voltage, and PMU frequency. These three sets of data are stored in three different lists.

```
dymola = DymolaInterface() # Instantiate the Dymola interface and start Dymola
dymola.openModel("E:/Spring2021/MnS4CPS/Final Project/2018_AmericanModelicaConf_PowerGrid_plus_PowerSystems/Modelica_Models/Openc
dymola.cd("E:/Spring2021/MnS4CPS/Final Project/") # change the working directory
dymola.simulateExtendedModel("OpenCPS_D53B.MLTraining.Configurations.SMIB_normal_noise_signalB_test",
                    startTime = 0,
                    stopTime = 300,
                    outputInterval = 0.001,
                    method = "Radau",
                    tolerance = 0.0001,
                    resultFile = 'MLtraining',
                    initialNames = ["signal.sine.amplitude", "signal.greaterThreshold.threshold", "signal.greaterThreshold1.thres
                    initialValues = [osci_amp, osci_start, osci_end],
                    finalNames = ["signal.sine.amplitude", "signal.greaterThreshold.threshold", "signal.greaterThreshold1.thresh
num_of_rows = dymola.readTrajectorySize("MLtraining.mat")
data = dymola.readTrajectory("MLtraining.mat", ["Time", "power_System.pMU.freq", "power_System.BUS3.V"], num_of_rows)
time = data[0]
PMU_Freq = data[1]
BUS3_V = data[2]
```

Figure 3.2 Model Simulation and Data Extraction

With the generated data, I plot the voltage at BUS3 and the frequency measured by PMU in Python, as shown below.

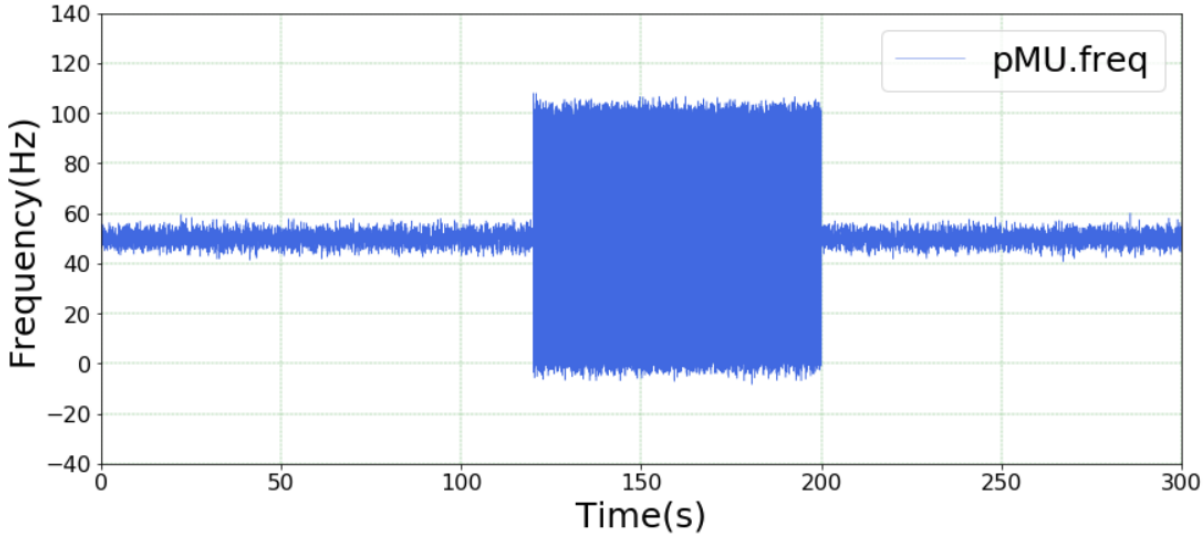


Figure 3.3 Plot of Frequency Measured by PMU

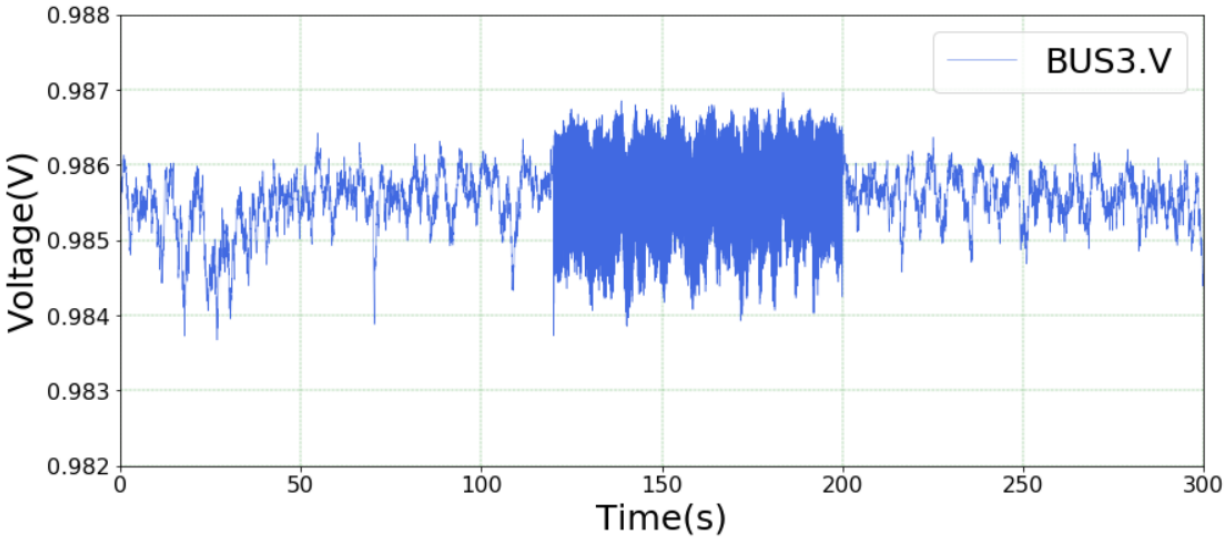


Figure 3.4 Plot of Voltage at BUS3

3.2 Training of the ML Model Using the Generated Data

To train the 1D CNN model using these data, I first need to organize and reshape the dataset into smaller groups so that the data can fit into the architecture of the neural network model. Each group of data also needs to be labeled, namely "1" and "0". Label "0" represents that the data is "during oscillation events". Label "1" represents that the data is "outside oscillation event".

```
while True:
    if time[index] >=290:
        break

    list1[0].append(BUS3_V[index])

    if len(list1[0]) == 31:
        list1 = np.array(list1)
        min_max_scaler = preprocessing.MinMaxScaler()
        x_scaled = min_max_scaler.fit_transform(list1.reshape(-1,1)).reshape((1,31))
        x_train_LSTM = np.reshape(x_scaled, (x_scaled.shape[0], x_scaled.shape[1], 1))
        for i in range(31):
            list2.append(x_train_LSTM[0][i][0])
        dataset.append(list2)
        list2 = []
        list1 = [[]]
        if time[index] < osci_start or time[index] > osci_end:
            labels.append(1)
        else:
            labels.append(0)

    timeindex += increment
    index = int(timeindex)

dataset = np.array(dataset)
labels = np.array(labels)
X_train, X_test, y_train, y_test = train_test_split(dataset, labels, test_size=0.2)
```

Figure 3.5 Reshape the Generated Dataset

Figure 3.6 presents the groups of data that are labeled as “during the oscillation” and “outside the oscillation”. They will be used as input with their labels to train the ML model.

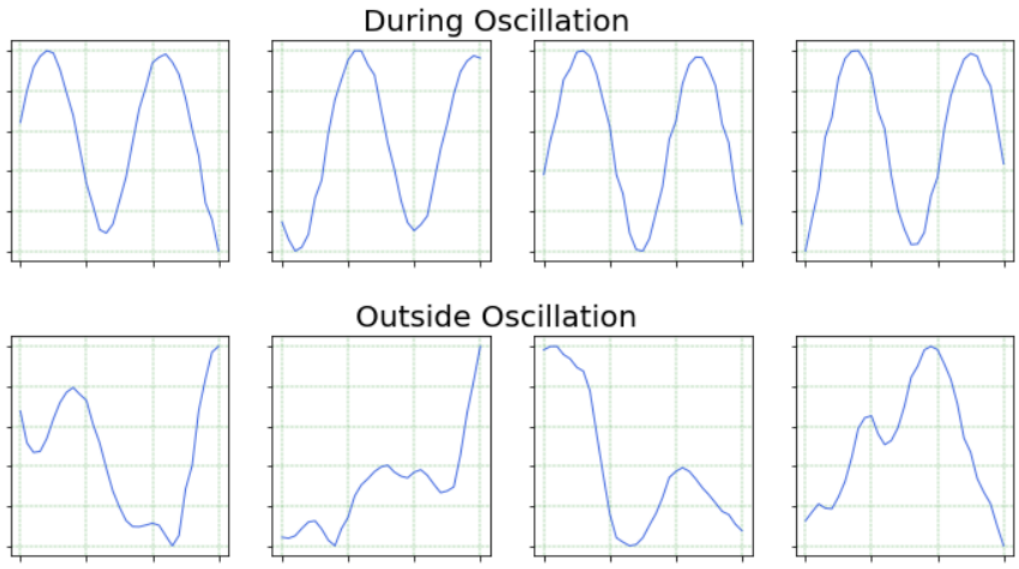


Figure 3.6 Label the Data

Using the following code, the conv1D model was constructed and trained for 50 epochs.

```
TIME_PERIODS = 31
num_time_periods = 31
num_sensors = 1
num_classes = 2
input_shape = (num_time_periods*num_sensors)
kernel_size=3

model_conv1D = Sequential()
model_conv1D.add(Conv1D(64, kernel_size, activation='relu', input_shape=(31,1)))
model_conv1D.add(Conv1D(64, kernel_size, activation='relu'))
model_conv1D.add(MaxPooling1D(2))
model_conv1D.add(Dropout(0.5))
model_conv1D.add(Flatten())
model_conv1D.add(Dense(100, activation='relu'))
model_conv1D.add(Dense(num_classes, activation='softmax'))

model_conv1D.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

print(model_conv1D.summary())

X_train_reshaped = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
X_test_reshaped = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))

history = model_conv1D.fit(X_train_reshaped, y_train, epochs=50,
          batch_size=100, verbose=1,
          validation_data=(X_test_reshaped, y_test))
```

Figure 3.7 Build and Train the ML Model

The training history of the model is shown below.

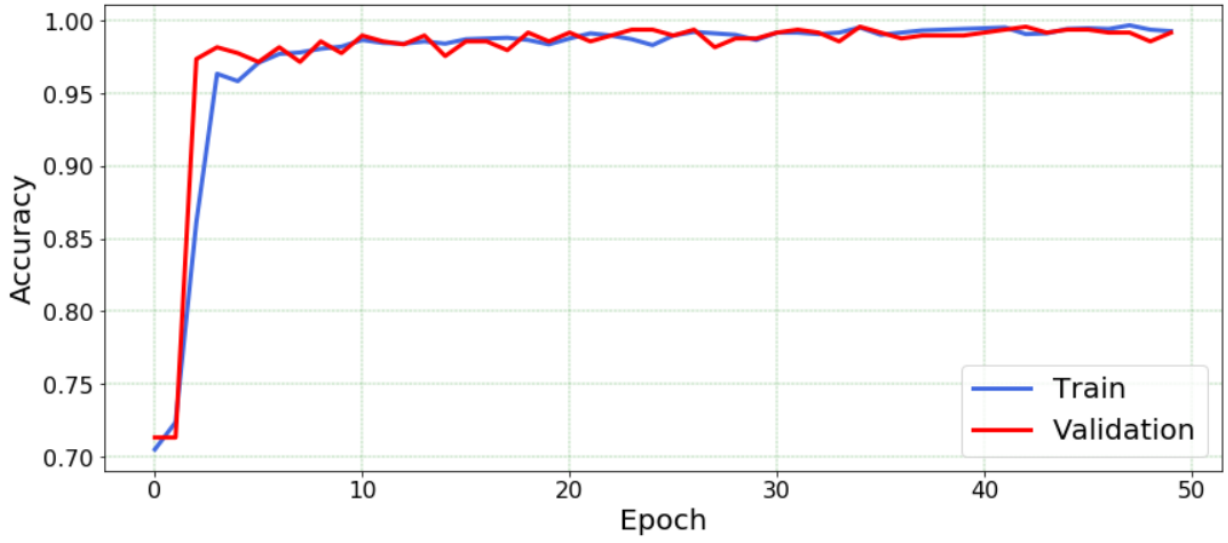


Figure 3.8 Training History of the ML Model

Finally, the trained ML model was used to predict real data from wind farms. The result below shows that the model behaves as expected, and it's inference accuracy is acceptable.

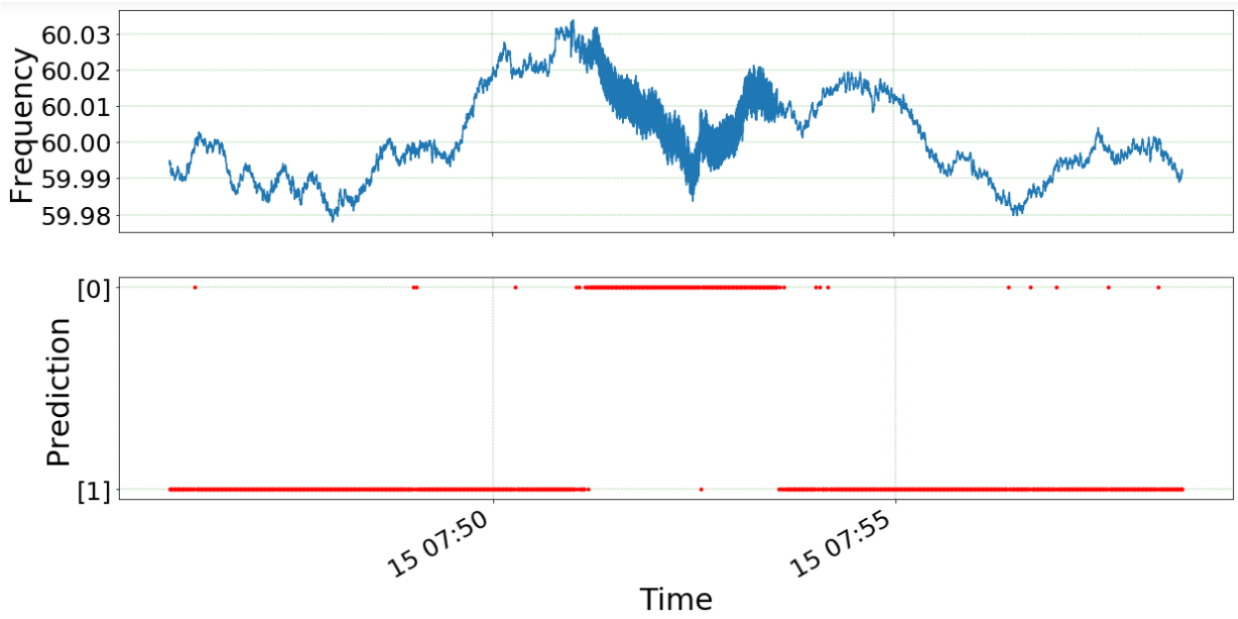


Figure 3.9 Predict on Real Data

4. Automatic Data Generation:

To increase the accuracy of this 1D CNN model, we could generate more data by automating the data generation process. Here is the code to achieve this functionality.

```
# Generation Start

dymola = DymolaInterface() # Instantiate the Dymola interface and start Dymola
dymola.openModel("E:/Spring2021/MnS4CPS/Final Project/2018_AmericanModelicaConf_PowerGrid_plus_PowerSystems/Modelica_Models/
dymola.cd("E:/Spring2021/MnS4CPS/Final Project/") # change the working directory

osci_data_count = 0          # Number of oscillation data that is already generated
nois_data_count = 0          # Number of noisy data that is already generated
stop_generate_osciData = 0   # Set to 1 if enough oscillation data is generated
stop_generate_noisData = 0   # Set to 1 if enough noisy data is generated
dataset = []                 # dataset that stores the training datas
labels = []                  # dataset that stores the labels for the training datas, namely 1 and 0
while True:
    if stop_generate_osciData and stop_generate_noisData:
        break

    dymola.simulateExtendedModel("OpenCPS_D53B.MLTraining.Configurations.SMIB_normal_noise_signalB_test",
                                 startTime = 0,
                                 stopTime = simulation_end,
                                 outputInterval = 0.001,
                                 method = "Radau",
                                 tolerance = 0.0001,
                                 resultFile = 'MLtraining',
                                 initialNames = ["signal.sine.amplitude", "signal.greaterThreshold.threshold", "signal.greate
                                 initialValues = [osci_amp, osci_start, osci_end],
                                 finalNames = ["signal.sine.amplitude", "signal.greaterThreshold.threshold", "signal.greaterT
    num_of_rows = dymola.readTrajectorySize("MLtraining.mat")
    data = dymola.readTrajectory("MLtraining.mat", ["Time", "power_System.pMU.freq", "power_System.BUS3.V"], num_of_rows)

    time = data[0]
    PMU_Freq = data[1]
    BUS3_V = data[2]
    index = 0
    timeindex = 0.0
    increment = 4.5
    num = 31
    list1 = [[]]
    list2 = []
    while True:
        if time[index] >= (simulation_end-10):
            break

        list1[0].append(BUS3_V[index])

        if len(list1[0]) == 31:
            list1 = np.array(list1)
            min_max_scaler = preprocessing.MinMaxScaler()
            x_scaled = min_max_scaler.fit_transform(list1.reshape(-1,1)).reshape((1,31))
            x_train_LSTM = np.reshape(x_scaled, (x_scaled.shape[0], x_scaled.shape[1], 1))
            for i in range(31):
                list2.append(x_train_LSTM[0][i][0])
            if time[index] < (osci_start-1) or time[index] > (osci_end+1):
                if not stop_generate_osciData:
                    dataset.append(list2)
                    print("a")
                    labels.append(1)
                    osci_data_count += 1
                    if osci_data_count == osci_data_num:
                        stop_generate_osciData = 1
            elif time[index] > (osci_start+1) and time[index] < (osci_end-1):
                if not stop_generate_noisData:
                    dataset.append(list2)
                    print("b")
                    labels.append(0)
                    nois_data_count += 1
                    if nois_data_count == nois_data_num:
                        stop_generate_noisData = 1
            list2 = []
            list1 = [[]]

        timeindex += increment
        index = int(timeindex)

dataset = np.array(dataset)
labels = np.array(labels)
X_train, X_test, y_train, y_test = train_test_split(dataset, labels, test_size=0.2)
```

Figure 4.1 Generate Data Automatically

The workflow of this program is also analyzed below. Separating into three subsections, the first part of the program serves as a user interface, asking for a number oscillation data and a number of noisy data that need to be generated. Then, the data generation section will generate the training data by simulating the model. After that, the data will be reshaped and labeled in the data processing section. If the program finds out that the data is not enough compared with the entered numbers, the model will be simulated again until enough training data is generated.

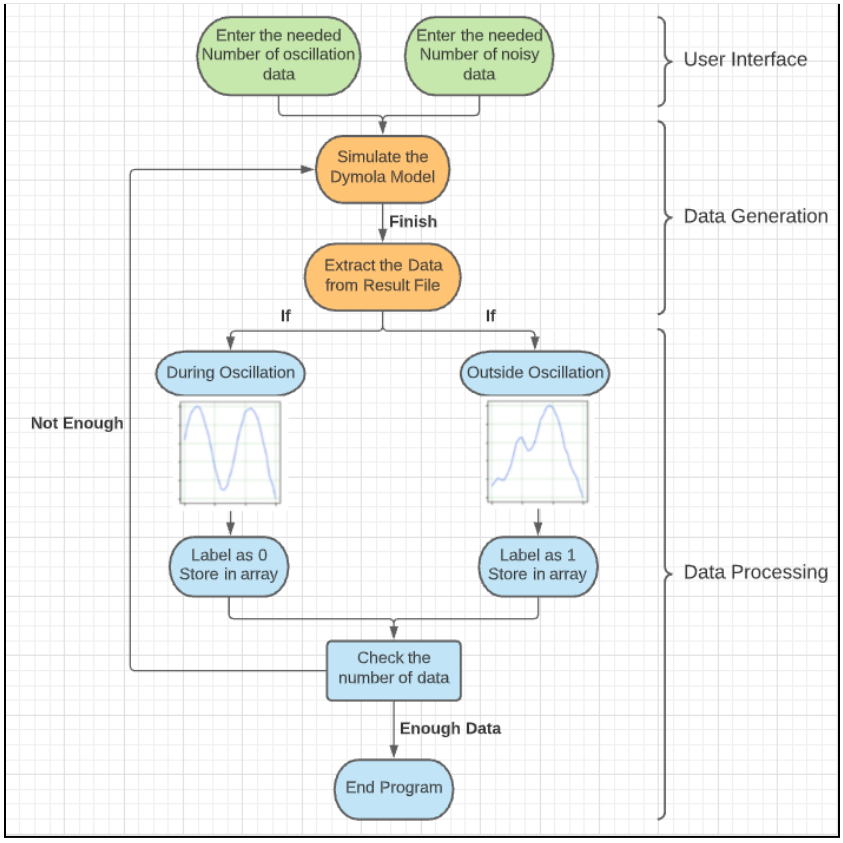


Figure 4.2 Workflow of the Program

The user could enter any amount of data they want to generate. In the case below, the number of oscillation data that needs to be generated is 3000, and the number of noisy data that needs to be generated is also 3000. The program will continuously collect the data until the requested number is fulfilled.

```
# Set the Parameters of the Dymola model
osci_amp = 4            # Oscillation Amplitude
osci_start = 120        # Oscillation Start Time
osci_end = 200          # Oscillation End Time
simulation_end = 300    # Simulation End Time

# Set the amount of data that need to be generated
osci_data_num = 3000    # Number of oscillation data that needs to be generated
nois_data_num = 3000    # Number of noisy data that needs to be generated
```

Figure 4.3 Generate Any Amount of Data

Using the settings in Figure 4.3, 6000 sets of data is generated. The program uses these data as an example to train the model and predict the real data from wind farms, as shown below. By analyzing the result, the inference achieves an accuracy of 0.99074, which is pretty good. To further increase the model's accuracy, we could use more data to train the model by entering a larger number in the last two lines of code in Figure 4.3.

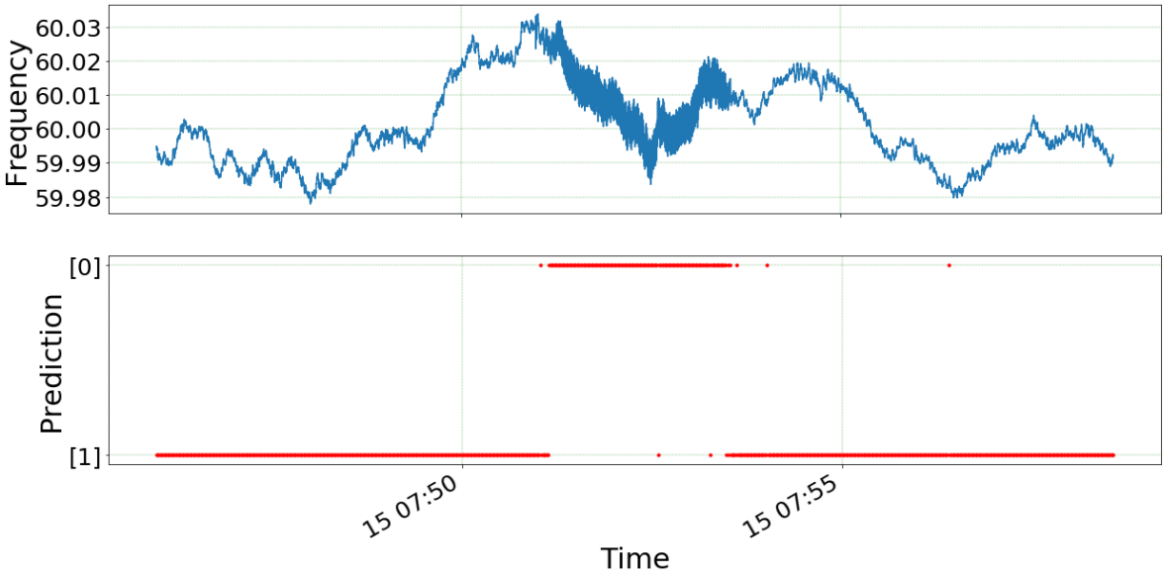


Figure 4.4 Prediction Using the Model Trained by Dymola Data

An experiment is performed to illustrate the relationship between model accuracy and the amount of data used for training. Presenting in the table below, as the number of data under generated increases, the model accuracy is improved. Therefore, in order to achieve a better accuracy, we could generate more data.

Table 1

Num of Oscillation Data	Num of Noisy Data	Num of Error	Accuracy
1000	1000	16	0.97884
2000	2000	14	0.98148
3000	3000	7	0.99074
4000	4000	2	0.99735

Below is the prediction result using the model trained by Python data and the model trained by Analog Discovery Board data. Since they are all predicting on the same set of real data, it's clear that the model trained by Dymola data is performing much better than the other two.

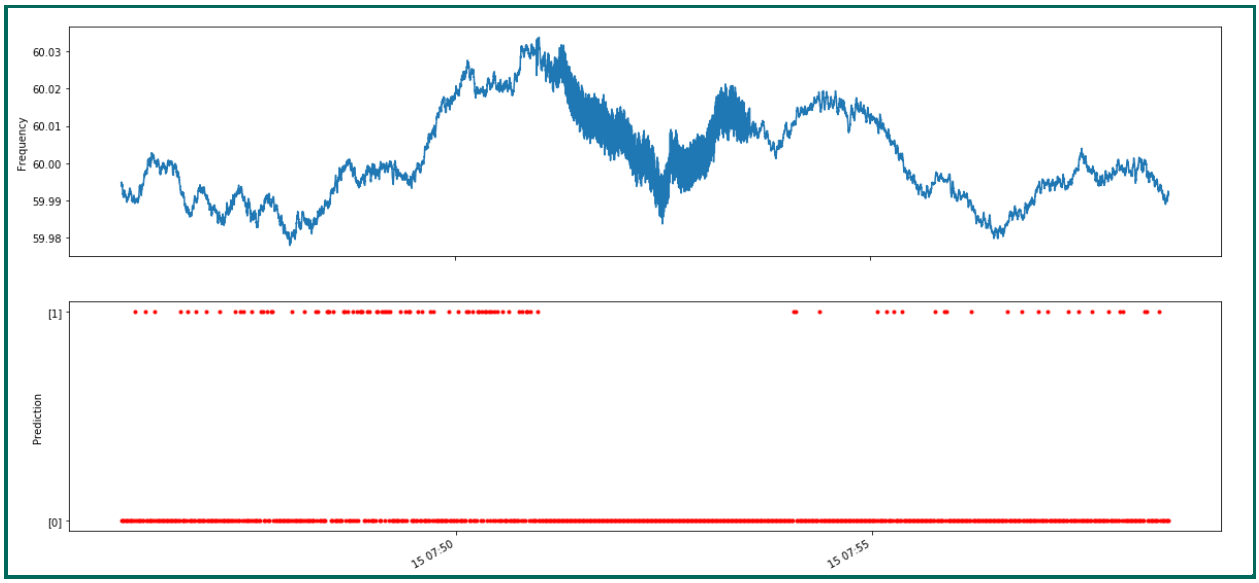


Figure 4.5 Prediction Using the Model Trained by Python Data

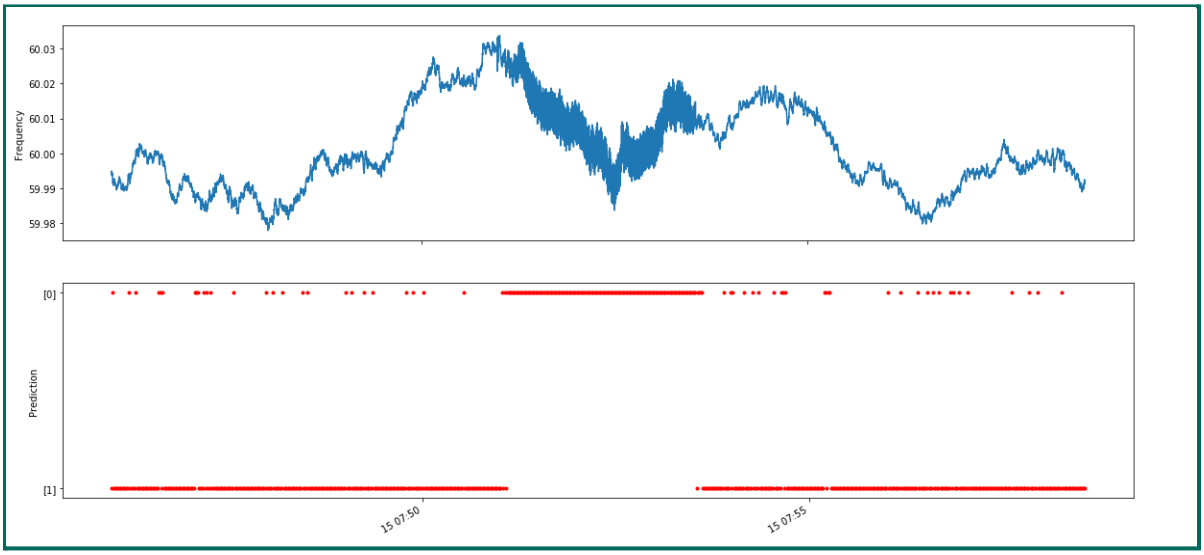


Figure 4.6 Prediction Using the Model Trained by Analog Discovery Board Data

5. Conclusion:

In conclusion, this project constructs a replaceable power system model in Dymola with noise injection, which induces a forced oscillation into the power system. This project also implements Dymola Python interfaces to achieve an automatic generation of data for ML training. Compared with the data generated by Python and Analog Discovery Board, the data generated by Dymola through stochastic simulation is able to emulate the real-world situation and is more suitable for ML training. More importantly, the data generated by Dymola is unlimited and easy-to-access. By doing experiments, the accuracy of the ML model is improved by generating more and more training data.

6. References:

[1] “A PMU-Based Machine Learning Application for Fast Detection of Forced Oscillations from Wind Farms”, Accessed March 25, 2021 [Online]: https://www.notion.so/A-PMU-Based-Machine-Learning-Application-for-Fast-Detection-of-Forced-Oscillations-from-Wind-Farms-1784b9aeddc3445f988422a988ccb383

[2] “Multi-Domain Modelica Models for Gas Turbine Detailed Model Analysis using Modelica, OpenIPSL and ThermoPower”, Accessed March 25, 2021[Online]: https://github.com/ALSETLab/2018_AmericanModelicaConf_PowerGrid_plus_PowerSystems#multi-domain-modelica-models-for-gas-turbine-detailed-model-analysis-using-modelica-openipsl-and-thermopower